IBM

WebSphere Version 4 Application Development Handbook

Complete guide for WebSphere application development

How to make the best use of available tools

Product experts reveal their secrets

Ueli Wahli
Alex Matthews
Paula Coll Lapido
Jean-Pierre Norguet

Redbooks

ibm.com/redbooks

International Technical Support Organization

WebSphere Version 4 Application Development Handbook

September 2001

SG24-6134-00

Take Note! Before using this information and the product it supports, be sure to read the general information in “Special notices” on page 569.

First Edition (September 2001)

This edition applies to Version 4 of IBM WebSphere Application Server, WebSphere Studio, and VisualAge for Java, for use with the Windows NT and Windows 2000 Operating System.

Comments may be addressed to:
IBM Corporation, International Technical Support Organization
Dept. QXXE Building 80-E2
650 Harry Road
San Jose, California 95120-6099

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.

Contents

Preface

This IBM Redbook provides detailed information on how to develop Web applications for IBM WebSphere Application Server Version 4 using a variety of application development tools.

The target audience for this book includes team leaders and developers who are setting up a new J2EE development project using WebSphere Application Server and related tools. It also includes developers with experience of earlier versions of the WebSphere product, who are looking to migrate to the Version 4 environment.

This book is split into four parts, starting with an introduction, which is followed by parts presenting topics relating to the high-level development activities of analysis and design, code, and unit test. A common theme running through all parts of the book is the use of tooling and automation to improve productivity and streamline the development process.

- In Part 1 we introduce the WebSphere programming model, the application development tools, and the example application we use in our discussions.
- In Part 2 we cover the analysis and design process, from requirements modeling through object modeling and code generation to the usage of frameworks.
- In Part 3 we cover coding and building an application using the Java 2 Software Development Kit, WebSphere Studio Version 4, and VisualAge for Java Version 4. We touch on Software Configuration Management using Rational ClearCase and provide coding guidelines for WebSphere applications. We also cover coding using frameworks, such as Jakarta Struts and WebSphere Business Components.
- In Part 4 we cover application testing from simple unit testing through application assembly and deployment to debugging and tracing. We also investigate how unit testing can be automated using JUnit.

In our examples we often refer to the PiggyBank application. This is a very simple J2EE application we created to help illustrate the use of the tools, concepts and principles we describe throughout the book.

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, San Jose Center.

Ueli Wahli is a Consultant IT Specialist at the IBM International Technical Support Organization in San Jose, California. Before joining the ITSO 17 years ago, Ueli worked in technical support at IBM Switzerland. He writes extensively and teaches IBM classes worldwide on application development, object technology, VisualAge products, data dictionaries, and library management. Ueli holds a degree in Mathematics from the Swiss Federal Institute of Technology.

Alex Matthews is a Consulting IT Specialist in the IBM Software Business, based in London, United Kingdom (UK). He has spent the last two and a half years providing post-sales services to customers who have purchased WebSphere products and related tools. Alex has seven years experience building distributed systems using a variety of middleware products. He holds a degree in Computing Science from Aston University, Birmingham, UK.

Paula Coll Lapido works as an IT Specialist in the e-business Innovation Center at Madrid, Spain. Her current area of expertise focuses on developing e-business applications using the WebSphere platform. She has been working at IBM for one year and a half. She holds a degree in Physics from the Complutense University of Madrid.

Jean-Pierre Norguet is an IT Specialist, Team Leader and Coach in the IBM e-business department in Belgium. He has been working at IBM for three years. His areas of expertise include the entire application development life cycle. He holds a 5-year Engineering degree in Computer Science from the Universite Libre de Bruxelles and a Socrates European master's degree from the Ecole Centrale Paris.

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

David Artus	IBM Software Business, London Solutions Group, UK
Keys Botzum	IBM WebSphere Services, Pittsburgh, USA
Kyle Brown	IBM WebSphere Services, Raleigh, USA
Peter Van Sickel	IBM WebSphere Services, Pittsburgh, USA

Special notice

This publication is intended to help application analysts and developers to create Web applications for WebSphere Application Server using a variety of application development and test tools. The information in this publication is not intended as the specification of any programming interfaces that are provided by WebSphere Application Server. See the PUBLICATIONS section of the IBM Programming Announcement for WebSphere Application Server for more information about what publications are considered to be product documentation.

IBM trademarks

Comments welcome

Your comments are important to us!

We want our IBM 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.

Part 1

Introduction

In this part we introduce information used throughout the rest of the book. In particular we describe:

- The programming model and application architecture appropriate for a WebSphere application
- The new features in the latest releases of WebSphere Application Server, WebSphere Studio and VisualAge for Java
- The PiggyBank application used to illustrate our examples

1

WebSphere programming model

This chapter outlines the programming model used to develop applications targeted for the IBM WebSphere Application Server Advanced Edition Version 4.

Characteristics of the programming model

For a programming model to be compelling, we must be able to use it to develop applications that exhibit the following qualities:

- Functional—satisfies user requirements
- Reliable—performs under changing conditions
- Usable—enables easy access to application functions
- Efficient—uses system resources wisely
- Maintainable—can be modified easily
- Portable—can be moved from one environment to another

Furthermore, the programming model must support a development process that has the following characteristics:

- Repeatable—has well-defined steps
- Measurable—has well-defined work products that result
- Toolable—has well-defined mapping of inputs to outputs
- Predictable—can make reliable estimates of task times
- Scalable—works with varying project sizes
- Flexible—can be varied to minimize risks

The challenge is to balance both sets of requirements while developing an application.

Architectures supported by WebSphere

WebSphere Application Server supports three basic application architectures:

- Web-enabled client/server
- Distributed object-based
- Web-enabled distributed object-based

We will discuss each in terms of its features, along with advantages and disadvantages to consider when making a decision about which pattern is most appropriate for your application. Of course, any large system will likely use all of the patterns discussed here, so understanding the trade-offs and when that pattern best applies is key to choosing the application architecture.

Web-based client/server applications

Web-based client/server applications have a “thin” client tier where a Web browser executes, a “middle” tier that runs the Web application server (such as WebSphere), and a “back-end” tier that hosts servers accessible to the entire enterprise, such as databases, and global directories.

The primary purpose of the Web browser is to display data generated by the Web application server components and then trigger application events on behalf of the user through HTTP requests. The data roughly corresponds to the static model associated with the application flow model states.

The Web application server’s purpose is likewise twofold: it controls the application flow in response to HTTP requests sent by the client. As noted in the previous section, transitions on the application flow model will trigger transitions on an underlying business process model. The business logic associated with the business process model (BPM) transition may access data and functions from enterprise servers.

An enterprise server’s main purpose is to provide access to the data associated with BPM transitions. In some cases, business process functions may be delegated to enterprise servers (such as CICS transactions). The protocol used will depend on the back end.

Figure 1-1 shows the relationship between these three tiers in a graphical fashion, indicating the system components normally hosted on that tier along with the primary protocol by which it communicates with the other tiers (the ‘???’ label on the connection indicates that there are possibly many different ones depending on the system).

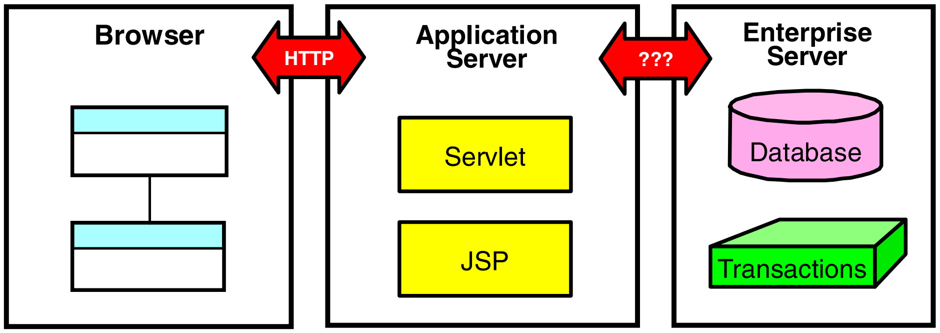


Figure 1-1 Web-enabled client/server architecture

Table 1-1 shows some of the advantages and disadvantages of Web-enabled client/server applications.

Table 1-1 Web-enabled client/server applications

Advantages	Disadvantages
► There is no need to install anything specific on the client tier, because the pages are rendered by a Web server or Web application server and passed back as part of a request. ► The end-to-end path length is relatively short (compared to the other supported architectures), because the Web application server components have direct access to the enterprise servers. ► HTTP connections are stateless, making it possible to scale to large numbers of clients, especially when load-balancing routers are employed. However, we should note here that a common function provided by Web application servers is to provide “state” for the application. Utilizing this function can reduce the benefits of statelessness (more on this point later).	► Controlling the application flow in the Web application server rather than in the client will have an impact on response time, making it crucial to minimize the number of HTTP requests from the browser. ► Components controlling the business process model must be installed on the Web application server as well as client code to the enterprise servers upon which these components depend, which makes maintenance much more difficult (especially if a number of servers are needed to handle the HTTP traffic). ► Having both the application flow and business process logic executing in the same Web application server can increase the processor and memory requirements of the host machines, which may impact throughput. ► Having the business process logic executing in the same tier as the Web application server can be considered a security risk, especially if the Web application is within the “demilitarized zone” (servers outside of the firewall). ► Also, having both the application flow and business process logic executing in the same Web application server makes it difficult to share the business logic with non-Web-enabled clients.

Distributed object-based applications

Distributed object-based applications supported by WebSphere are characterized by:

- An application client tier that controls both the application flow and associated data display through applets.
- One or more servers that host distributed objects encapsulating the logic associated with the business process model.
- One or more back-end enterprise servers that maintain the data associated with the business process model.
- Communication between the client and distributed object server tiers is achieved through the Internet Inter-ORB Protocol (IIOP).

Figure 1-2 shows a graphical view of a distributed object-based application architecture.

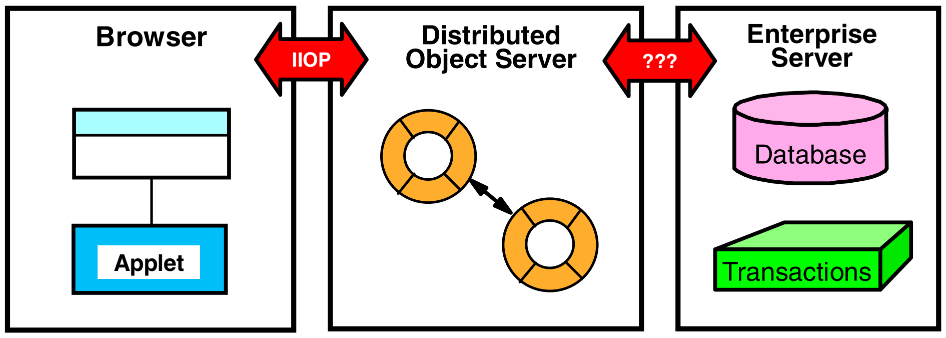


Figure 1-2 Distributed object-based architecture

Distributed object-based applications are considered to have “n” logical tiers because distributed objects can actually be clients of other distributed objects. The tiers are logical because the distributed objects can be co-deployed on the same physical tier.

Table 1-2 shows some of the advantages and disadvantages of distributed object-based applications

Table 1-2 Distributed object-based applications

Advantages	Disadvantages
► Controlling the application flow on the client tier usually makes for snappier response time, especially where heavily used data is cached locally. ► The business logic is separated from the application client, providing for better security and maintainability. ► Having the business logic separated means that it can be shared by multiple clients. ► It is also possible to load balance across multiple distributed object servers to get higher throughput and system availability. ► The application clients do not have to install the client code associated with enterprise servers.	► Application programs must be explicitly installed on the client tier, making maintenance a consideration. This can also increase the processor and memory requirements of the client machines. ► There is extra path length incurred by adding a distributed object server between the client, which will have an impact on response time.

Web-enabled distributed object applications

A powerful feature of the WebSphere programming model is that these two styles can be used together in a single application architecture, such as one where the Web application server components make use of distributed objects that encapsulate the business process logic. This style of architecture can be considered to be a Web-enabled distributed object-based application (Figure 1-3).

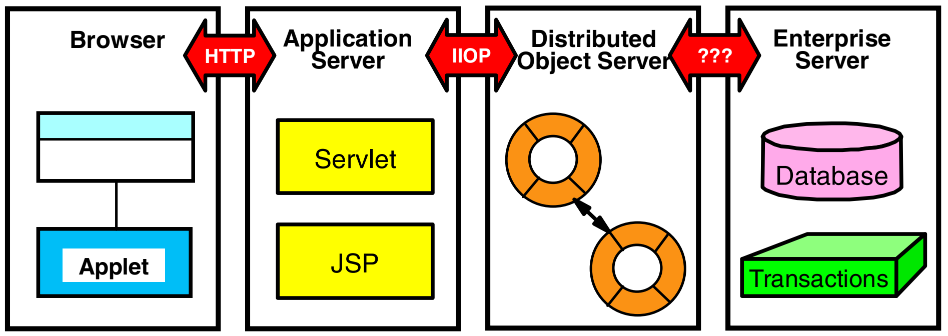


Figure 1-3 Web-enabled distributed object architecture

Table 1-3 shows some of the advantages and disadvantages of Web-enabled distributed object applications.

Table 1-3 Web-enabled distributed object applications

Advantages	Disadvantages
► There is no need to install anything specific on the client tier, because the pages are rendered by a Web server or Web application server and passed back as part of a request. ► HTTP connections are stateless, making it possible to scale to large numbers of clients, especially when load balancing routers are employed. ► The business logic is separated from the application client, providing for better security and maintainability. ► Having the business logic separated means that it can be shared by multiple clients. ► It is also possible to load balance across multiple distributed object servers to get higher throughput and system availability. ► The Web application servers need not install the client code associated with enterprise servers.	► Controlling the application flow in the Web application server rather than in the client will have an impact on response time, making it crucial to minimize the number of HTTP requests from the browser. ► There is extra path length incurred by adding a distributed object server between the client, which will have an additional impact on response time. This impact makes it crucial to minimize the number of distributed object requests from the Web application server.

We will look at our sample application in terms of this hybrid architecture, because it covers all the features of the programming model by WebSphere Application Server.

Features of a programming model driven design

Once we have the candidate architecture identified, the next step is design, where we map the requirements specified in the analysis phase to programming model features associated with the architectural tiers.

All programming models, regardless of the architectural tier, have three distinct features that are key to developing an application:

- The components that embody application functions
- Control flow mechanisms used to invoke one component from another

- Data flow sources that you can use to pass information from one component to another

Each of these features will be discussed in a separate section with the following information:

- A basic definition of the component or mechanism
- The role it plays in the architecture
- Some pros and cons as to its usage
- Alternative approaches, if any exist

Together these sections provide an end-to-end overview of how the components and mechanisms (services) can be used together effectively to develop a WebSphere-based application. Individual chapters that follow will get further into the details of how WebSphere supports the various APIs (which will drive the code phase), and what you can do at deployment time to exploit the WebSphere platform.

Application components

Application components are those that a developer will actually have to program, whether manually or with the aid of tools. The other features of the programming model represent services that the developer can use when coding an application component. The language used to develop a given application component will depend in large part upon the "tier" where the component will be executed at runtime.

For example, browser-based components will tend to use tag and script-oriented languages, while Web application server components will tend towards Java. Enterprise server components may use a variety of languages other than just Java, such as C, C++, COBOL and the like, so we will focus on the distributed object server, which tends towards Java as the language of choice.

Because the language differences tend to divide along tier boundaries, we will divide this section into three separate subsections as we describe the components you develop that are hosted by browsers, Web application servers, and distributed object servers.

We will discuss the components for each tier in turn.

Browser-hosted components

While a browser is not provided by WebSphere Advanced Edition, browser-hosted components make up a large part of any Web-enabled application. The reason, of course, is that the browser serves as the runtime engine for the user interface of a Web application.

The browser-hosted components that are most relevant to the WebSphere programming model include:

- HTML
- Dynamic HyperText Markup Language (DHTML) and JavaScript
- Framesets and named windows
- eXtensible Markup Language (XML), XML Style Language (XSL) and Document Type Definition (DTD)

We will discuss each in turn.

HTML

HyperText Markup Language (HTML) is the basic “programming language” of the browser. With HTML, you can direct the browser to display text, lists, tables, forms, images, and just about everything else you can think of.

Role in the architecture

Every state in a Web application will ultimately result in an HTML page or dialog of some sort, However, we need to draw the distinction between static and dynamic content in an HTML page:

- Static content does not change based on application events, but merely provides access to other states of the application.
- Dynamic content is generated by applications—in many cases based on database content—or from enterprise servers.

The reason that this distinction is important is that static HTML pages do not require that the content be generated by programmatic means, such as Web application components hosted within WebSphere (servlets and JSPs). These components will be discussed in the next section.

Pros	Cons
Static HTML Web pages are not generated by Web application components, such as servlets and JSPs. Their static nature means that they can be cached by either the browser or proxy servers. On the development side, they can be created and maintained with a WYSIWYG (what-you-see-is-what-you-get) editor.	Static HTML cannot be customized on the fly based on customer preferences or application events. Even pages that may seem to be “naturally” static, such as the Customer Home, might actually benefit from being generated dynamically. For example, you might limit the functions that a Customer sees based on the class of service for which they are registered.
Alternatives As mentioned above, the “programming language” of the browser is mainly HTML (with DHTML and JavaScript being the primary exception as described next). However, an XML-enabled browser can be used to generate the HTML on the client side. Finally, you should consider creating dynamic components for every “top level” (non-dialog state), even if it appears to be static. This approach not only makes it easier to add dynamic content later, but also makes it easier to compose into other pages.	

DHTML and JavaScript

Dynamic HyperText Markup Language (DHTML) is an extension to HTML wherein all the components of the HTML page are considered to be objects. Together these objects make up the Document Object Model (DOM) of the page.

Each object in the DOM has a set of associated attributes and events, depending on the type of object. For example, most objects have attributes describing their background and foreground colors, default font, and whether they are visible or not. Most have an event that is triggered when the object is loaded into the DOM or displayed. An object, such as a button, has attributes that describe the label and events that fire when it has been pressed.

Events are special because they can be associated with a program that executes when the event is triggered. One language that can be used for the program is JavaScript, which is a scripting language with Java-like syntax. JavaScript can be used to change the attributes of objects in the DOM, thereby providing limited control of the application flow by the browser.

Role in the architecture

This ability makes DHTML/JavaScript perfect for handling confirmations, data validations, cascading menus, and certain types of list processing on the browser side without invoking an HTTP request to the Web application server.

Where validations are concerned, it is important to draw the distinction between those that are merely syntactic from those that are more semantic in nature:

- Syntactic validations include checks on individual fields of an input form. For example, is the entry a minimum length? Is it alpha or numeric? Does it have the right format for a date, phone number or social security number? These simple types of syntactic validations should be done on the client.
- Semantic validations are those that ultimately require access to business process logic and data. For example, is an order or product number valid? Will the change in quantity make the resulting line item quantity less than zero? Is the requested price within 10 percent of the current average? Semantic validations belong on the server side.

In the middle ground are more complex syntactic validations that involve multiple fields or begin to incorporate business process policies. For example, is the start date less than the end date? Does the date requested fall on a weekend or holiday? There are arguments both for and against handling complex syntactic validations on the client side. The most forceful arguments against are that it introduces extra complexity and redundancy in the DHTML, and can cause a maintenance problem as policies change.

Pros	**Cons**
Hopefully, the benefit of using DHTML and JavaScript in these scenarios is obvious: one or more round trips to the Web application server are eliminated, making the application both more efficient and more usable (mainly because the response time is much snappier).	Using DHTML/JavaScript for application control flow, whether it is on the client or server side, requires programming skills and are more complicated to develop and test. You cannot use WYSIWYG editors for the code. There are differences among the browsers in the details of the functions supported. To avoid a browser dependency for the Web application, programmers are forced to either stay with a common subset of functions or add branching logic and optimize for each browser. When syntactic validations (either simple or complex) are handled in DHTML and JavaScript, you still have to revalidate on the server side for each request just in case the client circumvents the input forms. This leads to redundancy of code on the client and server.
Alternatives Really, there is no good alternative to DHTML and JavaScript for handling confirmations, validations, menus, and lists. The complexity for the HTML developer can be managed somewhat by having a separate programming group develop a set of common functions that encapsulate the differences between the browsers and have every page designer include this set of functions within their HTML.	

Framesets and named windows

Framesets and named windows are specialized HTML extensions that break up a page into separate frames (for framesets) or windows (for named windows). Each frame or window can be further subdivided as a frameset as well.

Various browser-initiated control flow actions (described in , “Browser component initiated control flow” on page 40) can be targeted to a given frame or window, leaving the other frames and windows untouched.

The main difference between framesets and named windows is that framesets tile the various frames within a single browser window, while named windows have a separate window for each unique name. Frames in a frameset can have various attributes that define behaviors such as whether they are resizable, scrolling, or have borders. Separate named windows can be cascaded or manually tiled by the user as they see fit.

From a targeting perspective, there is no difference between framesets or named windows. In fact, they can be used together. If no frame or window with a given name is open already, one will be opened by the browser to receive the result of the request. The opened windows can be resized and tiled manually to achieve an effect very similar to framesets.

Role in the architecture

Framesets are an excellent way to group related states in the application flow model. For example, an online buying application Web page could be implemented as a frameset that includes the following three frames (or windows):

- Navigation, an area that is populated with the Customer Home navigation links
- Main, an area that is populated with the Product Catalog, Order Status or Order Details data, depending on the link selected in the Navigation frame. This area would also be the target of an “open new order” action in the Order Status state, so it would possibly be populated with the Already Open page.
- Result, an area that displays the result of an add to order, modify line item, submit, or cancel operation.

Figure 1-4 shows a stylized view of how this page might look using framesets.

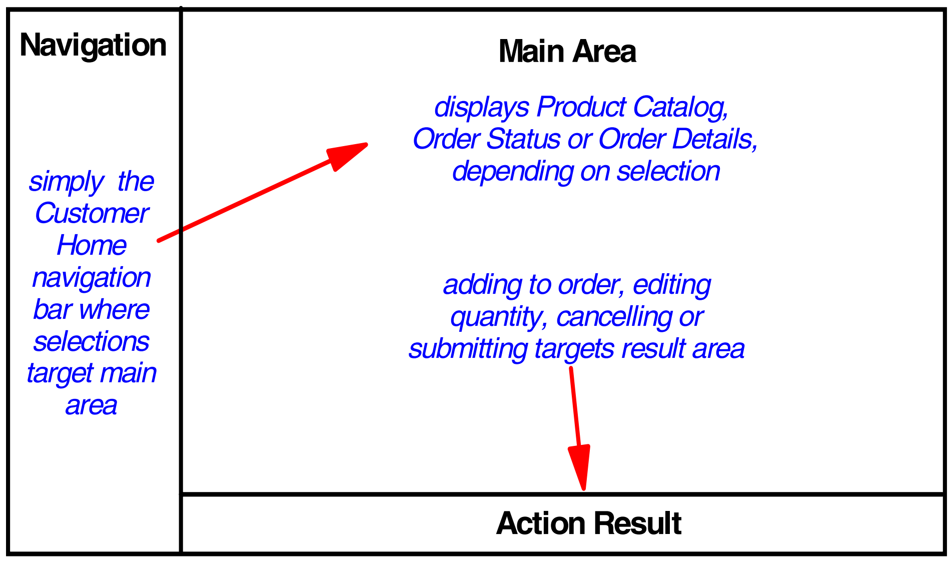

Navigation
simply the Customer Home navigation bar where selections target main area
Main Area
displays Product Catalog, Order Status or Order Details, depending on selection
adding to order, editing quantity, cancelling or submitting targets result area
Action Result

Figure 1-4 Stylized view of online buying application frameset

Although not explored in any more detail here, a frameset makes it easy to mingle Web publishing and business applications together. In this approach, you provide visual interest such as images, advertisements, news, and such in the “surrounding” frames, and keep the frames associated with the business of the application clean, simple, and most importantly fast (because they can be mostly text based).

Pros	Cons
Simplifies navigation—the home state is always visible. Maximizes visibility of the important data and functions. The main area can display long lists for scrolling. Minimizes the size of an individual request. Only the data required for the target area is returned from a given request. The Navigation area need never be rerendered. Improves the application flow and efficiency. Error messages can be displayed with the form data still available. Parallelizes requests. When a frameset is rendered, each frame is issued as an individual request, allowing them to be handled and displayed separately. Hides potentially “ugly” URLs of the individual frames.	Improperly designed, the navigation can be confusing. Also, if more than one frame accesses shared system resources, such as HttpSession state or databases, it can cause contention problems that affect performance, and may even cause deadlocks. When printing within a frameset, maybe only the “active” frame (usually where the cursor is located when the print is requested) is printed. Bookmarking a frameset uses the browser location line, and not the specific content frame URLs. Browser back and forward functions work a frame at a time. This can be somewhat disconcerting. Probably the most serious disadvantage is that not all browsers support framesets, so a non-frame version must be provided if the application is designed to be browser independent.
Alternatives Before we abandon framesets because of the disadvantages mentioned above, there are some workarounds to consider: Printing: develop explicit print functions. Bookmarking: maintain the last page in a database. Backward/forward: disable the back and forward buttons on the browser. Browser support: named windows instead of framesets. If these workarounds cannot be used in your Web application, the only real alternative to framesets is to compose the pages representing the individual states, and pay the cost of rerendering the entire page on every request.	

XML, DTD and XSL

XML provides a means by which documents can be encoded as a set of tags describing the associated values. The tag language is expressive enough that tags can be nested and can repeat, so that complex data structures can be encoded in a form that is both human and machine readable.

An XML document can be associated with a DTD, which defines the tags and the structure of the tags in the XML file. A DTD can be used by an XML parser to validate that the XML is not just well formed syntactically, but is also semantically legal with respect to the DTD. XML schemas will replace DTDs in the future; they provide support for stronger typing of the data values.

Finally, more and more browsers are becoming XML enabled. XML-enabled browsers can handle XML documents returned from the Web server in response to a request. The XML document can refer to an associated stylesheet coded in XSL. The stylesheet is used by the browser to map the XML tags to the HTML that is ultimately displayed. If no stylesheet is specified, the browser will use a default format that takes advantage of the tag names.

Role in the architecture

XML can play a role in every tier of the application architecture. For a Web-enabled browser tier, the response to a given request can be an XML document containing only the data to be displayed. For example, we could build XML documents representing the data described for each state and provide a default stylesheet in XSL to map the data to HTML tables and forms.

Pros	**Cons**
One advantage of using XML rather than HTML is that the stylesheet can be modified to change the look and feel without having to change the Web application components (described later) that generate the data. Another advantage is that the size of the result will be smaller than the resulting HTML in many cases. Yet another advantage is that the same XML document may be usable in other contexts than a Web browser, making it possible to reuse the Web application components.	The main disadvantage is that XML-enabled browsers are not yet available every where, although they are rapidly becoming so. Another disadvantage is that XSL-based stylesheets can be quite complex to code and difficult to debug. WYSIWYG editors for XML/XSL are not yet widely available either.
Alternatives One alternative is to have the Web application components check the browser type and either generate HTML for non-XML-enabled browsers or return the raw XML for XML-enabled browsers. The next subsection will discuss this idea further.	

Web application server hosted components

In the previous section, we discussed how HTML is the ultimate programming language for the browser tier, but drew a sharp distinction between static and dynamic content for Web pages.

We also discussed how a browser is not specifically provided by the WebSphere platform. This is not the case for the Web server and Web application server. WebSphere provides the IBM HTTP Server as a Web server that can be used to serve up static pages, but can be configured to use other popular Web servers from Microsoft and Netscape, among others.

Of course, the focus of this section is the WebSphere Application Server used to serve up dynamic pages.

By discussing HTML, DHTML, JavaScript, framesets and XML, we have already covered the static components of the programming model. The Web application server components hosted by WebSphere that are most useful in generating dynamic content include:

- Servlets
- JavaServer Pages (JSPs)

While no special support is provided by WebSphere Application Server, there are two other components that are useful for clients (including Web applications) of business logic and data hosted on back-end servers:

- Data structure JavaBeans
- Business logic access beans

Together these components provide the basis for a very effective model-view-controller (MVC) architecture, where data structure and access beans represent the business process model (model), servlets control the application flow (controller), and JSPs handle the layout (view).

An MVC architecture is effective because of the ability to independently develop, test, deploy and modify the various components. We will discuss each of these four components in the context of an MVC architecture in the subsections to follow. See “Model-view-controller pattern” on page 87 for more information.

Servlets

For purposes of understanding the programming model, you develop servlets to encapsulate Web application flow of control logic on the server side (when it cannot be handled by DHTML on the client side).

An `HttpServlet` is a subclass of a generic Java servlet. Most people mean `HttpServlet` when they say servlet, but there is a difference. An `HttpServlet` is specifically designed to handle HTTP requests from a client. In this redbook, we call it "servlet" unless we need to distinguish them.

The `HttpServlet` Java class from which you will inherit (extend) has a number of methods that you can implement that are invoked at specific points in the life cycle. The most important ones are:

- `init`, executed once when the HttpServlet is loaded
- `service`, by default calls `doGet` or `doPost`, unless overwritten.
- `doGet`, executed in response to an HTTP GET request
- `doPost`, executed in response to an HTTP PUT request
- `destroy`, executed once when the HttpServlet is unloaded

The service type methods (for example, `doGet` and `doPost`) are passed two parameters: an `HttpServletRequest` and an `HttpServletResponse` object, which are Java classes that encapsulate the differences among various Web servers in how they expect you to get parameters and generate the resulting HTML page.

Role in the architecture

Servlets are designed from the ground up to handle dynamic requests from an HTTP client. In an MVC architecture, servlets represent the controller component.

However, there is a question of granularity that needs to be addressed. That is, how many servlets are required to control a Web application?

At one extreme, there are those that create only one servlet to control the entire application (or worse, they may only build one servlet, ever). The `doGet` or `doPost` methods use a parameter (or the URI) from the request object to determine the action to take, given the current state. Possible shortcomings are:

- Unmaintainable, when implemented as a large case statement.
- Redundant with other approaches described next, when implemented by forwarding to an action-specific servlet or JSPs (you might as well route the request directly to the appropriate servlet).
- Redundant with the servlet APIs themselves, when implemented by loading an action-specific functional class (the class invoked has to look just like a servlet, with request and response objects).
- Security for a given function must be manually coded rather than use per servlet security provided by the WebSphere administration tools.

Is is possible to analyze the URI and execute a command (that has a matching name) for processing of the request. Frameworks, such as Jakarta Struts generate that kind of code.

At the other extreme of the granularity spectrum is one servlet per action. This is a better approach than a single servlet per application, because you can assign different servlets to different developers without fear that they will step on each other's toes. However, there are some minor issues with this approach as well:

- ► Servlet names can get really long to insure uniqueness in the application.
- ► It is more difficult to take advantage of commonality between related actions without creating auxiliary classes or using inheritance schemes.

In the middle is to develop a single servlet per state in the application flow model that has dynamic content or actions. This approach resolves the issues associated with the approaches described above. For example, it leads to a "natural" naming convention for a servlet: *StateServlet*. The `doGet` method is used to gather and display the data for a given state, while the `doPost` method is used to handle the transitions out of the state with update side effects. Ownership can be assigned by state. Further, commonality tends to occur most often within a given state and service method type.

Pros	**Cons**
Before the Servlet API became available, each Web application component (usually a CGI program) had to code to a Web server-specific API. Java servlets are very portable and can be used with the leading Web servers. Also, servlets stay resident once they are initialized and can handle multiple requests. CGIs generally start a new process for each request. Servlets can be multi-threaded, making them very scalable.The application server creates a new thread per client. Because servlets are Java programs, they can be developed with an IDE, such as VisualAge for Java.	A minor disadvantage to servlets is that they require explicit compiling and deployment into an application server.
Alternatives You can develop monolithic servlets that handle both the application flow logic and generate HTML, or even go to the extreme of handling business process logic as well. The only advantage of this approach is that the end-to-end path length is shorter. The problem with monolithic servlets is that the layout cannot be developed with a WYSIWYG editor, nor can the business logic be reused in other client types, such as Java applications. Further, it makes it much more difficult to move the application to alternate output media, such as WAP and WML. JavaServer Pages, to be discussed next, are considered by some to be a viable alternative to servlets, because they are functionally equivalent.	

JavaServer Pages

JavaServer Pages (JSPs) are a standard extension to HTML that provide escapes so that values can be dynamically inserted.

There are numerous tags that allow the developer to do such things as import Java classes, and declare common functions and variables. The most important ones used by a JSP developer to generate dynamic content are:

- Java code block (`<% code %>`), usually used to insert logic blocks such as loops for tables, selection lists, options, and so on
- Expressions (`<%= expression %>`), usually used to insert substitute variable values into the HTML.
- Bean tag (`<jsp:useBean>`), used to get a reference to a JavaBean scoped to various sources, such as the request, session, or context.
- Property tag (`<jsp:getProperty>`) is a special-purpose version of the expression tag that substitutes a specified property from a bean (loaded with the `useBean` tag).

There is also a standard tag extension mechanism in JSP that allows the developer to make up new tags and associate them with code that can either convert the tag into HTML or control subsequent parsing (depending on the type of tag created). This feature would allow a developer (or third-party providers) to build tags that eliminate the need to explicitly code expressions and java code blocks, making the JSP code look more HTML-like and less Java like. Custom tags can make it very easy for non-programmers to develop JSPs (those with Java skills can develop specialized tags to generate tables, option lists, and such).

In any event, a JSP is compiled at runtime by WebSphere into a servlet that executes to generate the resulting dynamic HTML. Subsequent calls to the same JSP simply execute the compiled servlet.

Role in the architecture

JSPs are best used to handle the display of data associated with a given state having dynamic content. This role represents the view in an MVC architecture and contrasts with that of the servlet that represents the controller. The way they work together is that the servlet gathers the data or handles the transition action, and then routes flow of control to the associated JSP to generate the response.

Whether extended tags are used or not, we recommend developing JSPs such that multiple states can be composed within a single page (see “HTML” on page 11 and “Framesets and named windows” on page 14 for more details on page composition). This approach actually simplifies the individual JSPs because they need not worry about setting headers or the <HTML><BODY> and other enclosing tags. The associated servlet can handle this setup, or can delegate it to an inherited servlet as discussed in the previous section. This approach will also make it easier to exploit dynamic caching that is supported in WebSphere Application Server Version 4.

Pros	**Cons**
One huge advantage of JSPs is that they are mostly HTML with a few special tags here and there to fill in the blanks from data variables. The standard extension mechanism allows new tags to be developed that eliminate the need to use the Java escape tags at all. Further, JSPs require none of the “println” syntax required in an equivalent servlet. This tag-oriented focus makes them relatively easy to WYSIWYG edit with tools such as WebSphere Studio Page Designer. This focus also makes it easier to assign the task of building JSPs to developers more skilled in graphic design than programming. JSPs can be used to provide meaningful error indicators on the same page as the input fields, including specific messages and highlighting. Static HTML does not provide this capability. Another advantage is that JSPs do not require an explicit compile step, making them easy to develop and test in rapid prototyping cycles. This feature tempts some developers to use JSPs instead of servlets to handle the data gathering and update-transition functions, logic that is traditionally associated with the controller component of an MVC architecture.	There are some good reasons not to use JSPs to control the application flow: ► Current JSP tools do not provide IDE functions for code blocks. ► A developer should not handle the application control flow and the layout. ► Combining application flow and layout in a JSP makes it difficult to migrate to another output media. ► All HTML tags are compiled into the servlets service method. This makes inheritance of common look-and-feel behaviors in JSPs very difficult. There are some minor issues associated with using JSPs. ► JSPs compile on the first invocation, which usually causes a noticeable response time delay. ► Communication between the JSP and servlet creates a name, type and data flow source convention issue. In other words, how do you pass data elements between a servlet and the corresponding JSP? The next section discusses using a JavaBean to encapsulate the data needed by a JSP.
Alternatives XML provides a viable alternative to JSP in some situations. It is possible to have the servlet for a given state return XML directly to an XML-enabled browser, using an XML parser-generator. Even if a user’s browser does not support XML, the servlet could use the associated stylesheet to generate the corresponding HTML without using a JSP. We will discuss this possibility further in the next section, where JavaBeans can be employed to simplify this process.	

If you insist on using JSPs to control the application flow, we recommend building two per state:

1. `StateServlet.jsp`, playing the role of servlet with nothing but a script tag implementing `doGet` and `doPost` type methods, It can safely inherit from a superclass `HttpServlet` as described in the previous section.
2. `State.jsp`, playing the role of an output JSP as described in this section.

This approach allows you to take advantage of the quick prototyping capability of JSPs early in the development cycle (no compile or deploy step needed). Later on you could convert the “servlet” JSP to a real servlet (to avoid the need to precompile the JSPs as described above).

However, we should say here that such tools as VisualAge for Java Enterprise Edition with its embedded WebSphere Test Environment provide the ability to rapidly develop and test servlets as easily as JSPs, minimizing the development cycle-time advantage described above that might motivate the use of JSPs for application flow control.

Data structure JavaBeans (data beans)

A JavaBean is a class that follows strictly specified conventions for naming properties, events and methods. An auxiliary class, called a `BeanInfo` class, contains additional descriptive information that can be used by tools to provide, among other things, extra levels of documentation and runtime support to edit property values.

A data structure JavaBean is usually nothing but a simple set of properties, with no need for events or methods (beyond gets and sets of the associated properties).

Data structure JavaBeans are sometimes made “immutable”. That is, all properties are private and only get methods are provided to prevent the data from being updated. Also, data structure JavaBeans sometimes are associated with a separate key subcomponent that encapsulates those properties that uniquely identify the associated data.

Immutable or not, key or not, a data structure JavaBean should implement the serializable interface that enables it to be passed remotely and stored in various files and databases. An implication of being serializable is that the object properties must be simple types or strings, or that any contained objects must be serializable.

Note: Data structure beans are also called cargo beans, value beans, and other names.

Strictly speaking, WebSphere Application Server has no special support for JavaBeans. However, data structure JavaBeans fill so many useful roles in the end-to-end architecture that we feel required to include them in a discussion about the programming model.

Role in the architecture

In an MVC architecture, data structure JavaBeans can be considered to represent the static properties associated with objects in the model. This makes them useful to maintain data reads from back-end systems, or results from executing back-end business functions (more on this in the next section on business logic access beans).

For purposes of the Web application server tier, we also see them used to maintain the data passed between the servlet and other middle-tier components, especially JSPs (described in “JavaServer Pages” on page 21) when there is more than one property involved. They may represent data from the model as it is transformed for a specific view associated with a JSP, or as occurs in many cases, it may be that the model object does not need transforming and can be passed to the JSP as is.

Some developers build a data structure JavaBean for every JSP whether it has more than one property or not, and whether or it is associated with a servlet or not. They may also make these data structure JavaBeans immutable, as described above, to make them easier to deal with in WYSIWYG editors (only get methods would show in the palette of functions available).

Tip: Consider the usa of view beans that provide an interface between data structure beans and JSPs. View beans will format the values contained in data beans into strings that are easily accessible by JSPs.

Some XML enthusiasts propose XML as a dynamic substitute for explicitly coded JavaBeans (see “XML, DTD and XSL” on page 16). With this approach, a single XML string is passed or stored rather than a data structure JavaBean. The receiving component then uses the XML parser to retrieve the data.

While we are strong proponents of XML, and see its merits as a possible serialized format of a data structure JavaBean, we would not recommend using XML-encoded strings as a substitute, especially in situations where the data structure is known at design time.

The extra overhead of generating and parsing the XML strings, plus the storing, retrieving and transmitting of all the extra tags, makes them very expensive with respect to the equivalent data structure JavaBean.

Pros	Cons
The data structure JavaBean represents a formal contract between the servlet and JSP developer roles involved. Adding properties is easy because the servlets already create, populate, and pass the associated JavaBeans, while the JSPs already use the bean and property tags. You can independently modify the programs to use the new properties. Also, the new properties can be optional with a default assigned as part of the constructor. Removing a property from the contract without modifying the associated servlets and JSPs that use them will cause errors to be caught at compile rather than runtime. It allows the servlet developer for a given state to focus entirely on Java to control the application control and data flow, while the JSP developer can focus entirely on HTML or XML-like tags that control the layout. Many tools are available that take advantage of JavaBean introspection for such varied functions as providing command completion and selection of properties from a palette at development time, to populating from and generating XML at runtime. Setting properties into a data structure JavaBean, and then setting the whole data structure into a data flow source (such as HttpServletRequest attributes to be discussed in "HttpServletRequest attributes" on page 49) is much more efficient than setting individual properties into that source one at a time. And getting a single data structure JavaBean from that same source, and then getting its properties locally, is much more efficient than getting multiple properties directly from the source. The same data structure JavaBeans are likely to be used as contracts with other components because of their simplicity, providing a high degree of reuse.	There are no serious disadvantages to using data structure JavaBeans. The only issue is that they can be rather expensive to create, and may cause extra garbage collection cycles as memory is used. To circumvent this problem, some developers use pooling techniques, where a number of pre-constructed JavaBeans wait to be requested, used, and then released back to the pool. Data structure beans can be tedious to develop, although in many instances tools will generate the data structure beans.
Alternatives There is honestly no good alternative to using data structure JavaBeans as the formal contract between components in the architecture. And, as we will see in the following sections, data structure JavaBeans are used just about everywhere, making them well worth the investment.	

Business logic access beans

We noted in the previous subsection on data structure JavaBeans that they represent the static properties of the model. In the same vein, a business logic access bean can be thought of as encapsulating the dynamic behavior of the model.

A business logic access bean is a Java class whose methods encapsulate a unit of work needed by any type of application, be it for the Web or a distributed client/server. In other words, a business logic access bean is intended to be user interface independent.

In our sample application we mapped each use case to a business logic access bean.

Note: Access bean as it is used here is intended to be a generic Java wrapper, and not to be confused with the specific kind of access beans generated by VisualAge for Java Enterprise Edition.

The other primary purpose of the business logic access bean is to insulate the client from various technology dependencies that may be required to implement the business logic.

Business logic access beans will almost always make use of data structure JavaBeans and associated keys in the input and output parameters. Further, any data cached within an access bean is likely to be in terms of data structure JavaBeans and associated keys, so the two concepts go hand in hand.

Like data structure JavaBeans, WebSphere Application Server has no special support for business logic access beans, However, they are so useful in the end-to-end architecture that we feel required to include them in this discussion as well.

Role in the architecture

We noted in the previous subsection that data structure JavaBeans represent the static properties of the model in an MVC architecture. In the same vein, a business logic access bean can be thought of as encapsulating the dynamic behavior of the model.

There are numerous approaches to developing business logic access beans, covering many different aspects that may be useful in a given application. We touch on a few of them here, but it is not within the scope of this book to discuss all the different patterns that may be used (see *Design Patterns: Elements of Reusable Object-Oriented Software*, by Erich Gamma, et al).

The first aspect we consider is whether the business logic is stateless or stateful:

- **Stateless** access beans have methods whose input parameters include all the data necessary to complete the unit of work and whose return values have the complete result. Stateless access beans retain no memory of what a given client program has done between invocations. However, “statelessness” does not mean that the access bean cannot cache data, just that any data cached must be accessible using parameters passed in a given method signature.
- **Stateful** access beans have methods that may rely on the result of previous methods, thus having an implied “state” model. Statefulness can be exploited to simplify the method signatures, because parameters or results of previous calls can be cached so that fewer parameters are needed in subsequent method signatures. Stateful access beans must have a one-to-one association with the client so that two different clients in two different “states” do not interfere with each other. This association is often called client/server affinity, and it can make stateful access beans less scalable than an equivalent stateless one by limiting the ability to arbitrarily load balance method calls.

You can sometimes simulate a stateful access bean with a stateless one by including extra parameters that either identify the client or contain the current state data.

The identity is used in the first approach to look up state data cached in the access bean. If the second approach is used, the current state data is used in the called method and a new current state is returned to the client (as a data structure JavaBean, as described in “Data structure JavaBeans (data beans)” on page 23) to keep until the next call.

Another aspect to consider is granularity. Like servlets described in “Servlets” on page 18, there is a continuum of granularity that could be considered when developing a business logic access bean:

- On the one extreme, there could be a single business logic access bean per unit of work.
- On the other, all units of work for the business process could be represented by methods on a single business logic access bean.
- In the middle, all the methods needed for the transitions in a given state in the application flow model could be grouped into a single business logic access bean.
- Also in the middle (but coarser grained), all the transitions associated with a state in the business process model could be represented by methods on a single business logic access bean.

Some of you may be familiar with the following types of business logic access beans:

- A ***command bean*** is a fine-grained stateful business logic access bean that encapsulates a single unit of work. The programming model for a command bean is to create it (or get one from a pool), set properties (its state) that represent the input parameters, perform the unit of work, and finally get the properties representing the return parameters (also part of the state). Refer to the redbook *Design and Implement Servlet, JSPs, and EJBs for IBM WebSphere Application Server*, SG24-5754 for more details.
- A business logic access bean that is stateful and holds the data returned from a query result is sometimes called a ***rowset***. A rowset can be considered to be a specialized form of command bean. Its programming model is to set parameters representing a query, perform the query, then iterate through the results in the rowset.

Pros	**Cons**
One big advantage of wrapping the business logic associated with the business process model is that wrappers provide an encapsulating layer that hides the details of where and how the business logic gets invoked from the client. This insulation will make it possible to evolve the technology used by the application over time. For example, in early stages, the code may make direct calls to JDBC, and then over time migrate to use Enterprise JavaBeans, all without having to recompile the client applications. Another big advantage is that once the method signatures are defined, it is possible to build the client applications in parallel with the back-end business logic, and speed the overall development process. Testing becomes simpler too, because the logic controlling the application flow is cleanly separated from the logic controlling the business process, minimizing the number of code segments that need to be tested.	There is absolutely no downside to wrapping business logic in objects separate from the client, even in cases where there is little or no opportunity to reuse the business logic in other applications.
Alternatives That said, there is an alternative to wrapping business logic into “vanilla” Java classes, and that is to directly use Enterprise JavaBeans in the client code. In the next section, we discuss the various types and applications in the next section that we feel make this a viable alternative.	

Distributed object server-hosted components

The WebSphere programming model provides support for Java-based distributed objects called Enterprise JavaBeans (EJBs). EJBs can be thought of as a standard mechanism to wrapper enterprise business logic and data (usually hosted on some enterprise server) that can take advantage of the following object services:

- Distribution—the ability for the server to be remote from the client
- Persistence—maintenance of the essential data associated with the component
- Transactions—providing ACID characteristics for the units of work
- Security—control of the roles that can access the objects and associated methods
- Trace and monitoring—configurable instrumentation for debugging and performance tuning

There are two main types of EJBs that can be developed as part of the programming model: session and entity. In a nutshell, session EJBs are those that have a short life cycle that lasts only as long as both the client and server maintain a reference to the session. This reference can be lost if the client removes the session, or if the server goes down or the session "times out".

An entity EJB is one that once created (either explicitly or implicitly) can be subsequently found with a "key" for use by a client application. It persists until the remove operation is invoked by a client application or until the data is explicitly removed from the underlying store.

Within a session EJB there are three implementation types: stateless, stateful, and stateful with session synchronization. Within an entity EJB there are two implementation types depending on how persistence is managed: container-managed persistence (CMP), and bean-managed persistence (BMP).

Stateless session EJBs

Stateless session EJBs are those whose method signatures have all the parameters needed to complete the associated unit of work; they return the complete result in the method return value (or exceptions that may be thrown).

The effect of being stateless is that any active instance of a stateless session EJB can service a request from any client in any sequence. This feature makes stateless session EJBs the most scalable.

There is no guarantee that two calls to the same stateless session EJB will be services by the same instance on the server. Because of this feature (good for scalability), there is a common misconception that stateless session EJBs cannot have instance variables and thus maintain “state”. They can, as long as the values can be used by any client, and in any sequence. For example, many applications cache connections to back-end resources and frequently used stable read-only data in stateless session EJBs.

Role in the architecture

Stateless session EJBs are ideal for implementing the business logic associated with the business process model, as we described in see “Business logic access beans” on page 26.

You could use the stateless session EJBs directly by the client program, or have the business logic access bean wrapper call the stateless session bean.

The implementations of the stateless session EJBs can be exactly the same as those provided for the business logic access beans, or they can take advantage of the features of stateless session EJBs.

For example, if the code manually manages a connection pool for a relatively expensive resource, you can cache the connection in the stateless session EJB (as long as it is not client specific). This approach effectively lets the EJB container act as the pooling mechanism, and makes getting the connection transparent to the business logic, which can simply use the connection.

As another example, if the methods managed standard Java Transaction API (JTA) transactions at the beginning and end of the business logic to provide ACID properties, this code could be removed, because it is provided automatically by the container.

Tip: Rather than look up the home in the JNDI context, narrow it, and create the session over and over again for each request, you can create the session once and cache it in the client (either the servlet or, preferably, the business logic access bean). This approach should be considered a “best practice” even though the IBM implementation of the JNDI context in WebSphere Application Server automatically caches homes to provide a high degree of scalability.

Also, it is a common practice to cache stable read-only data in a stateless session EJB (or in an associated singleton object) to minimize repeating expensive computations. For example, we may want to cache the product catalog data within a singleton referenced by the stateless session bean.

Pros	Cons
Besides simplifying the code to handle connection pooling, transactions and security, a key advantage gained when using stateless session EJBs is that the business logic can be moved out of (distributed from) the client tier without having to reprogram the client or server components. This ability can be important for security purposes. For example, we may be happy to have the Web server and servlets within the DMZ, but we would probably want to host the business logic behind the inner firewall of the DMZ to protect it from direct access by hackers. In other cases we may want to co-deploy the business logic with the application flow logic, but put both behind the DMZ. Another advantage of using stateless session EJBs is that it is possible to efficiently load balance them across multiple application servers and achieve a high degree of scalability. Distributing the business logic out of the client tier can make the client much “thinner”, because there is no need to install connectivity options. Only the Enterprise Java Servers would need to maintain the connectivity to the back-end systems. In large Web application server farms, or Java applets, having a thin middle tier can be a very attractive advantage. Finally, distributing the business logic out of the client means that it can be reused in multiple application types, not just Web applications, as we show in the introduction (“Distributed object-based applications” on page 7).	One disadvantage to using EJBs in general is that the overhead (distributed calls, security checks and transaction management) can be quite expensive even when the client and server are co-deployed. For this reason, you must take care to design the EJBs to minimize the number of calls required per unit of work. The second disadvantage to using EJBs is that the need to find a Home in the JNDI context, narrow it to the specific home interface type, and create the remote interface prior to using it, adds complexity to the client programming model. Still a third disadvantage when using EJBs is the increased complexity in testing, debugging, deployment, and administration. Specific to stateless session EJBs, a disadvantage with respect to other EJB types is the need to pass in extra parameters on the call, and receive all the data on the return value. This requirement can significantly increase the data transmission costs, if the objects are not carefully designed. Also, expensive computations may be repeated (if the EJB is called more than once in the logical session), because a stateless session EJB retains no memory of previous calls.
Alternatives If, for example, all the logic is handled by back-end CICS transactions, or all the data is maintained in a single DB2 database using precompiled SQLJ queries, then a simple business logic access bean that directly accesses these back-end systems may be the preferred approach.	

Stateful session EJBs

Stateful session EJBs have a complex life cycle model that allows methods to maintain state between calls. The effect is that a given task can span multiple invocations.

Unlike stateless session EJBs, stateful session beans can support a custom create that takes parameters useful in initializing the state. This feature can be very useful in simplifying the other method signatures, because they can assume that the state of the session EJB includes those parameters useful for the lifetime of the session.

Role in the architecture

Stateful session beans can keep the application/user state over multiple interactions (methods).

Pros	Cons
One benefit of using stateful session EJBs is that the methods map more closely to the transitions associated with the business process model than those of the stateless session EJB (or business logic access bean) described previously. Also, the fewer number of parameters means that there is less data to marshal and demarshal in a remote method invocation. Another benefit of a stateful session EJB is that it can reduce the number of calls to the back end by caching frequently used data as part of its state. Taking this idea to an extreme, stateful session EJBs can cache data considered to be work-in-progress, eliminating all calls to the back end until specific “checkpoint” type transitions. This can be especially advantageous in situations where application events may terminate the processing before its logical conclusion.	The primary disadvantage to using stateful session EJBs is that there are very few quality of service guarantees with respect to the ACID properties you might expect when working with components. The container is not obligated to provide for failover of stateful sessions by backing up the nontransient instance variables in a shared file or database; so in general, if the server hosting the stateless session EJB goes down, the state is lost. Further, the session may time out due to inactivity and the state is lost. Another disadvantage related to the quality of service guaranteed for stateful session EJBs is that the container does not roll back the state if the overall transaction fails.

Pros	Cons
A middle of the road approach would both cache the state in the EJB, and store it persistently on the back end. Any update methods on the stateful session EJB would write to the persistent store and would update the cache to reflect the results. Read-only methods on the EJB would simply use the data in the cache. Whatever you decide to cache using stateful session EJBs, the “state” is managed automatically by the container rather than by explicit programming. All the programmer need do is specify the instance variables as non transient, and they are considered to be part of the state that gets managed. If memory gets overloaded with sessions, the container will passivate a bean, reactivating it later if necessary.	Scalability is affected, because a client must be attached to the server hosting the specific stateful session EJB that is referenced. This requirement for client/server affinity limits the ability to balance the workload among multiple servers. Another downside is that the mapping of data from the non-transient variables to the backing store (file or database) during passivation/activation is through the serialization mechanism and stored as a binary large object (BLOB) that is always stored/retrieved as a whole.

Alternatives
There are alternatives to using stateful session EJBs. For example, any of the approaches for converting a stateful to stateless access bean described in “Business logic access beans” on page 26 can be used. These same approaches could be used to convert a stateful session EJB into a stateless one, especially in situations where the data is stable and read only, or if client/server affinity is already being used.

In either of these cases, a singleton memory cache can be shared by all instances of a stateless session EJB within the same JVM to maintain data. It is also possible to cache this data in the client or Web application server (see , “Data flow sources” on page 45 for details).

Another alternative to stateful session EJBs when failover and ACID properties are required is to use an entity EJB (discussed in detail below). In this case, the “pseudo session” life cycle would be explicitly managed by the application, but its state data would be immune to timeout as well as server and transaction failures.

Session EJBs with session synchronization

Session beans can support the session synchronization interface, which lets them participate in the container's transaction processing. The session synchronization interface includes methods that signal when a transaction has been started, when it is being prepared for commit, and when it is finally completed, either with a commit or a rollback.

The effect is that the same session EJB can be called one or more times in the context of a single transaction, and the container (in conjunction with the transaction controller) manages the calls required to close out the transaction without an explicit call from the business logic methods.

Session synchronization requires that the session EJB be stateful, because it adds life cycle states associated with transactional semantics. However, you should think of it as "converting" either a stateless or a stateful session EJB to support synchronization. The reason this is important is that the advantages and disadvantages of the underlying session EJB type tend to dominate. Also, from a programming model perspective, this characterization associates the choice of session synchronization with deployment rather than with the business logic itself.

Role in the architecture

Session synchronization must be carefully designed, for example, if timeout of stateful sessions is actually desirable, and that client/server affinity and lack of failover support are not issues. However, many of the business logic methods could fail after partially updating the cached data, and the programming required to restore the data to its previous state can be more or less complex depending on the business logic.

The business logic methods can throw a system exception or set a flag to cause a rollback; they can throw application exceptions or exit normally to cause a commit.

Another situation where session synchronization may apply is in situations where data is backed up in a resource with a non-JTA-based transaction model. For example, Persistence Builder (PB) is a VisualAge for Java feature that provides advanced object model to relational mappings, such as preload caching of related objects, that are not yet available in our CMP entity EJB implementations. Unfortunately, PB has its own transactional model that must be followed.

Pros	Cons
The nice thing about session synchronization is that the business logic of the session no longer has to be concerned with managing transactions and cached state. Instead, business logic methods need only throw an exception when an error occurs to cause a rollback, or return successfully to cause a commit. In either case, the associated state is properly managed. If the code needs to cause a rollback without throwing an exception (say for read-only methods), it can explicitly invoke a setRollbackOnly on its EJB transaction retrieved from the context. In cases where the session EJB was originally stateless and only added session synchronization (and state) to hold a transaction, then failover and timeout is definitely not an issue, because the client (HttpSession or business logic access bean) will create one as needed anyway.	Except for the simple cases the session synchronization interface can be very difficult code to implement, especially if the underlying resource does not provide support. Also, the code to manage transactions must apply to all methods on the session that require a transaction. For example, there is no way to process the backup/restore differently based on the method(s) invoked without involving the methods themselves. In this case, it may be best to handle the compensation in the methods themselves. Implementing the session synchronization interface cannot be considered to support true two-phase commit. The reason is that the transaction coordinator is not obligated to resurrect the session and complete the transaction if there is a failure between phases. The net effect is that there is a window of opportunity where resources can become out of synch. Finally, session synchronization is relatively expensive to achieve at runtime, because it adds an additional set of methods that must be called to manage a transaction. There should never be more than one or two per unit of work (either of our designs above have only one).
Alternatives If a stateful session EJB is being converted to use session synchronization simply to provide transactional semantics of the cached data, then consider using a CMP entity EJB. The advantage would be transparent transactional semantics on the persistent properties. In other cases, the best alternative is to defer session synchronization implementation to the deployer role and have the business logic developer code the session methods to be as independent of transactional semantics as possible. This alternative takes session synchronization out of the “normal” programming model and makes it a deployment responsibility.	

Container-managed persistence entity EJBs

While a session EJB represents an object with a transient identity lasting only as long as the client and server both maintain a reference to it, an entity EJB represents an object with a persistent identity that lasts until the object is actually removed from the container. Because of this difference, entity EJBs have an associated key, and the home supports methods to find references in various ways:

- Find methods that return a single EJB reference based on the primary key or a set of properties that uniquely identify an entity
- Find methods that return multiple EJB references based on zero or more properties that identify a subset of all entities in the container

An entity has a set of properties, including those that make up the key, which are considered to be part of its persistent state. The associated business logic methods operate upon these properties without regard to how they are loaded and stored.

In a CMP entity EJB, the container manages the persistent properties. When bean-managed persistence (BMP) is specified, the developer explicitly codes well-defined methods invoked by the container to manage the persistent properties.

Role in the architecture

In most applications, the business objects associated with the various states in the business process model are the most natural fit for CMP entity EJBs, whether we wrap these business objects with access beans or not.

As with all EJBs, care must be taken to minimize the interactions between the client and server, even if the two will be co-deployed (as when the client is a session EJB). For entity EJBs, we recommend the use of the following approaches:

- Custom creates. These are designed to create the object and initialize its properties in a single call, rather than the default create that takes just the key properties followed by individual sets (or a call to a copy helper method as described below).
- Custom finders. These are designed to return a subset of the entity EJBs associated with the underlying data store, usually by passing in various properties that are used to form a query.
- Copy helpers. These are get and set methods that use data structure JavaBeans to return or pass a number of properties at once.
- Custom updates. These are designed to do some update function and return a result in a single call.

As a general rule, you can design entity EJBs such that you do at most a single call to them after a find for a given unit of work. Following this rule will insure that the application can be distributed as painlessly as possible (although it is usually best to co-deploy client and server, unless the logic executed on the server side is expensive enough to warrant load balancing).

Where entity EJBs are used, you will usually end up with the following:

`<Entity>Key`: A data structure JavaBean that holds the key properties

`<Entity>Data`: A data structure JavaBean that holds both key and data properties of the entity. Some go as far as to create a `<Entity>DataOnly` that holds only the non-key properties to minimize the marshalling overhead for the gets and sets.

`<Entity>Home`: The home interface for finding/creating the EJB, usually with the following methods:

> `<Entity> create(<Entity>Data):`
> creates a new entity and initializes all the properties
>
> `<Entity> findByPrimaryKey(<Entity>Key):`
> finds based on the key
>
> `Enumeration find<Entity>sFor<RelEntity>(<RelEntity>Key key):`
> returns entity EJBs associated with the related entity

`<Entity>`: The EJB remote interface with at least the following methods:

> `<Entity>Data get<Entity>Data():`
> returns the data structure JavaBean representing the data
>
> `void set<Entity>Data(<Entity>Data data):`
> sets the non-key properties from the data

`<Entity>Impl`: Implements the business logic methods specified in the `<Entity>` interface above.

Of course, there are numerous approaches that can be used. For example, many like to include methods that have individual properties passed in rather than forcing the use of a data structure JavaBean.

Also, many will add methods on the entity EJBs to aid in navigation across associations between objects. Of course, the implementations of these navigation methods ultimately use the custom finders described above.

Pros	Cons
The primary benefit of CMP entity EJBs is that persistence and transactions are completely transparent to the business logic methods. When we used session EJBs, the only way to get similar functionality was to implement the session synchronization interface and use the methods to load or store the state from a backing store. This advantage is key from an evolutionary perspective. Let's say our early iterations used the Persistence Builder behind the business object access beans and thus required session synchronization in the stateless session EJB associated with the Entry business logic access bean. Later, we migrate the business object access beans to use entity EJBs. Once all the access beans are converted, we could reimplement the stateless session bean to drop session synchronization without having to touch the business logic. The transaction started by the stateless session bean propagates through to each entity so that any changes are all or nothing.	As with all EJBs, the downside to CMP entities shows how having a rich set of object services can be a double-edged sword: the overhead associated with managing distribution, security, and transactions can be very expensive. CMP entity EJBs require the developer to trust the container implementation to provide persistence in an efficient manner. Currently, there are numerous deployment choices available within WebSphere Application Server for entity EJBs. While this is not a problem for the programming model, and should be considered to be an advantage, it does complicate the decision whether or not to use entity EJBs in the first place. At the same time that there are a large number of choices, there are never enough. Some would like CMP containers for CICS VSAM files, or IMS DL/I. Others are fine with relational databases, but would like even more bells and whistles, such as preloading of related objects.

Alternatives
There are at least three alternatives to CMP entities when our current container implementations do not seem to meet your requirements:

- Client access beans. This option may make sense if you cannot afford the remote method call overhead associated with EJBs.
- Session EJBs. This option may make sense if you need a thin client tier or must isolate the business logic from the client for integrity or load-balancing purposes.
- BMP entity EJBs. This option may make sense if having a simplified programming model for the business logic is the biggest requirement, but you have database requirements not met by our current container implementations.

The first two options have already been discussed in detail in this section. All three options can be used together effectively: business logic access beans passing through to session EJBs, which use business object access beans passing through to BMP entity EJBs. We will discuss BMP entities next.

Bean-managed persistence entity EJBs

A BMP is simply an entity EJB where the developer manually implements the service methods, most notably `ejbLoad` to load the persistent state from the backing store and `ejbStore` to store it.

Role in the architecture

We recommend that all entity EJBs be implemented as if they were CMP for the business logic programming model. That is, business logic methods should assume that all instance variables are loaded prior to the method executing, and that they will be stored if needed when the method completes. The BMP methods to load and store the persistent instance variables should be implemented as part of the deployment process when the characteristics of the data store are known. This approach is very much the same as what we suggested for session synchronization methods on session EJBs.

In short, the ability to develop BMP methods expands the applicability of entity EJBs to situations where tighter control of the underlying data store is required. This requirement can occur when WebSphere does not support a legacy database. It can also occur when performance considerations preclude using the “vanilla” code generated for CMP entities.

Pros	Cons
This approach not only makes the business object logic much simpler to write, but also much easier to migrate to CMPs later, if the required container options eventually become available. Following this approach means that the BMP method implementations can be discarded and the entity EJBs can simply be redeployed, without having to change either the business logic methods or the client code.	The downside is that the persistence logic can be relatively complicated to implement efficiently. For example, in custom finders, you almost always need to cache the results of the query so that the iterative calls to the ejbLoad for each instance merely retrieve the data from the cache. In short, it can be very difficult to minimize the number of transactions and back-end accesses.
Alternatives The alternatives have already been discussed in the previous section: mainly, directly accessing the back end in a business logic access bean or session EJB. As with CMP entity EJBs, it is almost always a better practice to use a session EJB of some type as a wrapper, hiding the entity from the client. The advantage is that the session EJB can coordinate the transaction across multiple EJBs.	

Control flow mechanisms

If you have designed anything other than a monolithic component architecture (where all the application functions are controlled by a single program component) then you will need to understand the mechanisms by which you will transfer control from one component (the source) to another (the target).

Like the components themselves, the mechanisms vary by the tier upon which the source component executes at runtime. We will likewise divide this section up accordingly and have a subsection devoted to control flow mechanisms that can be initiated from:

- Browser-based components, such as HTML
- Web application server-based components, such as servlets

We deliberately do not include the enterprise tier, not because there are no mechanisms by which control flow is affected, but because they are pure Java method calls.

We will discuss the control flow mechanisms for each of the above in turn.

Browser component initiated control flow

As we discovered in the previous section, all browser-hosted components eventually are converted into HTML (or DHTML and JavaScript). And while there are lots of specific ways to transfer control between Web pages, they boil down to two that we will consider in this section:

- Those that issue HTTP GET requests
- Those that issue HTTP POST requests

HTTP GETs

An HTTP GET request can be effected in a number of ways:

- An `HREF` tag associated with text or an image
- Image maps, that allow specific areas of an image to target a given URL when clicked
- JavaScript `onclick='location=<URL>'` associated with a visible and clickable DOM object
- A `FORM` with `ACTION=GET` and a submit action invoked either through an associated `INPUT TYPE=SUBMIT` button, or a JavaScript `submit` action associated with a browser event

Once the link is established by any of these mechanisms, a user can click the link to transfer control to the next state.

Role in the architecture

HTTP GETs are used when the source state can directly transfer control to another because there are no update side effects, and where a small amount of data needs to be passed to the target.

Pros	Cons
Because there is no side effect involved, using HTTP GETs is the most efficient way to transfer control from one state to the next, especially where the next state is pure HTML that may be already cached by the browser. Pages invoked with an HTTP GET can be easily bookmarked to return to the same page with the same data where dynamic content is involved.	When using HTTP GETs, the ability to transfer data to the target state is limited to the URL query string (more on this in the next section), which has definite size limitations (often dependent on the Web server handling the request). Also, the location includes the data passed, which can be really distracting.
Alternatives There is no good substitute for an HTTP GET to transfer control with no side effects, because there is no need to involve an “intermediate” Web application component such as a servlet or JSP. However, you should remember that updating most of the data flow sources can be considered to be a side effect, which may be best handled by some other HTTP request type (such as a POST).	

HTTP POST (and other method types except for GET)

Unlike HTTP GETs, HTTP POST (and other types) can only be invoked from within a `FORM` with `METHOD=POST`. However, once a FORM context has been established, there are two primary mechanisms by which control is actually transferred:

- Clicking an `INPUT TYPE=SUBMIT` button associated with the FORM
- The JavaScript `<FORM>.submit` function, usually associated with a button or other clickable type

Once the link is established, triggering the associated event (such as clicking the link) will cause the POST request to be issued to the Web server. Usually, POST requests must be handled by a Web application component, such as a servlet or JSP.

Role in the architecture

HTTP POSTs are best invoked when update side effects are associated with the transition to the next state in the application flow model.

Pros	Cons
One advantage of an HTTP POST is that there are no absolute limits to the amount of data that can be passed to the Web server as part of the request. Also, the data passed does not appear on the location line of the browser. Another advantage of an HTTP POST is that the browser will warn the user if the request needs to be reinvoked (such as through a resize, back, forward or other browser event that needs the page to be reloaded).	However, some browsers display a rather ugly message if an HTTP POST request needs to be reinvoked due to a browser event, telling the user to reload the page. Also, an update side effect is usually expensive, so HTTP POST requests should be minimized by handling as many confirmations and validations as possible on the client side. Another disadvantage of a POST request is that it cannot be bookmarked because the associated data is not available in the URL query string as mentioned above (more on this in "Data flow sources" on page 45).
Alternatives There is really no substitute for an HTTP POST to attempt a transition with an update side effect. However, some transitions that may seem to have a side effect can actually be handled with an HTTP GET. For example, if a source page has a form to gather query parameters, it is possible to use an HTTP GET to transfer control to the servlet associated with the next state, which takes the parameters and reads the data to display. The reason that a GET is reasonable is that the action is read only and the amount of query data is usually relatively small.	

Web application server component initiated control flow

Just as all browser-based components reduce to HTML or DHTML and JavaScript, all Web application server components eventually compile to a servlet and use the servlet APIs.

We will briefly explore three mechanisms by which servlets can invoke other Web application components:

- `RequestDispatcher forward`
- `RequestDispatcher include`
- `HttpServletResponse sendRedirect`

RequestDispatcher forward

The `RequestDispatcher` is an object that can be obtained from the servlet's context (through the `getServletContext` method). The `RequestDispatcher` allows a target Web application component (HttpServlet and JSP) to be invoked from a source component in two ways: forward and include. We will discuss forward in this section and include in the next section.

Role in the architecture

The `forward` method is best used when the servlet completely delegates the generation of the response to a JSP.

Pros	**Cons**
When the forward call is used, the target has complete freedom to generate the response. For example, it can write headers, or forward or include to other Web application components as it sees fit. This freedom for the target makes programming the source component much simpler: it does not need to generate any headers or set up prior to delegating to the forwarded component.	A source component that invokes a target cannot generate any response prior to the forward call. Nor can it generate any response after the call returns. This restriction means you cannot compose pages with forward. A source component that was itself invoked by an include call (see "RequestDispatcher include" on page 43) cannot use the forward call. This restriction means a source component (one that will transfer control to another) has to know how it is being used. The target component must be a Web application component, requiring that targets of forward calls must be converted to JSPs, even if they contain purely static HTML.
Alternatives The most viable alternative to forward is for a servlet to set up the headers and enclosing HTML tags, then use the include mechanism (discussed next). This approach provides the ability to compose the response from multiple JSP components with as few changes as possible. This alternative also simplifies the JSPs involved, because they do not need to generate headers and enclosing HTML tags.	

RequestDispatcher include

The `include` method on the `RequestDispatcher` neither opens nor closes the response, nor does it write any headers, which means that multiple components can be included in the context of a single request.

Role in the architecture

Rather than use `forward` in the servlet's `doGet` method to transfer control to the associated JSP, it may make sense to include the associated JSP instead.

Pros	Cons
One reason to consider this approach is that the included components are much simpler to code, because they do not need to generate the <HTML>, <HEADER>, and <BODY> tags. For JSPs, the calling servlet can handle the code often required to prevent caching, simplifying them even further. The included components can often be reused in multiple places. For example, if we were not able to use framesets in our application due to restrictions on the browser, we could convert an HTML output to a JSP and compose the pages in the servlets. The components can be included by a superclass HttpServlet to provide a common look and feel across all states in the application. In future versions of WebSphere, included components can be cached, making it much more efficient to compose pages from multiple states. The ability to more easily exploit this feature when it becomes available is another good reason to consider including components.	Included components cannot write to the header or close out the response. Therefore, these actions must be done by the source component. Included components cannot be static Web pages (or fragments), requiring that they be converted to JSPs.
Alternatives When pages need to be composed, there is no really good alternative to include except to use framesets or named windows (see "Framesets and named windows" on page 14).	

HttpServletResponse sendRedirect

The `sendRedirect` method is implemented on the `HttpServletResponse` object that is passed in on the service methods associated with an HttpServlet. It generates a special response that is essentially code telling the browser that the requested URL has temporarily moved to another location (the target URL). No other response is generated by the source component.

The browser intercepts the response and invokes an HTTP GET request to the URL returned as part of the response, causing a transition to the next state.

Role in the architecture

The `sendRedirect` method is best used in a servlet after actions that cause update side effects to cause transition to the next state.

Pros	Cons
One benefit of using sendRedirect is that it prevents inadvertent re-execution of the side effects based on such browser events as forward, back, resize, print, view source, or reload among others (this unfortunate effect is sometimes called the reload problem). The reason sendRedirect solves this problem is another advantage: the URL for the update never appears in the browser's location line or history. The effect is that only the URLs of the "states" in the application flow model appear in the location and history, which is exactly the behavior desired.	The one disadvantage of sendRedirect is that it causes an extra round-trip between the browser and Web application server. Luckily, this extra round-trip only occurs during major transitions in the application flow model, and is well worth it, because sendRedirect solves a major source of data integrity errors in Web applications.
Alternatives There are no good alternatives to using a sendRedirect after processing requests in servlets that require update side effects.	

Data flow sources

Whenever two components must interact, whether they are separately developed components, or whether a single component is iteratively executed over time, it is likely that there will be a need to flow data from one to the other.

Like the first two sections, this section is divided into subsections describing data sources associated with each of the three tiers:

- Browser
- Web application server
- Enterprise servers

And as with control flow mechanisms, we show how the choice of data source can have a huge impact on the overall performance and integrity of the application.

Browser-maintained data flow sources

There are a number of browser-maintained data sources that we will discuss in this section:

- URL query string
- POST data
- Cookies

All of these sources provide the best scalability characteristics (because the data is maintained on the client), but with a trade-off that they may not be completely reliable (because the user has control over the data source).

The discussion in this section will address the details of these and other trade-offs.

URL query string

Whenever an HTTP GET is invoked, data can be passed in the query string part of the target URL. This includes `FORM` data (hidden or otherwise) with `METHOD=GET`.

In any event, the query string syntax is:

```
?<name>=<value>{&<name>=<value>}
```

Neither the names nor values can have embedded spaces; instead spaces and other special characters must be encoded.

The values can be retrieved through various methods associated with the `HttpServletRequest` object, most notably `getParameter`, which returns the value for a given name.

Role in the architecture

The most obvious place to use the query string in the online-buying application flow model is where data is being passed from one state to the next along a transition without side effects.

Another use for a URL query string is URL encoding of the session ID for `HttpSession` on the Web server (see “HttpSession state” on page 50) instead of cookies (discussed later in this section).

Pros	**Cons**
The benefit of using the query string is that it is very simple to retrieve the associated data.	Encoding the URL query string in sendRedirect calls and generated HREFs can be quite complicated and only a small amount of data can be passed. The query string is visible on the location line, and can sometimes be very long and confusing. This visibility in the query string extends to hidden fields in forms (when METHOD=GET)
Alternatives There is no good substitute for the URL query string to send a few small key values to the target component. However, where the data is common across most states in the application flow, it may be better to use cookies or HTTP sessions (both discussed later) to make the data flow transparent to the programs.	

POST data

When an HTTP POST is invoked from an HTML `FORM` with `METHOD=POST`, the input fields in the form are passed as part of an encoded input stream to the servlet. The `HttpServletRequest` can be used to access the fields in two ways: directly from the stream, or through the `getParameter` methods as if they were part of the URL query string (even though they are not).

Role in the architecture

Post is normally used to pass larger amount of data to a servlet.

Pros	**Cons**
As with the URL query string, one benefit to using POST data is that it is easy to retrieve, either by name or iteratively. However, unlike the URL query string, the main benefit to using POST data is that there is no absolute limit to the amount of data that can be sent. Finally, the data passed does not clutter up the URL, so hidden fields remain hidden to the casual user, and the encoding of the data is transparent to the source component.	
Alternatives As with the URL query string, there is no good substitute to POST data to provide the input parameters to actions with update side effects. However, where hidden fields are used to provide common data across the entire browser session, it may be wise to consider using cookies or HTTP sessions.	

Cookies

Cookies are data maintained on the client by the browser on behalf of the server. Cookies can be made to persist within or across browser sessions. Cookies are passed to the Web server in the header of the request. Any updates are passed back on the header in the response.

Within the servlet API, there are methods that allow you to get and set cookies.

Role in the architecture

Cookies are the preferred way to pass a session ID, if any, to the Web application server. The same approach can be used for other data that is constant across the Web application. Cookies are an excellent way to store a small amount of user preference data.

Pros	Cons
Cookies are automatically passed in the header, and thus do not require explicitly coding hidden fields or URL query strings in the HTML and JSPs. This feature of cookies makes the application much simpler to develop, test, and maintain. The ability to maintain persistent cookies means that the client machines can be enlisted to help share the cost of running the application. In an e-business application with millions of users, not having to maintain often used preference data for each one can be a significant savings in both space needed to store it and time needed to retrieve it.	Passing cookies back and forth can be relatively expensive. Further, the amount of data that can be maintained per server may be limited by the browser. The effect is that cookies should be used sparingly. Another problem is that not all browsers or levels of browsers support cookies. Even if they are supported, users can turn cookies off as a security or privacy measure, which means that: ▶ Your Web application has to be coded for the case where cookies are not available, and use alternative techniques (discussed below), ▶ You must make an explicit decision to support only users with browsers having cookies enabled. Also, other HTTP-based clients, such as applets, may have trouble dealing with cookies, restricting the servlets that they may invoke.
Alternatives URL encoding techniques can be used to put the equivalent data in the URL query string rather than relying on cookies.	

Web application server maintained data flow sources

There are three main sources of data maintained by the WebSphere:

- `HttpServletRequest` attributes
- `HttpSession` state
- `ServletContext` attributes

All these sources share a characteristic not associated with the other ones: only a Web application component (servlet or JSP) can store or retrieve data using them.

We discuss the advantages and disadvantages of each in the context of the role that source should play in the architecture. We also discuss any alternatives.

HttpServletRequest attributes

`HttpServletRequest` attributes (or more simply, request attributes) are maintained by the Web application server for the duration of the request in what amounts to an internal hash table.

The `HttpServletRequest` interface has methods to set and get the attribute values by name. You can also retrieve a list (`Enumeration`) of all the attribute names currently maintained in the request.

A JSP can use the expression syntax or Java escape tags to get request attributes using the servlet API, or it can use a bean tag scoped to the request (the default) with introspection to automatically load attributes whose names match the bean properties.

Role in the architecture

The most prevalent purpose of request attributes is for maintaining the data bean passed to a JSP by the servlet `doGet` method handling the display of a state. The point here is that by having a systematic naming convention, the contract between the servlet and JSP developer roles is very clear.

Pros	**Cons**
Of all the data sources, whether maintained by the browser, Web application server, or enterprise servers, HttpRequestAttributes are the second most efficient (behind passing the data directly in parameters of a method or in a shared variable). Because its scope is limited to the request, there is no need to write logic to “clean up” the data.	Setting too many objects into request attributes can cause problems with: ► The contract between the source and target component developers. For example, what do you name the attributes? What is their type? ► Performance, because each set is a Hashtable put and each get is a Hashtable lookup. The HttpServletRequest object does not persist across calls, so it cannot be used to hold data between states in the application flow model. The net effect is that request attributes can be passed only to targets using forward and include. Request attributes cannot be passed to targets invoked through sendRedirect.
Alternatives When using forward or include to dispatch to an associated JSP, a controlling servlet can pass data through HttpSession and ServletContext. When invoking a JSP or servlet through the sendRedirect, data can be passed using cookies or the URL query string.	

HttpSession state

An HTTP session is a short term (transient) relationship established between a client browser and a Web application server through which data can be maintained. It "lives" as long as both the client and the server maintain the reference to the relationship.

`HttpSession` state (in this section simply session state) is maintained by the Web application server for the lifetime of the session in what is basically a hash table of hash tables, the "outer" one keyed by the session ID (the session hash table) and the "inner" one keyed by the state variable name (the state hash table).

When the session is created, the ID is passed back and forth to the browser through a cookie (the preferred approach) or URL encoding. Since servlet API 2.2, the scope for sessions is a Web application (not the whole server).

The session is effective as long as both:

- The browser stays up to maintain the session ID cookie (or the pages with the ID encoded in the URLs), and
- WebSphere maintains the state hash table for the session

The outer session hash table can be lost if the Web application server goes down (and the session is not backed up). The inner state hash table can be lost on a timeout or through explicit application events (the `remove` method, for example).

The `HttpServletRequest` interface has methods to get the session (optionally causing it to create a new one if none exists), which returns a reference to the `HttpSession` object. Once you have a reference to the session, you can get and set state values by name. You can also retrieve a list (`Enumeration`) of all the state names currently maintained in the session.

A JSP can use the expression syntax or Java escape tags to get session state using the servlet API, or it can use a `useBean` tag scoped to session with introspection to automatically load states whose names match the bean properties.

Role in the architecture

Many Web applications handle login explicitly as part of the application flow, rather than use the security mechanism provided by the Web Application Server.

In this case it is customary to store some sort of "login" token into the session state. The session state maintained could be as simple as a customer ID, or it could be a complex object that includes additional data common to all the states in the application flow, such as open order.

Pros	Cons
Session state is rather easy to use in the program (especially if a data structure JavaBean is stored instead of individual values). The Web application server manages it at runtime based on configuration parameters, making it easy to tune non-functional characteristics such as failover and performance. This ease of use makes it tempting to store some application flow data (the current open order for example) in the session state rather than in a database that has to be explicitly administered. When the data is already being stored in the back end, and when accesses are expensive, the performance gains of using session state to cache the data can be significant.	Session state suffers from the same problems that request attributes do if you store too many objects in them in the course of a single request: there is a name and type contract problem with the target component, and a performance penalty with every additional Hashtable put and lookup. Session state has some additional disadvantages: ► Timeout. A session can time out when you least expect, making it risky to store significant application flow data. Usually you end up explicitly modeling and programming “save” and “load” type flows to make the problem less acute. ► Server failure. Even if you have an infinitely long timeout (and expect servlets to programmatically invalidate the session state), the server can fail, causing the data to be lost. Specifying that a session state be backed up in a database gets around this, and provides for failover. ► Cache consistency. When a session state is used to cache back-end data, how do you make sure the session state is in synch with the data stored in the back-end system. To provide for cache consistency means adding code to the doGet methods to check the key of the data in a session state with that in the request, and adding code to the doPost methods to remove the affected session states. ► Cluster consistency. It is likely that you will want to scale the Web site by adding a cluster of WebSphere application servers. Even if you add all of the extra logic to manage cache consistency from the previous item, you must either force client/server affinity (see “Stateful session EJBs” on page 32) and lose failover support, or back the session up in a shared database and impact performance. Of course, the memory resources required for session state should be taken into consideration. Indiscriminate use of HttpSession can use up vast amounts of data. For example, if there were 1000 active user sessions each needing to maintain a megabyte of data, your application would use up a gigabyte of memory for the session state alone.

Alternatives

When a session state is used to cache data stored in back-end servers, a viable alternative is to delegate caching to access beans or even EJBs, keeping the application flow logic in the servlet clean and simple. Another advantage to this approach is that the access beans are best able to keep the cache consistent because of their knowledge of the business logic.

If you use WebSphere security so that the getRemoteUser method on the HttpServletRequest returns an authenticated user ID, you can avoid the use of HttpSession altogether by keying explicitly modeled business objects with this user ID. The development costs of explicitly modeling session as a business object may be worth it in the ability to use that data by other types of applications (client server or distributed object as the case may be). Of course the primary benefit of eliminating the use of session state is that the application will scale much better, because client-server affinity is not required between the browser and Web application server.

If security is not turned on (maybe the application does not require it), and there is only a small amount of data to be stored, you can use cookies as described above, with the advantage that the client maintains the data.

However, there is no good substitute for HttpSession in scenarios with relatively small amounts of data that are relatively stable and must be maintained on the Web application server for security purposes, such as a login token.

Servlet context cache

The Web application server provides a context within which properties can be shared by all servlets and JSPs within that scope. This context is commonly called the "servlet context" and is accessible through the `getServletContext` method on the Servlet API.

Servlet context is used to obtain a `RequestDispatcher` through which `forward` and `include` can be invoked to flow control from one component to another (see "Web application server component initiated control flow" on page 42).

Like request attributes and session state, the servlet context also maintains an object that is the equivalent of a hash table, providing methods to get and set attributes by name as well as list the names stored within.

Unlike request attributes, which are scoped to a request, and session state, which is scoped to a session, servlet context is scoped by a Web application. The current specification explicitly states that sharing of servlet context in a cluster is unsupported. Note that session data is scoped by the Web application as well since servlet API 2.2.

Role in the architecture

Servlet context can be useful for shared read-only data, or when only one servlet updates the data periodically.

Another interesting use of servlet context is to cache references to business logic access beans (even if they are singleton wrappers).

If used for either purpose, we would likely set attributes into the servlet context as part of the `init` method, which would allow all servlets to use the data.

Pros	**Cons**
Proper use of servlet context can greatly reduce both the amount of session state data and the number of back-end accesses required to load it. As with session state, servlet context is very easy to deal with, and can eliminate the need to explicitly model extra business objects. Because servlet context attributes cannot be shared in a cluster, there is no requirement that data stored therein be serializable. This allows servlet context to be used to store very complex objects, such as access beans (preferred) or EJB references. Also, storing singleton references in a servlet context can prevent them from being garbage collected, because the reference is maintained for the life of the Web application server.	Also as with session state (and request attributes), you should minimize the number of attributes stored, and make sure that there is a systematic name and type convention in place. Unlike HttpSession, the specification prohibits sharing of servlet context in a cluster, primarily to force its use as a true cache. This limitation is not really a disadvantage when servlet context is used as a cache for stable read-only data, because each application server will perform better having its own copy of the data in memory. If for some reason there is a requirement to store common data, yet allow updates to it, then client/server affinity must be used to prevent cluster consistency issues. Of course, this means that the updates have to be associated with a specific user. Also, because the servlet context is shared by the entire Web application, you have to be careful to manage the code carefully, since multiple servlet threads could be accessing the same attributes simultaneously.
Alternatives Where servlet context is being used to store data from the back end to avoid extraneous accesses (a caching pattern), an alternative is to delegate caching the data to the business logic access bean. Where the default servlet context is accessed (the parameterless version of the API), then a viable alternative is to use the singleton pattern. These alternatives do not supersede the advantages of storing business logic access beans or connection objects in a servlet context to hold a reference and prevent garbage collection.	

Enterprise server-maintained data sources

Of course, there are many enterprise server-maintained data flow sources provided by and fully supported by IBM, such as CICS, IMS, and MQ. But in a discussion of the WebSphere programming model, we are only concerned with those that use standard Java APIs to provide access to the data or function maintained:

- Java Naming and Directory Interface (JNDI)
- JDBC

What separates these data sources from the others is that they can be used outside the context of a Web application server.

Java Naming and Directory Interface (JNDI)

JNDI provides a name value pair oriented interface very much like the Web application server-maintained data flow sources (request attributes, session state and servlet context cache).

The primary difference is that the JNDI name context is managed by a distributed name server, which allows the names and values to be shared across requests, sessions, application servers, and a cluster.

There are three types of objects that can be maintained in JNDI:

- Simple serializable JavaBeans (or Java objects)
- Distributed object references, such as EJB homes and remote interfaces
- Common object services, such as transaction contexts

The JNDI implementation provided in WebSphere Application Server caches home references after lookup, providing for additional scalability in a multi-user distributed environment.

Role in the architecture

One common use of JNDI in an application is to maintain user preference data, including credentials that aid in authentication.

In a Web application, JNDI would be used by business logic and business object access beans to get access to the home of EJBs.

Pros	Cons
The benefit to using JNDI is that is designed for storing small to medium amounts of relatively stable data per name, without requiring the involvement of a database administrator to create and maintain a new table. The fact that JNDI is distributable, sharable, and persistable makes it applicable in Web application scenarios where the other data flow sources cannot be used.	JNDI accesses are relatively expensive even with the automated caching support provided by WebSphere Application Server. Therefore, calls to them should be limited using the techniques discussed in “Stateless session EJBs” on page 29. This approach will make it easier to port to competitive products without having to worry about their implementation. Updates are even more expensive, so only relatively stable data should be stored in JNDI name contexts. The pattern is write once, read many. For example, user preference data fits into this category, but customer data, with its reference to the currently open order, does not.
Alternatives You can always explicitly model the data stored in JNDI as a business object and use either JDBC or EJBs (preferably behind an access bean).	

JDBC

JDBC provides a Java interface to relational databases, allowing dynamic SQL statements to be created, prepared, and executed against pooled database connections.

Any database that supports relational semantics can be wrapped with the JDBC interfaces and provide a “driver” for use in the client application or creating a data source.

Role in the architecture

In our online buying application, we would use JDBC to implement the business object access beans in cases where performance is crucial.

An example of when we might use JDBC is in loading a product catalog into the cache (distributed object overhead may be considered to be excessive for the benefits achieved).

Pros	Cons
JDBC provides all the benefits of relational databases to Java applications in an implementation-indepe ndent manner. Directly using JDBC in a client application will likely provide the most efficient implementation of the application, especially if connection pooling of data sources is used.	JDBC client code can be rather complicated to develop properly. Minimizing the number of statements executed in the course of a unit of work is key. Also, explicitly managing the transaction context can be complicated. If auto commit is turned off, care must be taken in the program code to commit or rollback the transaction as appropriate. If auto commit is left on, care must be taken when there are multiple statements in a single unit of work: each statement is a separate transaction, which can cause significant extra overhead and complicate error handling logic. Directly using JDBC locks your application into relational technology, although wrapping it within a business object access bean can help insulate the client application code, and make it easier to migrate later. Even if wrappers are used, JDBC requires that a JDBC driver be installed on the application server, potentially making it a “thicker” client that it would be if EJBs were used.
Alternatives The best standards-based alternative to JDBC is to use EJBs, which makes persistency transparent to the business object programming model, and allows the client to be “thinner”. Of course, you can use non-standard connector-based technology such as CICS, MQ, and IMS. But whether behind wrappers or not, these connectors make the client even thicker by requiring additional software to be installed.	

Chapter summary

We showed how dividing the programming model into its three fundamental features makes it easier to understand the issues that you will face when developing a WebSphere-based application. We will summarize these aspects in this section.

Throughout this chapter, we applied the programming model aspects to an online buying application to provide a concrete example. We will briefly summarize the mapping in this section as well, and show how the WebSphere programming model meets the challenges outlined in the chapter introduction.

Summary of programming model aspects

Table 1-4 shows the various features of the WebSphere programming model at a glance.

Table 1-5, Table 1-6, and Table 1-7 summarize the details of the components, control flow mechanisms and data flow sources.

Table 1-4 WebSphere programming model features

	Browser	**Application Server**	**Enterprise Server**
Component	HTML, DHTML and JavaScripts, XML, framesets	Servlets, JavaServer Pages, JavaBeans	Session and Entity Enterprise JavaBeans
Control flow mechanism	HTTP (GET & POST)	Java (forward, include, sendRedirect)	Java (RMI/IIOP)
Data flow source	URL query string, POST data, cookies	Request attributes, session state, servlet context	JNDI, JDBC

Table 1-5 Programming model components

Component	Tiers	Role in architecture
HTML	Browser	Specifies page content associated with a given state in the application flow model
DHTML and JavaScript	Browser	Handles client-side validations, confirmations, cascading menus, list processing and so on to minimize requests to Web server
Framesets and named windows	Browser	Groups related states on a single page to allow for smaller, more parallel requests and minimize need for explicit navigations
XML, DTD, and XSL	Browser	Allows request results to consist of data only and provide client control of display format
Servlet	Web application	Controls application flow for a given state; inherits common look and feel from superclass HttpServlet
JavaServer Pages	Web application	Handles generation of HTML/DHTML/XML for a given state in an application flow model
Data structure JavaBean	Java application	Serializable data passed between the other components such as servlets and JSPs/access beans, EJBs and copy helpers, etc.
Business logic access bean	Java application	Wrapper encapsulating units of work (can be equated with transitions in the business process model); can be stateless or stateful
Business object access bean	Java application	Wrapper encapsulating persistent business objects (can be identified by object model associated with states in the business process model)
Stateless session EJB	Enterprise Java server	Distributable implementation of stateless units of work (analogous to business logic access bean)
Stateful session EJB	Enterprise Java server	Distributable implementation of stateful units of work that cache resources or data on behalf of a user for the duration of a session
Session synchronization	Enterprise Java server	Methods added at deployment time to allow session EJBs to support transparent transactional semantics in business methods
CMP entity EJB	Enterprise Java server	Distributable implementation of persistence layer and associated business logic (analogous to business object access bean)
BMP entity EJB	Enterprise Java server	Methods added at deployment time to allow entity EJBs to control quality of persistence service

Table 1-6 Control flow mechanisms

Mechanism	Source components	Target components	Role in architecture
HTTP GET	HTML or DHTML	Any URL	Directly invoke the target URL associated with the next state, invoking a servlet or JSP for dynamic content
HTTP POST	HTML FORM	Servlet	Invokes the target servlet indicated in the ACTION to handle update side effects
Dispatcher forward	Servlet doGet	JSP	Delegate the generation of the HTTP response to the target JSP
Dispatcher include	Servlet doGet	JSP	Compose the response from one or more target JSPs that generate response fragments
Response sendRedirect	Servlet doPost	Any URL	Transfer control to the target URL representing the next state based on the ACTION result

Table 1-7 Data flow sources

Data flow source	Managed by	Control flow mechanism	Role in architecture
URL query string	Browser	HTTP GET sendRedirect	Pass small amounts of “key” data used to drive queries in doGet of the servlet associated with the target state
POST data	Browser	HTTP POST	Pass input data used to drive updates in the doPost of the servlet associated with the current state
Cookie	Browser	Any	Maintain data common to the user or session used to drive queries or updates in any state
HttpRequest attribute	Application server (WebSphere)	Dispatcher forward and include	Pass data representing the dynamic content between the controlling servlet and the associated JSP used to generate the response

Data flow source	Managed by	Control flow mechanism	Role in architecture
HttpSession state	Application server (WebSphere)	Any	Maintain stable data common to the session used to drive queries or updates in any state where cookies are not feasible
Servlet-Context cache	Application server (WebSphere)	Any	Maintain a cache of stable read-only data accessible for all requests on a single server to drive queries or updates in any state
JNDI	Name server (WebSphere)	Any	Maintain small amounts stable data accessible to all servers
JDBC	Database server	Any	Maintain any amount of any type of data accessible for any request

Meeting the challenges

The WebSphere programming model is compelling because with it you can meet all the challenges associated with developing a quality application that we identified in the chapter introduction:

- Functional—the WebSphere programming model features support everything you need to develop Web-enabled and distributed object applications.
- Reliable—by following the approaches discussed in this chapter, you can change the deployment characteristics of WebSphere hosted applications to handle different operational environments without changing the programs.
- Usable—the programming model supports the development of components customized to handle specific client requests for application functions that are automatically launched by the WebSphere Application Server.
- Efficient—the programming model features have clearly defined trade-offs that govern when they best apply to maximize use of system resources.
- Maintainable—the programming model supports a separation of concerns that make it easy to independently develop, test, and modify components.
- Portable—the features of the programming model are based on Java standards that make it easy to deploy application components on different platforms without change.

Furthermore, the programming model helps you meet the challenges associated with defining an optimal development process:

- Repeatable—analysis, architecture and design, relatively standard steps found in many development processes can be followed to develop quality WebSphere-based applications.
- Measurable—following the analysis, architecture and design steps results in a well defined number of servlets, JSPs, JavaBeans and Enterprise JavaBeans.
- Toolable—the systematic mapping from business process models to JavaBean and Enterprise JavaBeans, and from application flow models to servlets, JavaServer Pages and JavaBeans has made it possible to use a number of wizards, IDE and WYSIWYG tools.
- Predictable—given specific skill levels and tool choices, a team should be able to make and correct productivity estimates that can be used to drive project plans.
- Scalable—the ability to exploit a separation of concerns with well-defined contract objects not only makes an application easy to maintain, but also enables small or large teams of Java programmers and HTML page designers to work together on projects of any size with minimal amounts of coordination required.
- Flexible—separation of concerns also enables a team to use an iterative and incremental development process driven from the top, bottom, or middle in order to focus attention on high-risk items as early as possible.

If you develop your applications according to these principles, you will have an application that is not only functional, efficient, maintainable and portable, but also is able to exploit the deployment options best suited to your operational environment. Many of these options are discussed in more detail in the remaining chapters of this book.

2

Tools overview

In this chapter we introduce the tools we refer to in the rest of the book. The tools fall into two categories:

- IBM tools
- Third party tools

For each tool we include a brief description, links to further information, and provide references to the chapters where we use and describe the tool.

IBM tools

The following sections briefly describe the new IBM products that support the development of applications in the WebSphere environment. For each product we focus on the new and important features provided to developers.

The tools we discuss are:

- WebSphere Application Server Version 4.0
- WebSphere Studio Version 4.0
- VisualAge for Java Version 4.0
- WebSphere Business Components Composer

WebSphere Application Server Version 4.0

Version 4.0 is the latest release of WebSphere Application Server (WAS). For complete information about this latest version of WAS go to:

`http://www.ibm.com/software/webservers/appserv/`

The key new features in Version 4.0 of WAS that are of particular interest to developers include:

- Support for all Java 2 Enterprise Edition (J2EE) 1.2 APIs, including:
  - Java Development Kit (JDK) 1.3
  - Java Servlet 2.2
  - Java Server Pages (JSP) 1.1
  - Enterprise Java Beans (EJB) 1.1
  - Java Message Service (JMS) 1.0.2
  - Java Database Connectivity (JDBC) 2.1
- Incorporates support for Web services:
  - Simple Object Access Protocol (SOAP)
  - Universal Description, Discovery and Integration (UDDI)
  - Extensible Markup Language (XML)
  - Web Services Definition Language (WSDL)
- New tools to support J2EE application development
- New lightweight single server and developer editions
- Improved performance, including:
  - Improved scalability on SMP machines
  - Configurable caching of dynamic web content
  - Improved plug-in performance using built-in web server
  - Improved HTTP session clustering

Java 2 Enterprise Edition support

Version 4 of WAS is the first release to fully support the J2EE APIs. Previous releases supported earlier APIs that evolved into the versions included in Version 1.2 of the J2EE specification. If you have an existing application developed for an earlier Version of WebSphere, the application architecture and much of the code will be fully supported in Version 4. In cases where there have been significant changes in the J2EE APIs, for example in the move from Version 1.0 to Version 1.1 of the EJB specification, WAS provides backwards-compatibility modes that will allow existing applications to run largely unaltered while they are migrated.

Web services support

Web services are applications that use open standards to advertise, describe and provide services to other applications over the Internet. Because Web services are based on open standards and use XML to encode exchanged information they enable heterogeneous systems to interoperate without the need to develop an adapters for each new system. Furthermore, Web services provide the ability to dynamically locate and bind applications that provide a required service at runtime.

WebSphere Application Server is the fundamental building block upon which IBM's Web services offerings are based. The latest versions of the application server and related tools include built-in support to enable the development of applications that provide and use Web services. We touch on these tools only briefly in this publication—Web services will be covered in detail in a forthcoming redbook.

New tools

WAS provides a number of new tools to assist in the development and management of applications.

Of particular interest to developers is the Application Assembly Tool (AAT), which provides a single GUI interface for creating and managing J2EE module, such as EJBs and Web applications. It is also used to assemble individual modules into J2EE Enterprise Applications, ready for deployment into the application server. Via the AAT GUI you can create and update the Java archive (JAR) files that contain the packaged J2EE modules, and create and edit the XML deployment descriptors that describe the contents of the modules. The assembly tool is described in more detail in the section "Application Assembly Tool (AAT)" on page 390.

New single server version

There are two versions of WebSphere Application Server Version 4.0 on the distributed platforms (Windows, AIX, Solaris, HP-UX, and Linux):

- Advanced Edition (AE)
- Advanced Single Server Edition (AEs)

AE is the full multi-server version, and supports all WebSphere features, including work load management, clustering and failover. AEs supports all of the J2EE APIs (including support for EJBs) but is intended for entry-level, single server environments that do not require the full set of features provided by AE.

The AEs version of WAS is also available with a restricted license that limits its use to development environments only. This version is included with a number of IBM tools and is also available for free download from the IBM WebSphere Web site.

Differences between the AE and AEs versions

The points below highlight the ways in which the lightweight AEs version of WAS differ from the full AE version:

- They do not require a database in which to store configuration information—the configuration is stored in XML files in the file system
- No services are added to the Windows services panel—the server is started and stopped by executing shell scripts or batch files
- Only one application server process can be configured
- There is no support for clustered configurations or workload management (WLM)
- Administration is via a browser or editing configuration files—there is no standalone administrator's console
- WebSphere enterprise extensions are not supported
- Security is supported, however the only security registry supported is the local operating system—there is no support for LDAP or custom security

Improved performance

Version 4 of WAS introduces a number of enhancements in the area of performance. Of particular relevance to developers are the improved scalability on multi-processor (SMP) machines, and the ability to cache dynamic Web content.

Improved scalability on SMP impacts developers by improving the efficiency of code that takes advantage of Java’s multi-threading capabilities. This is especially relevant in a J2EE environment, where servlet requests and EJB method invocations are serviced by thread pools managed by the application server.

The ability to cache dynamic content produced by JSPs and servlets may also influence your application implementation. For each JSP and servlet in your application WebSphere now offers the ability to specify whether the output from the component is to be cached, and if so, how long it may be cached.

Often pages displayed contain varying levels of dynamic content—for example a customer’s account balance is unique to the customer, and so should never be cached. News headlines displayed on the same page for every customer, however, may be dynamic in that they are obtained from a database, but change relatively infrequently. If you separate the headlines into a separate JSP or servlet and include it in the page using the JSP `include` directive, you can tell the application server to cache the generated content for a certain amount of time, significantly reducing the number of database queries required to obtain the same information for every user.

WebSphere Studio Version 4.0

Version 4.0 is the latest release of WebSphere Studio. For complete information about this latest version of Studio go to:

http://www.ibm.com/software/webservers/studio/

The new version of WebSphere Studio includes several new features:

- Support for WebSphere Application Server Version 4.0: this implies support for the J2EE specifications: servlet 2.2 and JSP 1.1
- Creation of Web archives (WAR files): in this new version of Studio, it is possible to create the Web modules and publish them to the server locally or via FTP
- Support for custom tag libraries (as part of the JSP 1.1 specification)
- New creation and consumption wizards for Web services
- A copy of WebSphere Application Server 4.0 AEs, licensed for development use only

WebSphere Studio Version 4.0 is discussed in more detail in Chapter 10, “Development using WebSphere Studio” on page 237.

VisualAge for Java Version 4.0

Version 4.0 is the latest release of VisualAge for Java. For complete information about this latest version go to:

http://www.ibm.com/software/ad/vajava/

The new version of VisualAge for Java includes support to assist developers in writing code for and deploying code to WebSphere Application Server Version 4.0.

The WebSphere Test Environment (WTE) included in the new version uses Version 3.5.3 of the WebSphere runtime, which includes support for Version 2.2 of the Servlet API and Version 1.1 of the JSP specification—these are the versions supported by Version 4.0 of WAS.

Version 4.0 of VisualAge for Java also includes a new menu option that enables EJBs developed using the built-in EJB development environment to be exported as EJB 1.1 JAR archives. These archives include the XML deployment descriptor and database schema and mapping information.

In order to take advantage of the enhanced CMP mapping capabilities provided by VisualAge for Java, you will also need to download an enhanced EJB deployment tool (EJBDeploy) from:

http://www.software.ibm.com/vadd/

This is a new command-line tool that replaces the EJBDeploy tool shipped with WAS AEs. The new tool will be shipped with WAS AE.

VisualAge for Java is discussed in more detail in Chapter 11, “Development using VisualAge for Java” on page 259.

WebSphere Business Components Composer

WebSphere Business Components Composer (WSBCC) is a framework used in developing Web applications. It is discussed in Chapter 7, “Designing with frameworks” on page 153 and Chapter 12, “Development with frameworks” on page 283.

Third party tools

The third party tools described in the following sections are use in this redbook. The tools are:

- Rational Rose
- Rational ClearCase
- Jakarta Ant
- Jakarta Log4J
- Jakarta Struts
- JUnit

Rational Rose

Rose is a model-driven development tool from Rational that allows developers to model their applications using the Unified Modeling Language (UML). Information about Rose can be obtained from the Rational Web site at:

`http://www.rational.com/products/rose/`

Rose is discussed in more detail in Chapter 5, “Requirements modeling” on page 89 and Chapter 6, “Modeling and code generation” on page 123.

In developing this book we used the version of Rose included with the Rational Suite Enterprise Version 2001A.04.00.

Rational ClearCase

ClearCase is a software configuration management (SCM) tool developed by Rational. Information about ClearCase can be obtained from the Rational Web site at:

`http://www.rational.com/products/clearcase/`

ClearCase is discussed in more detail in Chapter 14, “Software Configuration Management” on page 385.

Jakarta Ant

Ant is an open source build tool developed as part of the Apache Jakarta project. Information about Ant can be obtained from the Ant Web site at:

```
http://jakarta.apache.org/ant/
```

Ant is discussed in more detail in Chapter 9, “Development using the Java 2 Software Development Kit” on page 183.

Jakarta Log4J

Log4J is an open source Java logging framework developed as part of the Apache Jakarta project. Information about Log4J can be obtained from the Log4J Web site at:

```
http://jakarta.apache.org/log4j/
```

Log4J is discussed in more detail in Chapter 13, “Guidelines for coding WebSphere applications” on page 325.

Jakarta Struts

Struts is an open source framework for developing Web applications using the Java Servlet API. It is a subproject of the Apache Jakarta project. Information about Struts can be obtained from the Struts Web site at:

```
http://jakarta.apache.org/struts/
```

Struts is discussed in more detail in Chapter 7, “Designing with frameworks” on page 153 and Chapter 12, “Development with frameworks” on page 283.

JUnit

JUnit is an open source framework that can be used to develop and execute automate unit tests against Java code. Information about JUnit can be obtained from the JUnit Web site at:

```
http://www.junit.org/
```

JUnit is discussed in more detail in Chapter 18, “Automating unit testing using JUnit” on page 517.

3

About the PiggyBank application

This chapter introduces the PiggyBank application that we use to illustrate examples throughout this book.

Introducing the PiggyBank application

The PiggyBank application is a very simple banking application that we designed and built while we were developing this Redbook. We use this example application to illustrate various points throughout the book.

The source code for the multiple versions of the application and other supporting files such as Ant build scripts and Rose models are included in the Web material that supports this book and is described in Appendix A, “Additional material” on page 557.

This chapter provides background information about the PiggyBank application that is intended to help place the various discussions relating to the application in context.

What is a piggy bank?

Not every reader is familiar with the term *piggy bank* so for those readers that are not, we attempt to explain. A piggy bank is a child’s toy, a container for saving small amounts of money—typically coins that are inserted through a small slot in the top. Traditionally a piggy bank is shaped to look like a pig, although the term is often applied to any such container.

Our application has much in common with a piggy bank—it is little more than a toy, although we hope that you can learn from it.

Functional overview

The PiggyBank application manages accounts and customer records for our fictitious bank. The application has two separate user interfaces:

- A Swing-based GUI that runs in a standalone Java application
- An HTML-based Web interface that runs in a client browser

Both interfaces access the same back-end system, and operate upon the same data. We often refer to these interfaces as *channels* because they implement just two of potentially many different channels of communication with our application.

Standalone client

The standalone client application is intended to be rolled out to customer service staff working in PiggyBank premises such as branches and call centers—it implements the full range of application functionality, including:

- ► Create customer
- ► Open an account
- ► Display customer information
- ► Display account information
- ► Transfer money
- ► Cash a check

Our user interface, shown in Figure 3-1, is basic but functional.

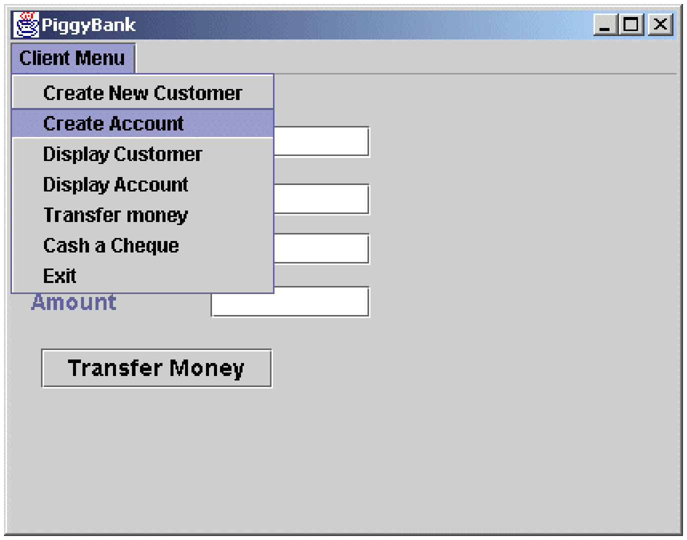


Figure 3-1 The PiggyBank standalone client in action

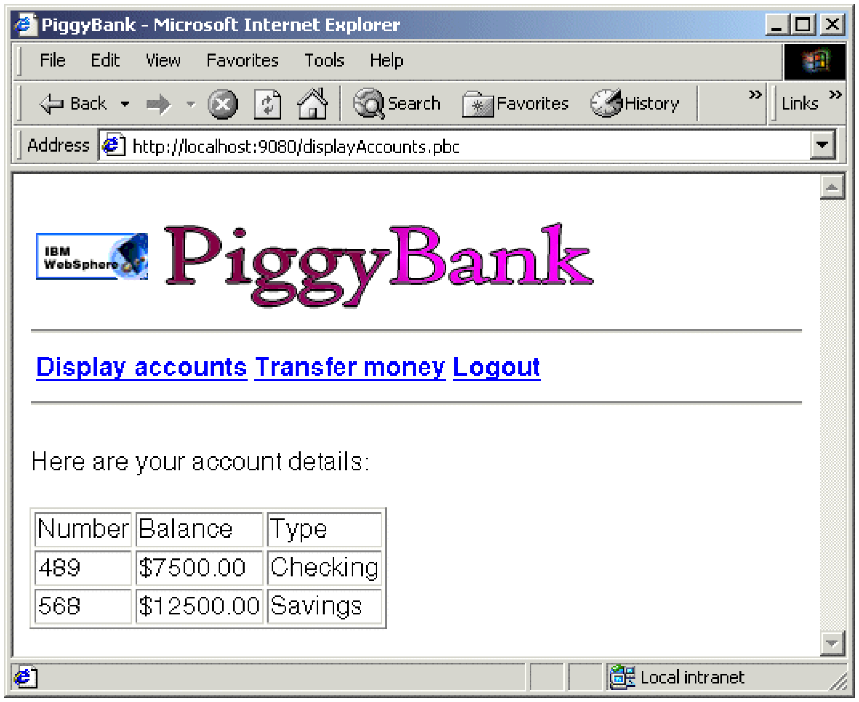
Web client

The PiggyBank Web client is intended to be rolled out to PiggyBank customers. It offers a much reduced set of functionality:

- ► Log on
- ► Display account details
- ► Transfer money
- ► Log out

In Chapter 12, “Development with frameworks” on page 283 we also show how to implement a multi-lingual Web interface using the open source Struts framework from the Apache Jakarta project.

Figure 3-2 shows the PiggyBank Web client in action.


PiggyBank - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites History Links
Address http://localhost:9080/displayAccounts.pbc
PiggyBank
Display accounts Transfer money Logout
Local intranet

Here are your account details:

Number	Balance	Type
489	$7500.00	Checking
568	$12500.00	Savings

Figure 3-2 The PiggyBank Web client in action

Security functionality

Typically security is a primary consideration in a banking application. The PiggyBank application is unusual in this respect—there is no security functionality implemented at all. All users of the standalone client have full access to all customer and account information. Web clients need only enter a valid customer ID to log in—there is a password field on the log on form but it is ignored.

We have omitted security from the application in order to allow us to focus on the real message of this book—how to design, build and deliver well-structured and serviceable J2EE applications for WebSphere Application Server.

Application architecture

The high-level architecture of the PiggyBank application is illustrated in Figure 3-3, showing the two types of application clients sharing the same back-end business logic and data.

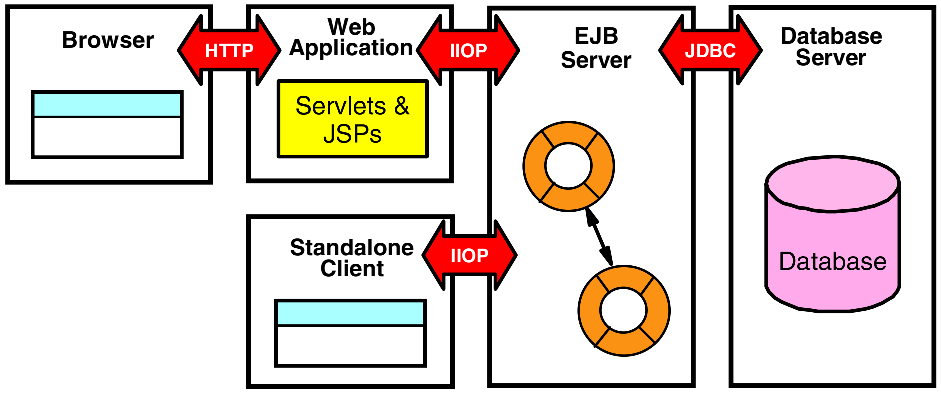


Figure 3-3 PiggyBank high-level application architecture

All of the application business logic is implemented as Enterprise JavaBeans (EJBs). The EJBs store persistent application data, such as account and customer information, in the database. Rather than make direct JDBC calls to persist data, the application uses container-managed persistent (CMP) entity EJBs, leaving the task to the WebSphere EJB container.

Both client channels communicate with the EJBs using RMI over IIOP—the standalone client communicates with the EJBs directly, whereas the Web client uses HTTP to connect to servlets that make RMI calls on behalf of the client, and display the results using Java ServerPages (JSPs).

The only logic implemented locally in the clients is basic validation and conversation management specific to the channel. The application implements the model-view-controller (MVC) architecture as described in “Model-view-controller pattern” on page 87—each client channel implements its own view and controller, but shares the same model.

Application modules

The PiggyBank application is split into five modules:

- Common code
- EJBs
- Use cases
- Standalone client
- Web application client

The layered, modular design of the PiggyBank application is intended to allow the replacement of any one module, while maintaining the interface to that module, without requiring any changes to the internal implementation of other modules.

For example, in Chapter 12, “Development with frameworks” on page 283 we replace the entire Web application module with a new one based on the Struts framework—although the internal implementation of the module changes significantly, no changes are required in any other module.

The relationships and dependencies between the modules are illustrated in Figure 3-4.

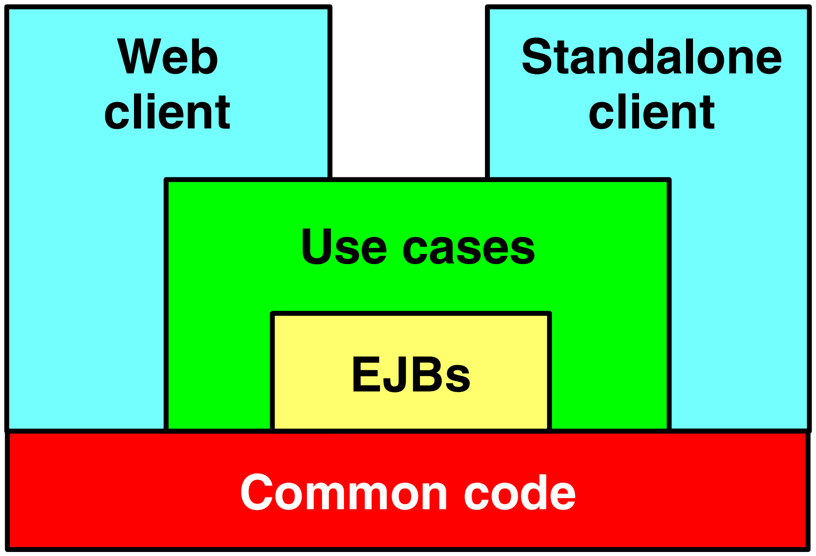


Figure 3-4   PiggyBank application modules

Common code

Common code underpins the entire application. The PiggyBank common code falls into two categories:

Helper classes	Helper classes provide common functionality to all other modules in the application, such as message logging or EJB lookup.
Dependent classes	Dependent classes bridge the boundaries between modules—they generally appear in the method signatures that define the interfaces between modules, and classes are thus dependent upon them. They consist of data only objects, as described in “Data structure JavaBeans (data beans)” on page 23, and application exceptions.

One fundamental characteristic that applies to all of our common code is that is has no dependency on any other application code—that is to say it can be compiled in isolation without reference to any other module. Dependencies on external modules, such as a third party logging library, are allowed.

EJBs

The EJB code implements all of the PiggyBank persistence and business logic. The EJB module provides services to clients via a session bean façade—the internal implementation of the business logic hidden behind the session bean interface is not intended for use by clients of the module (although this policy is not enforced). This arrangement is illustrated in Figure 3-5.

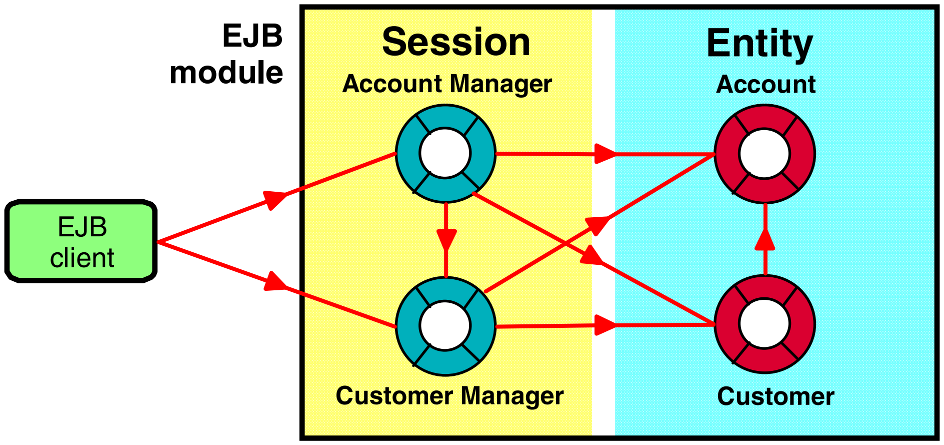


Figure 3-5 EJB module session façade

The EJBs depend only upon the common code at build and runtime—no other modules are required.

Use cases

The PiggyBank application use case classes fulfil the role of business logic access beans as described in “Business logic access beans” on page 26. Each use case class supports a single use case defined during the requirements analysis described in “Use case analysis” on page 90.

The use cases implement the command pattern—clients create an instance of the use case class, populate it with data using setter methods, and then invoke the `execute` method. All use cases return data-only beans as a result of their execution.

Internally the use case classes are EJB clients—they delegate the processing of business logic to the EJB layer. In the PiggyBank application they are the only EJB clients (apart from EJBs such as the session EJBs, of course). This extra layer gives us a number of advantages:

- Flexibility to modify the business logic implementation without affecting our client channels
- Removes the need for client channel programmers to understand how to access the EJBs, or indeed anything about the EJB implementation, including the fact that there are EJBs involved at all
- Encapsulates the EJB access code into a single place, where it can be managed and maintained more easily

The use case code depends upon the common code and the EJB code to compile.

Standalone client

The standalone client application is a simple Swing GUI application. It uses the use case classes to access the business logic. Results from the use cases are returned in the form of data only beans—the GUI code extracts information directly from the data only beans and presents it to the user.

Because the client code is—via the use case code—an EJB client, it must be packaged as a J2EE application client and execute inside the WebSphere client container.

Despite this, there are no code dependencies on the EJB layer—the standalone client requires only the common and use case code to compile.

Web client

The Web application client is implemented using servlets and JSPs. At the core of the Web application is a crude controller servlet that implements the command pattern. The controller examines the incoming request URI and uses it to determine which command must be executed. The commands are created as individual classes that implement an interface with a single `execute` method. This approach is convenient to work with and avoids many of the deficiencies in the single servlet approach described in “Role in the architecture” on page 19.

The Web application commands access the business logic using the use case layer. The data only beans they receive in reply are packaged in channel-specific view beans, which are then placed into the HTTP request and passed to a JSP for display. The view beans are a wrapped around the data beans, providing JSP-friendly services such as iteration and text formatting.

As with the Swing client, the code that implements the Web channel requires only the common and use case code at compilation time—there are no references to classes in other layers.

Application implementation

The PiggyBank application is implemented in a number of Java packages. Table 3-1 lists the packages included in each module, and describes the contents of each package.

Table 3-1 PiggyBank modules and packages

Module	**Package**	**Description**
Common	`itso.was4ad.data`	Data only beans
	`itso.was4ad.exception`	Business exceptions
	`itso.was4ad.helpers`	Helper classes
EJB	`itso.was4ad.ejb.account`	Account EJBs
	`itso.was4ad.ejb.customer`	Customer EJBs
Use case	`itso.was4ad.usecase`	Use cases
Standalone client	`itso.was4ad.client.swing`	Swing GUI client
Web application	`itso.was4ad.webapp.command`	Web commands
	`itso.was4ad.webapp.controller`	Controller servlet
	`itso.was4ad.webapp.view`	Web channel view beans

Application delivery

The deliverables from our development process are the J2EE modules that comprise the EJBs, Web application and standalone client, plus the common supporting code (the common code and use cases) required by these modules.

In a real-world development project, these modules would be assembled into one or more enterprise archive (EAR) files for deployment into the production environment. Typically there would be at least two EAR files in production. One would contain the server side code—the Web application and EJBs, and would be installed in the application server. The other EAR file would be distributed to the client machines, and would include only the standalone client code and code required for the client to run. Another permutation would have the EJB and Web application code split into separate EAR files and deployed into different parts of the infrastructure.

The deployment of an application into a production is beyond the scope of this book. Nevertheless we have to deploy our code into a WebSphere runtime environment in order to properly test it—in order to do this we describe how to create a single EAR file that contains all of the PiggyBank modules, and how to deploy the EAR file into WebSphere.

From a development perspective the configuration, assembly and deployment of the various modules that make up the application is irrelevant. As long as we produce truly portable, J2EE compliant modules with well-defined interfaces and dependencies, we can largely disregard such considerations. This gives the deployment team the flexibility to configure the production environment according to the expected and actual workload and usage patterns, and allows developers to concentrate on the application business logic, rather than topology questions such as the number of machines and the connectivity between them.

This is not to say developers can ignore deployment issues completely—it is essential, now more than ever, to write code that will function correctly no matter how it is deployed. Developers must assume, for example, that EJB server components are physically remote from a client, and that clustering features such as workload management and session persistence are in use. This means that there may be multiple copies of a component active at any one time, and that subsequent requests from a particular user may not necessarily return to the same Java process or even the same machine. These are the foundations upon which truly flexible, scalable applications are built.

Part 2

Analysis and design

In this part we discuss activities associated with the analysis and design of a WebSphere application. We describe how we may model our application requirements and use that model to drive our application design. We also discuss how tools such as Rational Rose can assist us with analysis and design activities.

The final chapter in this part of the book discusses how we can use frameworks in our application design. It introduces two frameworks—Jakarta Struts and WebSphere Business Components Composer—that we use in later chapters to implement elements of our example application.

4

Overview of development activities

In this chapter we introduce the analysis and design activities described in this part of the book. Next we discuss the assembly of a development team and the roles within it. We also introduce the design patterns referred to in this book.

Analysis and design activities

The analysis activity has a lot of added value, especially in a development “from scratch”. First, it captures the requirements in a set of use cases. Then, applying several methods such as brainstorming, mind-mapping, lexical analysis and heuristic modeling on the earlier deliverables identifies candidate objects, which will be precisely defined in a data dictionary. Refining the list allows static object modeling to provide a class diagram and optionally CRC cards. Finally, dynamic modeling delivers a set of sequence diagrams combined with some optional state diagrams. These steps can be iterated as long as required.

Figure 4-1 shows the trail that is followed in Part 2, “Analysis and design”.

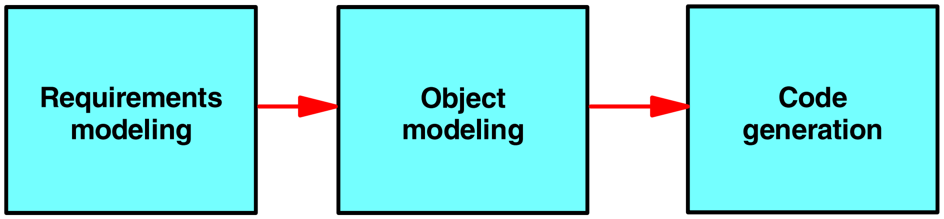


Figure 4-1 Analysis and design trail

The term “modeling” is hereby used in the common sense of “a simplification of the reality”. Modeling can be split into two main activities: analysis and design (Figure 4-2).

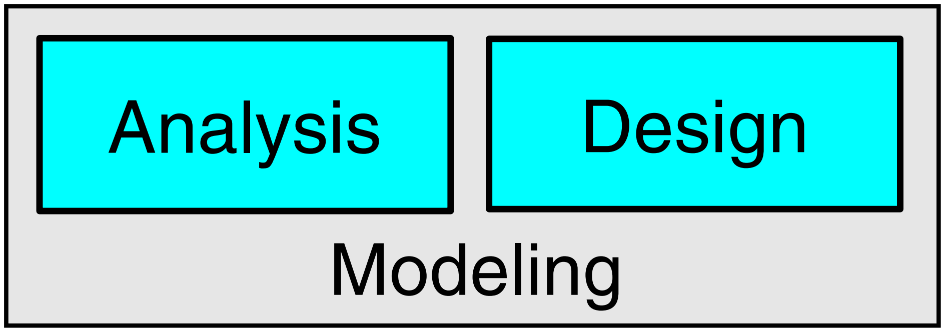


Figure 4-2 Modeling parts

The distinction between analysis and design must be permanently kept in mind:

- Analysis describes what the system does.
- Design describes how the working system will actually perform its task.

As a corollary of this:

- Analysis deliverables are kept at the business level.
- Design activity produces the technical deliverables.

This part of the book assumes that the project is already covered by a more complete OO methodology.

Assembling a development team

The e-business applications are created by multidisciplinary teams. The skills required are contributed from graphic artists, Web page designers, client and server side script writers, and Java programmers.

Whether there is only one person or one hundred, the concept of the separation of roles and responsibilities is key to the successful creation and maintenance of the e-business application.

The command pattern involves separating the tasks by each role, such as:

- The *HTML developer* uses a tool like WebSphere Page Designer to generate HTML pages and JSP templates.
- The *Script developer* uses a Java programming environment like VisualAge for Java to edit, test and debug servlets and JSPs.
- The *Java business logic developer* uses a Java programming environment, like VisualAge for Java, and builders, like the integrated EJB builder, to specify the details of the business logic, to access legacy applications and data, and to build the commands.

Further reading

A good reference for assembling a development team is the book *The Rational Unified Process, An Introduction*. Philippe Kruchten.

Development roles

In the book, for the sake of clarity, when we refer to, for instance, the senior business analyst, we mean “the person who plays the role of senior business analyst”. It might be the same person who plays the role of say the junior Java developer.

In considering our fictitious project analysis and design, we assume that the project team will consist of a number of persons who broadly fit into the following categories:

Business analyst	This person is supposed to have a lot of business knowledge while few technical skills. He can write documents in good language. In this particular context he knows about the use case driven approach and has basic notions of OO analysis and application development.
OO designer	This person is highly skilled in IT and OO and knows most of the technical environment. He works intensively with the application architect.
Java developer	Senior developers have significant experience from working on previous projects. They are entrusted with delivering the more technically challenging and performance critical sections of the code. Junior developers are competent Java programmers with less experience than their more senior colleagues. The junior developers concentrate their coding efforts on the servlet and client portions of the code.
Application architect	The application architect is the technical lead on the project. He has overall responsibility for determining the high-level structure of the application, and the interaction between components. He is also a senior developer.
Web developer	Web developers are skilled in the use of HTML and the tools used to develop and maintain Web content. They are responsible for delivering HTML and JSP pages and the various interface elements such as images contained within them.

Patterns

Most application designs follow certain documented patterns that have been proven in many successful installations. The original book describing patterns is *Design Patterns: Elements of Reusable Object-Oriented Software*. Erich Gamma, et al.

We will use some of the patterns for the PiggyBank application.

Model-view-controller pattern

The model-view-controller (MVC) pattern consists of three parts:

Model	A set of objects that represents the business logic of the application. This usually includes classes to represent business abstractions (such as accounts, purchases, and so forth) as well as real-world objects (such as employees and customers).
View	A particular way of presenting a set of information to a user. The best way to think of a view is to think of a particular Web page or screen that displays a single set of linked data to the user.
Controller	The layer in an application that handles the details of application flow and navigation. It translates information from the model layer into a form the View layer can understand, and deals with the all-important decisions as to what View to display next in response to a particular user action.

The key here is that the model is kept separate from the details of how the application is structured (the controller) and how the information is presented to the user (the view).

The question that may be arising in some readers' minds is "Isn't J2EE already a framework to solve these problems?". Well, in a way it is. Many development organizations have successfully applied the following mapping of J2EE API's to the three roles in the MVC pattern:

- Model: JavaBeans and Enterprise JavaBeans
- View: JavaServer Pages
- Controller: Servlets

Here servlets act as controllers and are the recipients of HTTP POST requests, and are responsible for passing POSTed data to the model and selecting which JSP page will be invoked to display results. This is often called the "Model II" JSP architecture.

This architecture has been described at length in:

- *Using JavaServer Pages: Servlets Made Simple*. Kyle Brown and Gary Craig. The Java Report, August 1999.
- *Enterprise Java Programming with IBM WebSphere*. Kyle Brown, et al.

We will come back to the MVC pattern in “MVC pattern” on page 100.

Command pattern

The command pattern is often used to further isolate business logic from controller activities. Commands are used to invoke business logic in the same or in other machines from the servlet controller. Such a design provides additional flexibility because the business logic may be moved to other servers without impacting the front-end and the servlets.

We will come back to the command pattern in “Command pattern” on page 105.

5

Requirements modeling

In this chapter we describe methods to start building Web applications with a use case driven approach, based on use case analysis principles.

Next we discuss several design techniques we can use to realize the use cases and discuss how to interact with external systems.

Finally, we introduce some common principles to design a Web user interface tied to the use case analysis.

Use case analysis

The use case analysis describes the functional requirements of the system under development. The model uses graphical symbols (Figure 5-1) and text to specify how users in specific roles will use the system through use cases. The textual descriptions describing the use cases are from a user's point of view; they do not describe how the system works internally or its internal structure or mechanisms.

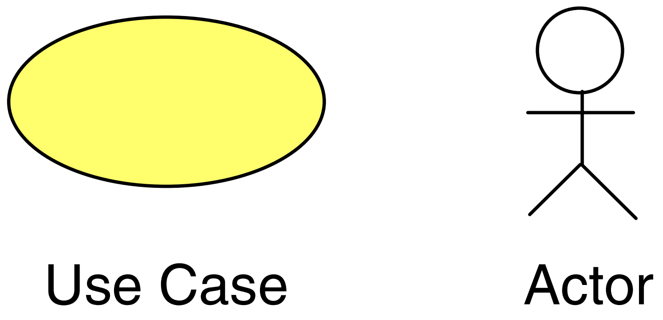


Figure 5-1 Use case analysis UML icons

The use case analysis basically includes the following elements (Figure 5-2):

- Actors
- Use cases
- Communication associations between actors and use cases
- Relationships between use cases (also known as use case associations)
- Termination outcomes
- Conditions affecting termination outcomes
- Termination outcomes decision table
- Problem domain concept definitions in a glossary (also known as data dictionary, but, be careful, this term is often debased)
- Flows of events and system sequence diagrams

Some of them can be optional or informal at this step, depending on the methodology.

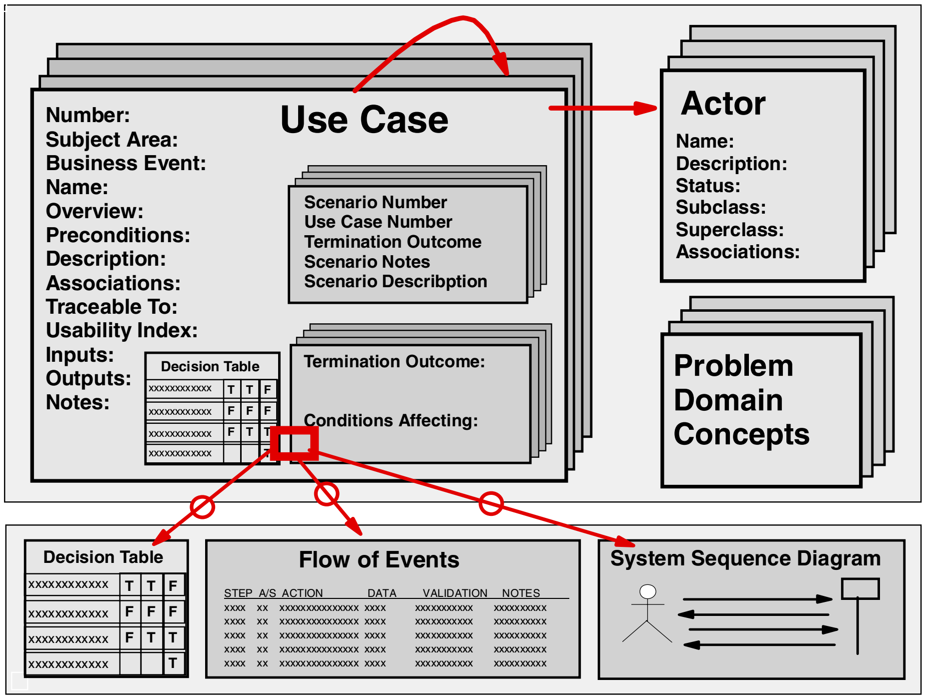


Figure 5-2 Use case analysis elements

Actor names, actor descriptions, use case numbers, use case names, use case business events, and use case overviews as well as communication-associations between the actors and the use cases provides an overview of the functional requirements. The other constructs of the model document the expected usage, user interactions, and behaviors of the system in different styles and depth.

The main purpose of use case analysis is to establish the boundary of the proposed software system and fully state its functional capabilities to be delivered to the users. Other purposes of use case analysis are:

- Provides a basis of communication between end users and system developers.
- Is the primary driver for estimating the application development effort.
- Provides a basis for planning the development of the releases.
- Allows scheduling of common functionality early in development.
- Allows development of smaller increments while maintaining broad coverage.

- Allows scheduling of complex functionality later in development without changing the code that already exists or having to test for damage to earlier releases.
- Provides a basis for identifying objects, object functionality, interaction, and interfaces (see Chapter 6, “Modeling and code generation” on page 123).
- Provides the primary basis for defining the user interface requirements (see “Designing the user interface” on page 115).
- Provides a basis for defining test cases.
- Serves as the basis for acceptance testing.
- Provides a basis for producing user support materials, such as user documentation and electronic performance support interventions.

Here are some definitions to provide a base for discussion:

- A *requirement* is a condition or capability to which the system must conform.
- A *use case* is a sequence of actions a system performs that yields an observable result of value to a particular actor.
- An *actor* is anything external to the system under development that interacts with it. An actor can be an instance of a user (specific human) or another system. An instance of an actor can create instances of uses cases. They are prospective users playing a specific role with the system. Several users can play the same role and one user can perform several roles.
- *Primary actors* are those roles for which the system is being built to serve. They are associated with sets of use cases that reflect the primary functions of the system.
- *Secondary actors* have associated use cases for supporting the system in providing its primary functions. These roles will typically be for maintaining and customizing the system, such as a system administrator. This is more considered at design time.

The are many reasons why actors are used with use cases:

- Defining actor types allows us to define use cases in terms of specific expectations (uses) of the system. In other words, it allows us to narrow the expectations to specific roles in which a human user would be using the system.
- Defining actors helps to identify the system border; what is inside the system and what is outside the system.
- Defining actor types helps us show user training needs for particular aspects of the system.

- Defining actor types helps us show which security needs are required for which user types.

Finally, it is important to remember that because the system under development has not been placed into the user environment, user types/job titles probably do not exist specifically for the new system. After the new system is placed in the user environment, it is not unlikely that job titles/user types will be redefined and possibly reflect the actor types defined in the use case model.

For additional information, the reader can consult the *UML User Guide*, Rational Unified Process.

PiggyBank use cases

These are the PiggyBank use cases:

- Transfer money
- Cash a cheque
- Display balance

Transfer money

In this use case we transfer funds from a PiggyBank account to another PiggyBank account.

- Input:
  - Customer ID
  - Credit account number
  - Debit account number
  - Amount to be transferred
- Basic path:
  1. The customer enters the required input information and submits the request.
  2. The system checks that both accounts exist, that the customer is owner of the debit account and that the amount to be transferred is lower than the debit account current balance.
  3. The system debits the customer account and credits the other account by the specified amount.
  4. The system displays the customer a summary of the transaction.
- Alternative path:

  2b. One of the checks fails.

  3b. The system displays a message explaining why the transaction cannot be completed.

Cash a cheque

In this use case we cash a cheque from the city bank and credit a PiggyBank account.

- Input:
  - Cheque reference
  - Cheque amount
  - Account number to credit
- Basic path:
  1. The customer enters the required input information and submits the request.
  2. The system checks that the credit account exists.
  3. The system calls the remote city bank system and provides it the cheque reference and amount.
  4. The city bank replies with a YES if the cheque can be debited from the referenced city bank account.
  5. The system credits the account by the specified amount.
  6. The system displays the customer a summary of the transaction.
- Alternative path:

  2b. The credit account does not exist.

  3b. The system displays a message explaining why the transaction cannot be completed.
- Alternative path:

  4c. The city bank replies with a NO: the cheque cannot be debited from the referenced city bank account.

  5c. The system displays a message explaining why the transaction cannot be completed.

Use case analysis typically includes additional use cases for main business objects life-cycle management, also known as CRUD use cases, where CRUD stands for create-read-update-delete.

Therefore, with regards to the PiggyBank class diagram, the PiggyBank use case analysis should also include a display balance use case.

Display balance

In this use case we display the balance of a PiggyBank account.

- Input:
  - Customer ID
  - Account number
- Basic path:
  1. The customer enters the required input information and submits the request.
  2. The system checks that the account exists and that the customer is the owner.
  3. The system displays the customer the account balance.
- Alternative path:

  2b. One of the checks fails.

  3b. The system displays a message explaining why the transaction cannot be completed.

The actors of CRUD use cases can be different depending on the instruction. For instance, delete use cases are usually reserved for privileged users.

This ends the use case description at analysis level. Additional steps and use cases can be added at design time:

- Login and logout use cases
- Every use case can be started by checking whether the user is logged in and has enough privilege to perform the action
- Physical communication with the city bank might fail and lead to an additional alternative path
- Database transactions, included commits and rollbacks, could be specified in more details

PiggyBank use case diagram in Rational Rose

In this section we are going to write a class diagram in Rational Rose to model the PiggyBank features:

1. In Rational Rose, create a new J2EE model: *File -> New*. This opens the *Create New Model* wizard, which is automatically opened at Rose startup. Select *J2EE* and click *OK*. This loads the required subunits to model a J2EE application.
2. Save the currently-empty model as `PiggyBank.mdl`.

3. Create a new class diagram: in the model browser, right-click on *PiggyBank -> Use Case View* and select *New -> Use Case Diagram*. Call it `PiggyBank` and open it by double-clicking on it.
4. Put three use case icons and two actor icons on the diagram then link them as shown in Figure 5-3.

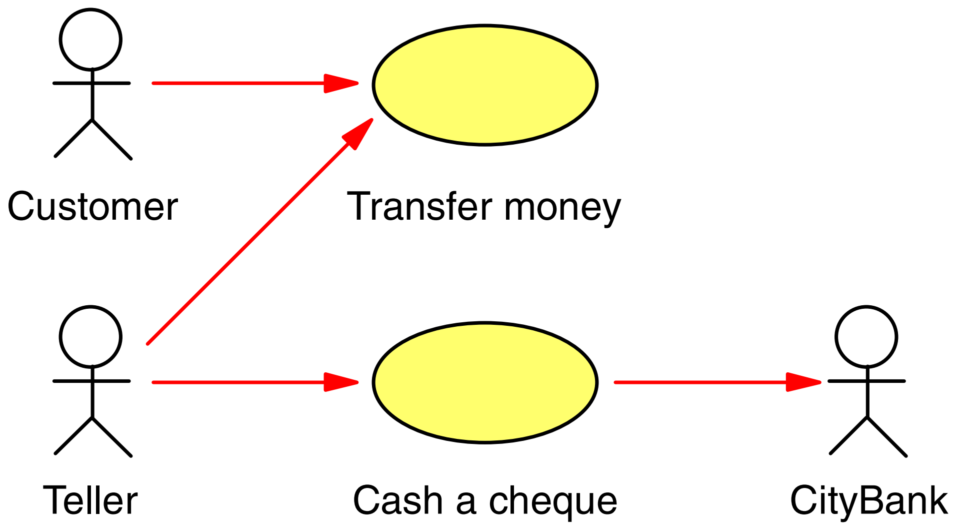


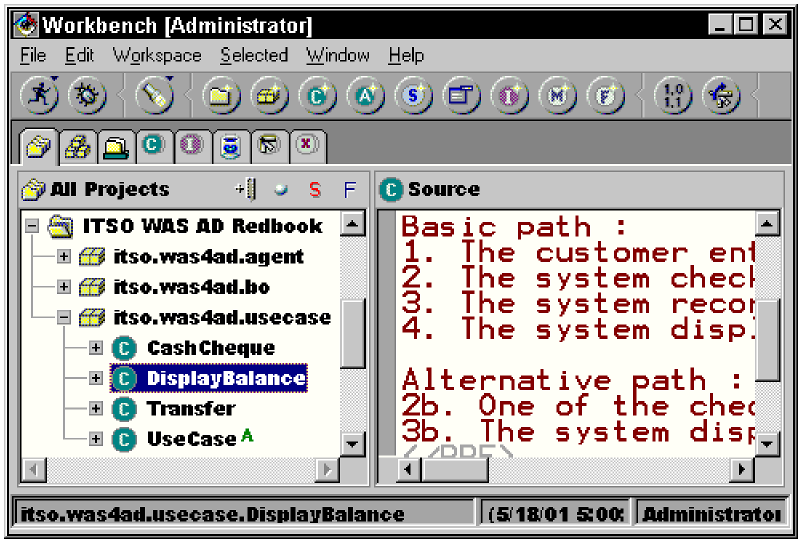
Figure 5-3 PiggyBank use case diagram

Use case descriptions in VisualAge for Java

Use classes descriptions can be put in the class-level Javadoc comment zone:

```
/**
* <PRE>
* Use case name and description
* </PRE>
**/
public class UseCaseName {
}
```

Figure 5-4 shows a use case description edition in VisualAge for Java.



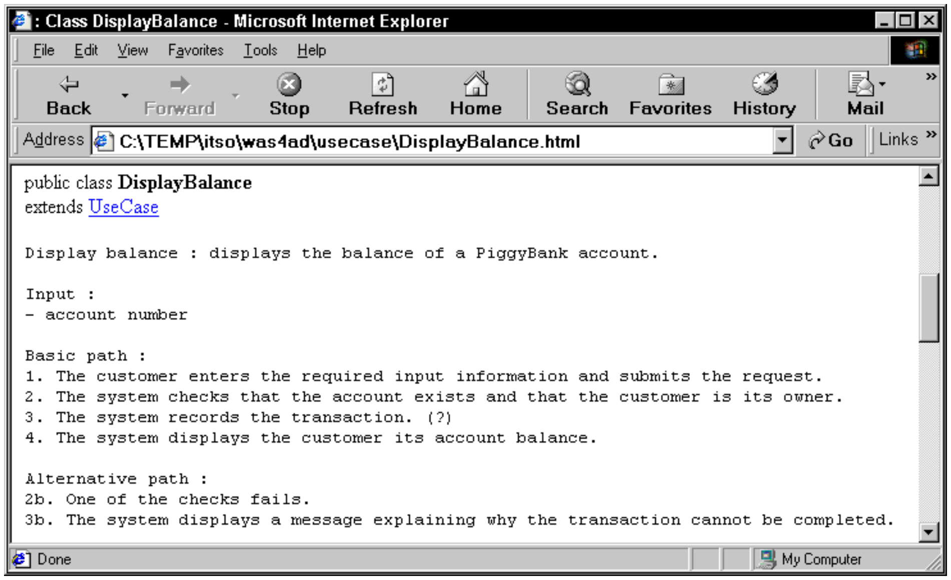
```
Basic path :
1. The customer ent
2. The system check
3. The system recor
4. The system disp.

Alternative path :
2b. One of the chec
3b. The system disp
```

Figure 5-4 Use case description in VisualAge for Java

The PiggyBank display balance use case would be commented like this:

```
/**
<PRE>
Display balance : displays the balance of a PiggyBank account.

Input :
- account number

Basic path :
1. The customer enters the required input information and submits the
request.
2. The system checks that the account exists and that the customer is its
owner.
3. The system records the transaction. (?)
4. The system displays the customer the account balance.

Alternative path :
2b. One of the checks fails.
3b. The system displays a message explaining why the transaction cannot be
completed.
</PRE>
*/
public class DisplayBalance extends UseCase
{
   .........
}
```

Figure 5-5 shows the Javadoc after HTML generation.


: Class DisplayBalance - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites History Mail
Address C:\TEMP\itso\was4ad\usecase\DisplayBalance.html
Go Links
Done
My Computer

public class **DisplayBalance**
extends UseCase

```
Display balance : displays the balance of a PiggyBank account.

Input :
- account number

Basic path :
1. The customer enters the required input information and submits the request.
2. The system checks that the account exists and that the customer is its owner.
3. The system records the transaction. (?)
4. The system displays the customer its account balance.

Alternative path :
2b. One of the checks fails.
3b. The system displays a message explaining why the transaction cannot be completed.
```

Figure 5-5 HTML documentation generated from Javadoc

Use case realization

After the use case analysis is completed, we have a list of use cases with their descriptions. The purpose of this section is to present different use case realization techniques, that is, different ways of designing the code that will actually implement the system use cases.

There is no ideal way—there are several neat techniques that do the things right. Each of the techniques brings different enhancements that can be combined. We start from a very basic approach, then we refine the concepts to keep it simple, while leaving no major drawback. Finally, we present the latest and finest improvements that help having a modular, versatile design.

The basic approach

The most naive way of implementing a Web application is to leave the development in anarchy and stick to the servlet framework itself. This typically results in several servlets having `doGet/doPost` methods implemented with SQL calls through JDBC, and HTML output performed through an incredibly unreadable bunch of `out.println` instructions.

The JSP technology brings an interesting concept of integrating the HTML code with the Java code. The integration is performed through the inclusion of so-called JSP tags with Java code into HTML pages, which become JSPs. At runtime, the JSP are compiled into servlets that do exactly the same thing as above. The main advantage is that a JSP is more readable.

Had the application to be JSP-centric or servlet-centric, this approach, also known as model 1, awfully mixes presentation and business logic. It is good for sample examples demonstrating the technology and quick-and-dirty application development, which is not what WebSphere is intended for.

A very popular technique known as model 2 suggests to mix traditional business logic development, servlets and JSP together. See “MVC pattern” on page 100 for further explanations.

Servlet mapping

This method maps each use case to a servlet class.

Tip: The servlet logic is coded in a `doGet` or `doPost` method, dependent on the HTML code that invokes the servlet and specifies a GET or POST method. As a good practice you can always code both `doGet` and `doPost` methods and call a `performTask` method that contains the logic.

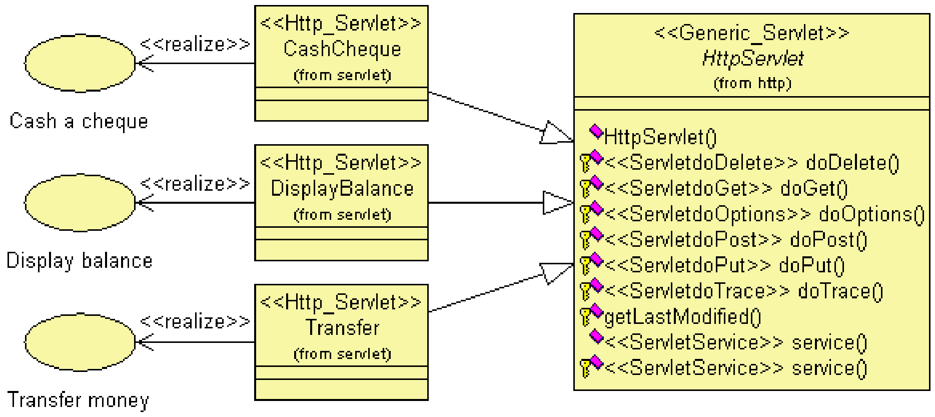
Remember the servlet model implies there is usually one class instance in the JVM memory, which is multithreaded, unless the servlet implements the `javax.servlet.SingleThreadModel` interface.

It must be also observed every servlet alias has to be declared in the servlet engine configuration (see Chapter 15, “Assembling the application” on page 389 and Chapter 16, “Deploying to the test environment” on page 431). If the application has many use cases, its maintenance can become tedious unless the servlet name is kept to the fully-qualified class name and will therefore be accessed through an URL, such as:

```
http://hostname/webapp/itso.was4ad.servlet.ServletName
```

The section “Servlet multiplexing” on page 104 explains how to get rid of this limitation, multiplexing entry points into one servlet.

Figure 5-6 shows the PiggyBank servlet use case realization diagram.



Cash a cheque
Display balance
Transfer money
<<realize>>
<<realize>>
<<realize>>
<<Http_Servlet>> CashCheque (from servlet)
<<Http_Servlet>> DisplayBalance (from servlet)
<<Http_Servlet>> Transfer (from servlet)
<<Generic_Servlet>> HttpServlet (from http)
HttpServlet()
<<ServletdoDelete>> doDelete()
<<ServletdoGet>> doGet()
<<ServletdoOptions>> doOptions()
<<ServletdoPost>> doPost()
<<ServletdoPut>> doPut()
<<ServletdoTrace>> doTrace()
getLastModified()
<<ServletService>> service()
<<ServletService>> service()

Figure 5-6   PiggyBank servlet use case realization diagram

Drawbacks of this realization are:

- This method does not follow the MVC pattern.
- The use case associations (that is extends and uses/includes) are difficult to implement (see the redbook *Servlet and JSP Programming with IBM WebSphere Studio and VisualAge for Java*, SG24-5755, from page 73 for more information on servlet interaction techniques).
- The servlet thread and object model, left to the servlet engine vendor-specific implementation, does not allow normal use of servlet instance variables.

MVC pattern

As a good structure for e-business applications, we suggest to isolate the business logic of an interaction from the work flow and the view by using the model-view-controller paradigm. This leads us to the three components of program logic as shown in Figure 5-7:

- The *user interface logic* is the ***view*** and contains the logic which is necessary to construct the presentation.
- The *servlet* acts as the ***controller*** and contains the logic which is necessary to process user events and to select an appropriate response.
- The *business logic* is the ***model*** and accomplishes the goal of the interaction. This may be a query or an update to a database.

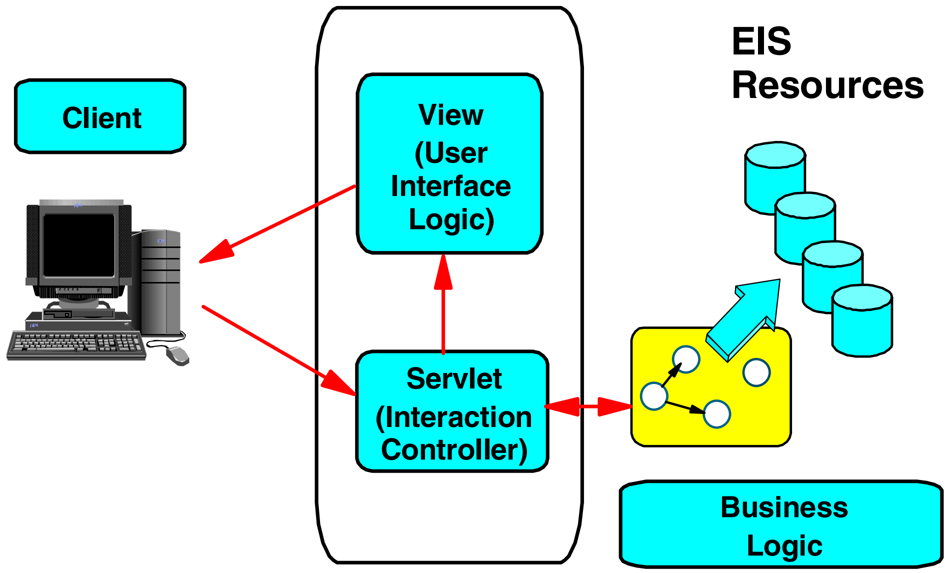


Figure 5-7 Web application model with MVC pattern

It is critical to maintain a clean separation between the different types of program logic, because the link between the servlet and the business logic is especially sensitive. As the reason for that we have to face several more problems in the communication between these layers, such as:

- **Problems with performance**: The granularity of artifacts on the server (that is, objects, tables, procedure calls, and so forth) often causes a single client-initiated business logic request to involve several round-trip messages between client and server, which consumes a significant amount of system resources. This may include several calls to perform the business task and several more calls to retrieve the results of that task. This can cause efficiency concerns and make programming difficult.
- **Problems with stability**: Changes in the business logic may affect the servlet if the interface of the business logic (for example, the EJB) is modified. As a consequence all servlets using that business logic must be changed.
- **Problems with implementing the technology**: There are several possible technologies for how business logic can be implemented. This includes EJB, JDBC, or the Common Connector Framework. In addition to different implementation programming models, each of these technologies has a different way to invoke a request to execute business logic. That means that the servlet has to be aware of all used technologies and has to implement interfaces to them.

Additionally, we may run into problems when calling EJB directly from the servlets. This communication is being based on RMI-IIOP and has significant deployment problems when passing through a firewall.

Facade pattern

Figure 5-8 shows a very simple, fast and convenient way of designing the use cases into the system with regards of the MVC pattern by using the facade pattern (*Patterns In Java*, *Volume 1*. Mark Grand; and *Design Patterns: Elements of Reusable Object-Oriented Software*. Erich Gamma, et al.).

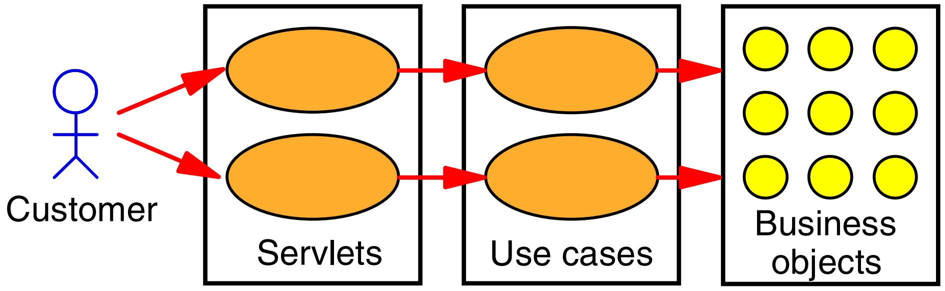


Figure 5-8 Use case facade pattern

This approach is modular: it makes the application more independent of the architecture. For instance, if the Web application had to be used stand-alone or if the entry point had to become different, the use case calls can be combined with the Web entry path, as shown in Figure 5-9.

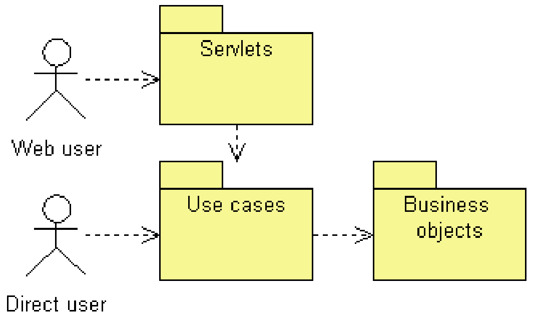


Figure 5-9 Application entry from the Web and from a standalone client

Figure 5-10 shows the PiggyBank facade use case realization diagram.

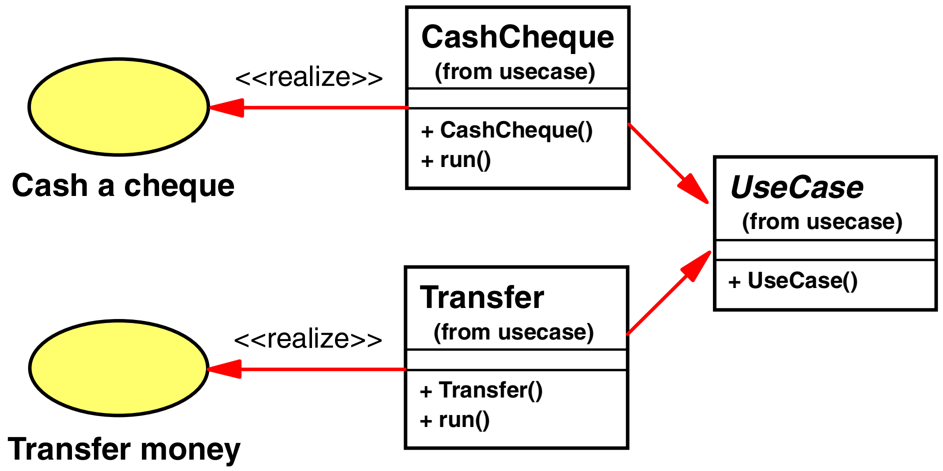


Figure 5-10 PiggyBank facade use case realization diagram

The instantiation policy from the servlets can be in four forms:

1. Every servlet thread can create its own `UseCase` object: the advantage is that it avoids multithreading issues, allows the use of instance variables. This avoids side effects on instance variables. Main disadvantage is that in a heavily loaded environment it can produce object creation overhead.
2. Servlets use a shared pool of `UseCase` instances: this reduces the object creation overhead at the cost of more complexity. It must be ensured that pool objects instance variables are properly initialized before use.
3. Servlets use a singleton `UseCase` instance, which prevents us from using instance variables.
4. `UseCase` methods are declared static, which is a bad practice in general.

We recommend methods 1 and 3.

Stand-alone clients could also want to easily start the use cases in new threads. This can be done by having the use case classes implement the `java.lang.Runnable` interface, as shown in Figure 5-11.

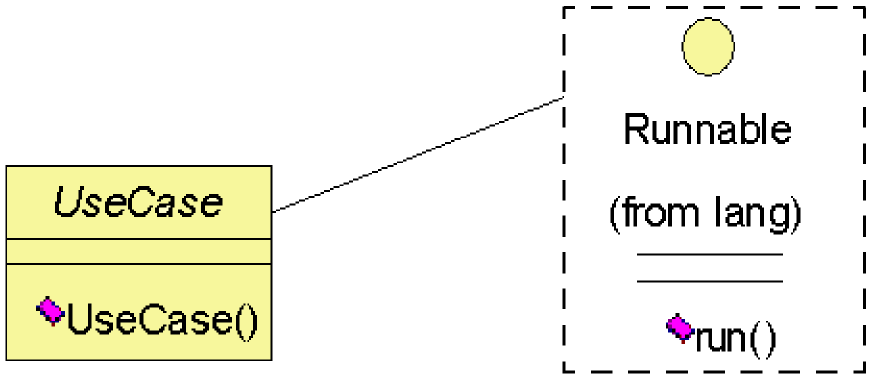


Figure 5-11 Use case multithreading support

Servlet multiplexing

Up to now, we have been designing entry points from the Web into the application using servlets. The section “Servlet mapping” on page 99 outlines the drawback of having multiple servlets, typically one per use case. This can easily be improved by using a multiplexing mechanism based upon a URL parameter called *action* or id, operation—you name it. In this case, there is only one servlet that controls the entire application entries: the controller servlet. The incoming requests therefore contain the servlet name or its alias and an ID as a URL parameter to specify the action to perform in the system:

```
http://hostname/webapp/ControllerServlet?action=transfer
```

At first sight this looks like a bottleneck. Actually, given the servlet thread and instantiation model, this reduces code overhead while keeping full thread flow through the servlet code.

In this model, every use case has an ID defined.

The drawback is that the controller servlet can be very big, forking the execution flow with a long `if-then-else-if` list, such as:

```
String action = request.getParameter("action");
if (action == null) {
   // return error message
} else if (action.equals("transfer")) {
   // call transfer use case
} else if (action.equals("cashCheque")) {
   // cash cheque use case
} else if ...
```

Command pattern

A good way to solve the above problems and a good way to separate the program logic is by the use of *commands*. Commands encapsulate business logic tasks and provide a standard way to invoke the business logic request and access to data using a single round-trip message.

A command is a stylized Java class with the following characteristics:

- A command object corresponds to a specific business logic task, such as a query or an update task. There is one or many commands per use case.
- A command has a simple, uniform usage pattern.
- A command hides the specific connector interfaces and logic from the servlet.
- A command can cache information retrieved during the business task.

Commands are used as shown in Figure 5-12, where the servlet instantiates a command object. Then, the servlet sets the input parameter of the command and executes it. When the command has finished performing the business logic, the result, if any, is stored in the command, so that the servlet or the view can get the result values by interrogating the command object.

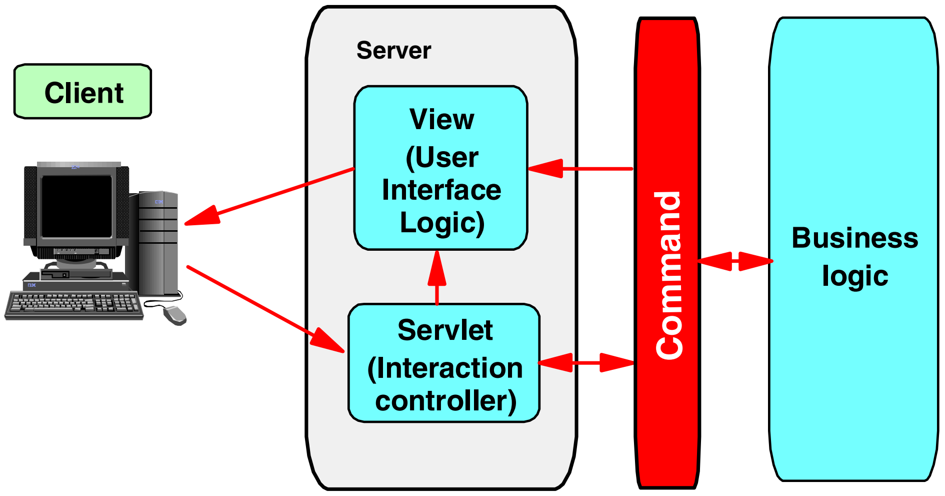


Figure 5-12 Using commands

We recommend that you implement the command as a JavaBean, that is, a Java class with naming restrictions:

- There must be a method for each input property: `void setXxxx(Xxxx val);`
- There must be a method for each output property: `Xxxx getXxxx();`

Display commands

In our programming model the command bean can be interrogated by the standard bean mechanism. That means that a JSP programmer has to have knowledge about Java programming, because the output properties of a command may include complex structures, such as arrays.

To solve the problem we introduce *display commands*. The idea is to eliminate any handwritten code in the JSP, therefore, supporting a development model in which non-programmers can develop, modify, and maintain presentations. Display commands are commands except that they are intended to run locally. A display command calls other commands to run the business logic and encapsulates all the dynamic content of the page by converting the output properties of the executed command into HTML (Figure 5-13).

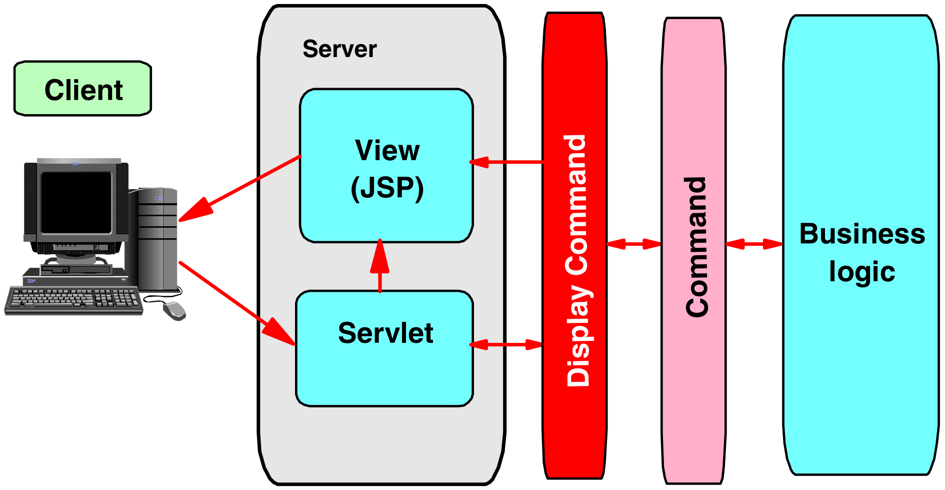


Figure 5-13 Using display commands

For more information and advanced command pattern design, see the redbook *Design and Implement Servlets, JSPs, and EJBs for IBM WebSphere Application Server*, SG24-5754.

Figure 5-14 shows the application developers roles in command development.

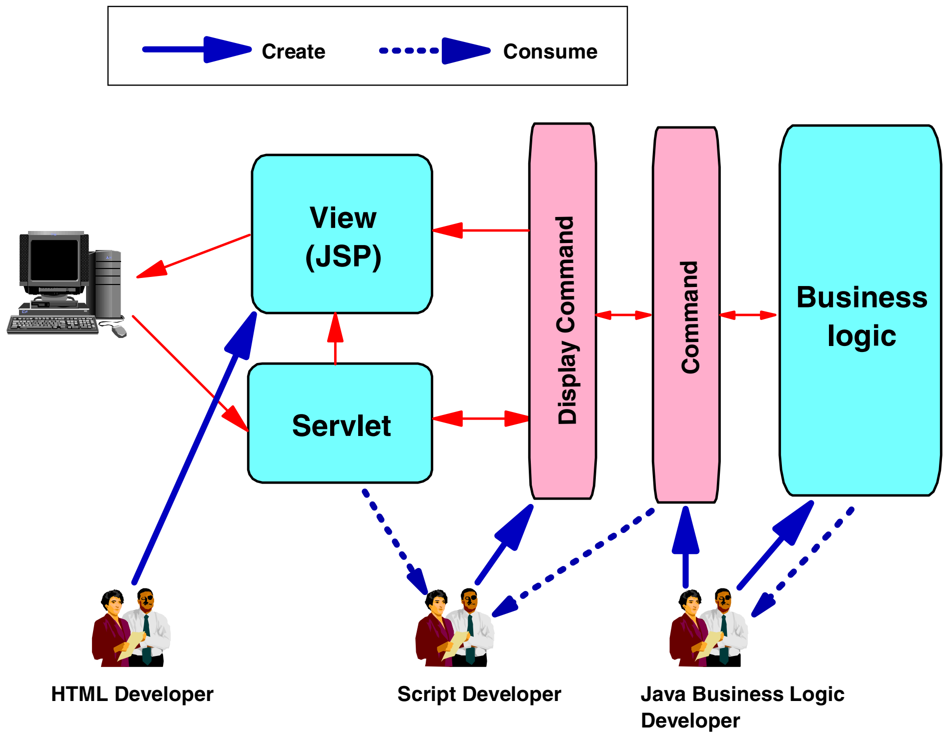


Figure 5-14 Separation of roles and responsibilities

The value of commands

Commands add a valued layer to the e-business architecture and they give us the following advantages:

- Because commands are implemented as serialized objects, they can be shipped to and executed within any Java environment, within any server supporting Java access to its resources and providing a protocol to copy the command between the command client JVM and the server JVM. The protocol does not have to be IIOP. This gives the opportunity of enhancing the performance and the deployment (firewall) by executing the command on the enterprise server.
- Commands allow an application to be partitioned into efficient units of client-server interaction.
- Commands allow the caching of data.

- Because commands are implemented as JavaBeans, it is easy to get access to the output data. Therefore, it is possible to store an executed command bean in a session environment of a servlet and use it by a JSP.
- Commands can even be used or integrated in most Web design tools (for example, WebSphere Studio), because they are implemented as JavaBeans.
- The servlet code is independent of the style of the command's implementation and it is independent of where the command is physically executed.
- Another advantage of choosing the command pattern approach is that it facilitates a cleaner separation of roles in a development team.
- The use of commands leads to a stable boundary between business logic and user-interface logic.

Command granularity

The command pattern can be used to reduce the overhead of cross-tier communication.

There is no perfect answer to the trade-off between good interface design and a reduced communication overhead. Our proposal is, that the requesting tier (the servlet) of a communication has to design the commands exactly to support its tasks. The idea is, that a servlet should only execute one command per invocation which encapsulates all of the controller function except for the HTTP request parsing. That means when implementing an e-business application, the servlet only interprets the HTTP request and executes commands. As a consequence of this approach, we will get as many commands as we have server interactions for a given use case.

Using session beans

To benefit from the power of EJB or if the project already plans to use EJB, use cases can be realized by sessions beans.

A way to structure things is to group them into object categories. If we look into the object modeling chapter, we can see the PiggyBank business objects are:

- Customer
- Account

And if we look at the complete use case diagram, including the CRUD use cases, we can see that they fit into two categories:

- Customer-related use cases
- Account-related use cases

These categories suggest to group the use cases into two packages. We then map these packages to stateless session beans that we call *manager beans*:

- CustomerManager
- AccountManager

Following the mapping consideration, each use case can be modeled as a session bean method. The input parameters are obviously modeled as method parameters and the returned information can be modeled as data beans.

Relationship between command beans and EJB session beans

An obvious question is: Why not use IIOP to session beans to accomplish the same objective as commands? The answer is that commands have the following advantages:

- Command beans handle multiple protocols to accommodate any target server, not just IIOP to EJB servers. This includes, of course, IIOP but also HTTP.
- When acting in a distributed environment, command beans require fewer round-trip messages. For a session bean EJB whose container runs in a separate server, several remote messages are required to do a single logical request:
    1. Look up the home
    2. Narrow the home
    3. Create the session bean instance
    4. Call the method
    5. Destroy the session bean instance

Steps 1 and 2 can often be cached, but there are still three round-trip messages required per instance.

Caching

Using commands, the cross-tier communication is reduced to one round-trip per task. Caching is a technology which can be used to reduce this to even less than one.

Caching is not a new technique; it is a general principle that can be used to reduce cross-tier communication, database queries and computation.

The principle of caching is simple: Do not ask a question twice if you can do it once and save the result to use the second time. This principle can be difficult to implement, because the amount of saved data may become unmanageable and the results can be reused only if they are still accurate.

In e-business applications, there are two types of information that can be cached:

- Formatted information such as whole or partial HTML pages can be cached. This works well when many people need to view the same material presented in the same way, such as on a sports or news site. Caching partial pages adds the flexibility to customize pages for users while still retaining many of the benefits of caching. Because view commands represent partial HTML pages it makes sense to cache those commands.
- Data can be cached. This works well when the same data has to be viewed in different ways. This means that commands (which are executed by the view commands) can be cached.

The two types can be used together. For example, a commerce site may cache product descriptions in a formatted form, while caching customer-profile information as data.

Another common practice is to cache data in the user HTTP session. This can be done after the login, when the session is established. It is very useful to perform such an initial load when the presentation logic plans to use a common information many times. In the PiggyBank application, we often offer the customer to select an account from the account list. It is therefore interesting to cache this information in the user session, and retrieve it from any JSP that needs it. That makes the input screen more convenient without having to call the business logic layer.

External systems integration

In every application, the development scope is limited to a certain boundary representing “the system”. Every element out of this limit is considered as external. If it interacts with the system, it is an actor of the system. Each actor can be:

- A person, in its broadest meaning
- A system

If an actor initiates the interaction with the system, it is an initiating actor of the system. If the system initiates the interaction with the actor, this is a supporting actor. It must be noted that an actor can be both an initiating and a supporting actor of the system, playing different roles.

In this section we are discussing the case when the system needs some external system help to perform its task, initiating the interaction with the appropriate supporting actor. Figure 5-15 shows how to represent this kind of interaction in UML.

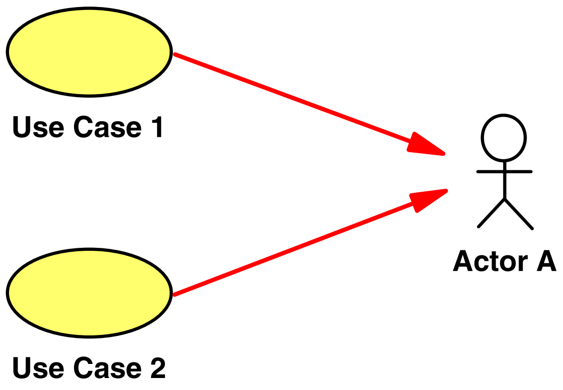


Figure 5-15 Supporting actors

Representing external use cases

As we mentioned, the supporting actors are most of the time systems. Let us assume for the example that the `A` actor is a system.

From the A system point of view, our system is considered as an initiating actor, as shown in Figure 5-16.

Because A is a system, it can be structured in use cases too, each of them delivering a result of value to the corresponding initiating actor, that is, our system.

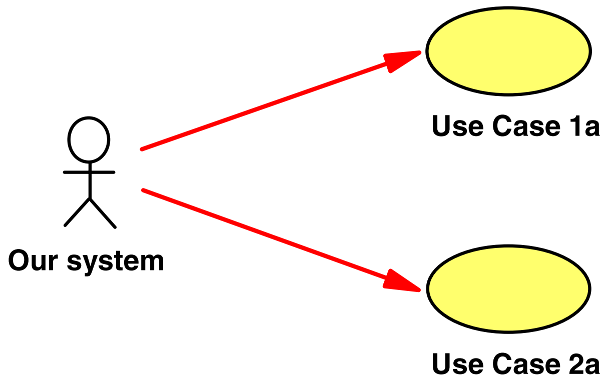


Figure 5-16 External system point of view

There is a tight relationship between our system use cases and the supporting actors relationships. Most of methodologies recommend to represent external use cases relationships using internal relationships with proxy use cases (Figure 5-17).

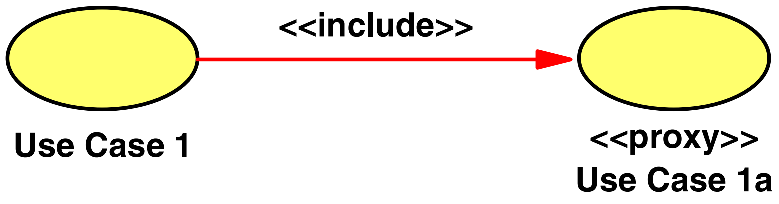


Figure 5-17 Proxy use case

Now external use cases have their equivalents inside our system.

For the PiggyBank, the only supporting actor is CityBank. This provides one supporting use case to the system: *validate a cheque*. This can be modeled as a proxy use case included in the *Cash a cheque use case* (Figure 5-18).

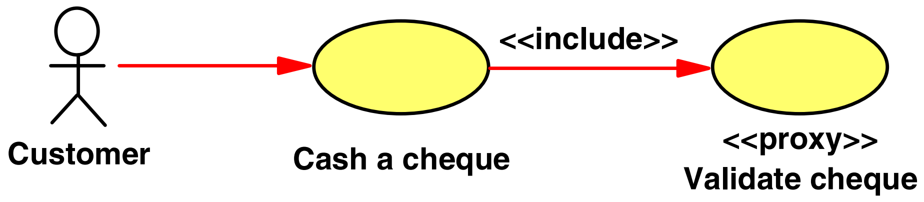


Figure 5-18 PiggyBank proxy use case

Realizing proxy use cases

To realize the proxy use cases and to isolate the communication with the supporting actors into boundary classes, we recommend to apply the facade pattern. For each supporting actor, one class—you can call it an *agent* or a *proxy*—offers single-point access to the supporting actor use cases in the form of methods.

This class will be responsible for all the communication aspects regarding the corresponding supporting actor. It can use helpers classes, connection pooling and other advanced objects to perform its task. These all form a boundary layer against the supporting actors.

Figure 5-19 represents the proxy use case realization for PiggyBank. We can see the realization class responsible for the communication with the CityBank. Its name concatenates the supporting actor name with the `Agent` postfix. It has one method to validate a cheque against the CityBank. All communication and business considerations are hidden from the rest of the PiggyBank system.

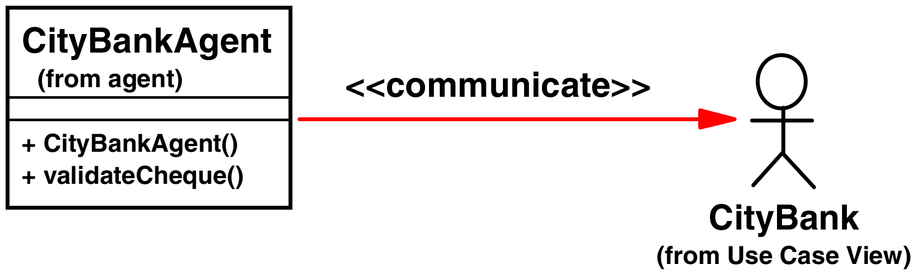


Figure 5-19 PiggyBank proxy use case realization

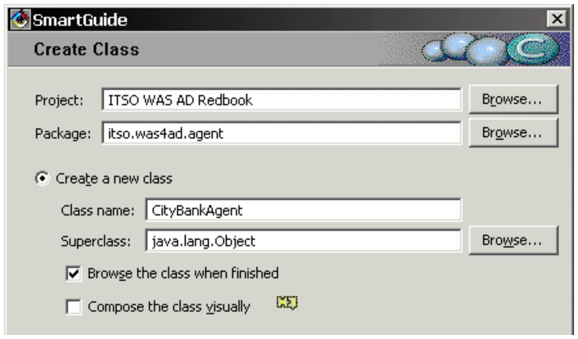
Representing agents in VisualAge for Java

The business analysis coordinator creates—possibly with the help of a more experienced VisualAge for Java user—an agent package in VisualAge for Java:

- Right-click on the project (ITSO WAS AD Redbook) -> *Add* -> *Package* and name the new package `itso.was4ad.agent`.
- Add all the business analysts, designers, developers in the project as as additional users to the package.

The business analyst, who has identified a supporting actor and its proxy use cases, creates an agent class:

- Right-click on `itso.was4ad.agent` -> *Add* -> *Class* and name the new class `CityBankAgent` (Figure 5-20).


SmartGuide

Create Class

Project: ITSO WAS AD Redbook

Package: itso.was4ad.agent

(●) Create a new class

Class name: CityBankAgent

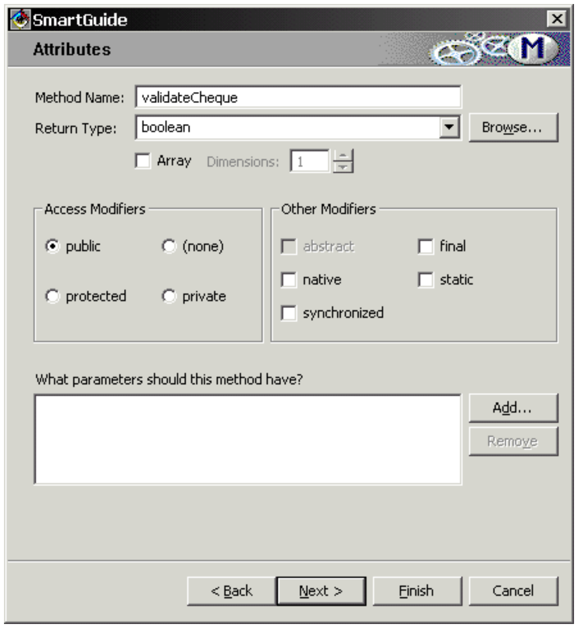
Superclass: java.lang.Object

[✓] Browse the class when finished

[ ] Compose the class visually

Figure 5-20 Create an agent class

- Right-click on `CityBankAgent` -> *Add* -> *Method*. Click *Next*. Name the method *validateCheque* (Figure 5-21). Make it return a *boolean* and click *Add* to add one parameter.


SmartGuide

Attributes

Method Name: validateCheque

Return Type: boolean

[ ] Array Dimensions: 1

Access Modifiers: (●) public ( ) (none) ( ) protected ( ) private

Other Modifiers: [ ] abstract [ ] final [ ] native [ ] static [ ] synchronized

What parameters should this method have?

Figure 5-21 Create an agent method

- In the *Parameters* window (Figure 5-22), name the parameter `cheque` and set the type radio button to *Reference Types* and enter *Cheque* in the field below. Click *Add* once and *Close* then *Finish*.

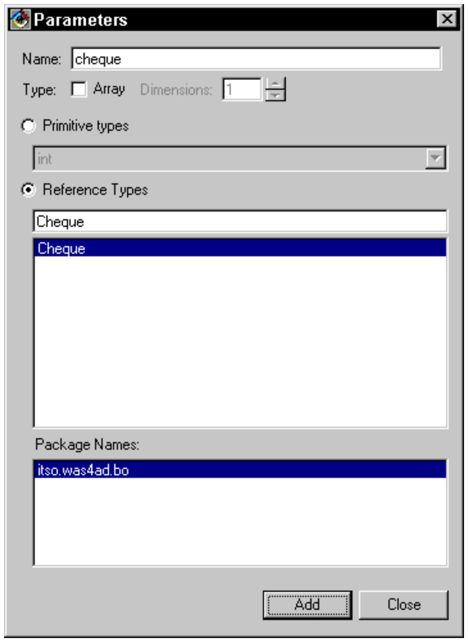


Figure 5-22 Adding a use case parameter

Designing the user interface

A measure of the success of a software solution is its capability in supporting the tasks and activities users perform to complete their job. A thorough understanding of these tasks and activities, the environment in which they are performed, the tools used to do these tasks, and the manner in which these tasks "fit together" provide a key input for the solution's high and low-level user-interface design.

The purpose of this section is to describe a comprehensive technique for collecting and analyzing data for user-intensive use case models. Specifically, this technique applies to those use cases that have significant user interaction and have a critical impact on user satisfaction and performance.

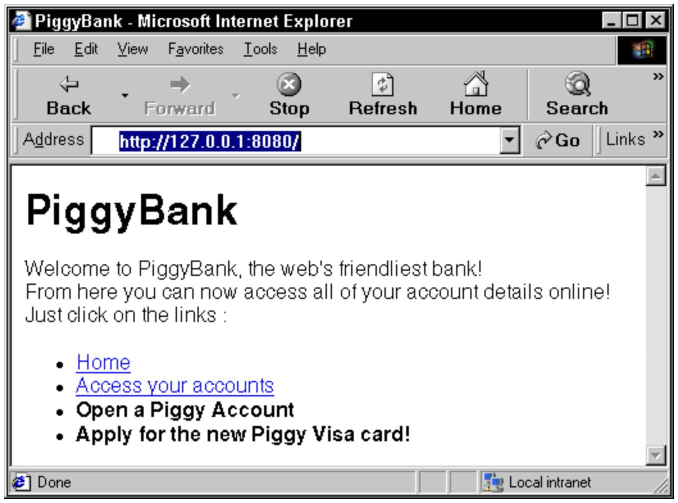
Like use case modeling where the actors are human users of the system, user interface analysis is a means to understand the interaction between users and their environment. It is an end to end process description that details the steps for a user-system interaction. User interface analysis method focuses heavily on the process and techniques for collecting data from users. By integrating task analysis and use case modeling techniques, business analysts have a better understanding of how users do their work currently and how they might accomplish their work in the future and how this knowledge can be communicated in a better use case model.

Screen composition

Both activities are tightly tied together and should be performed in a parallel fashion by business analysts. Therefore, the user interface technical aspects should be kept as simple as possible in a first step. That is, we do not deal with the complicated aspects of designing an HTML page. This is a job for the Web developer. For the requirements modeling phase of the project, the latter role is stand-by and waits for business analyst UI output to start its more technical and creative activity. The business analyst can focus on the business and functional aspects of the user interface, and the main concerns are:

- What information and input elements are shown on the screens
- The navigation between the screens
- Informal description of the information transfers between the screens

This activity should produce very simple screens (Figure 5-23).


PiggyBank - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search
Address http://127.0.0.1:8080/ Go Links
Done
Local intranet

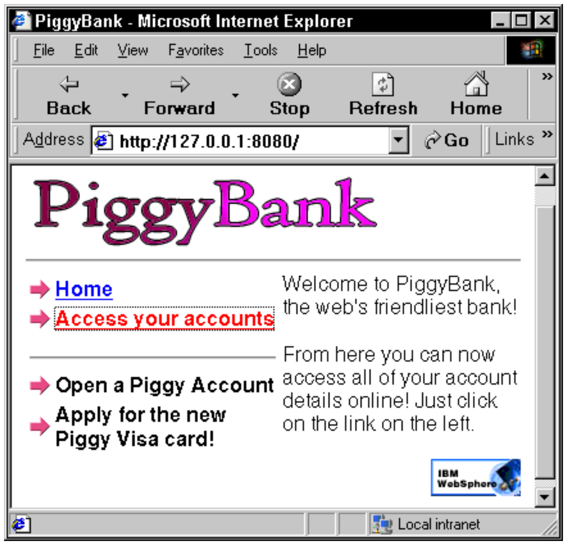
PiggyBank

Welcome to PiggyBank, the web's friendliest bank!
From here you can now access all of your account details online!
Just click on the links :

- Home
- Access your accounts
- **Open a Piggy Account**
- **Apply for the new Piggy Visa card!**

Figure 5-23 Simple HTML screen

This deliverable can later be refined by Web developers to produce more detailed HTML screens (Figure 5-24).


PiggyBank - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home
Address http://127.0.0.1:8080/
Go Links
PiggyBank
Home
Access your accounts
Open a Piggy Account
Apply for the new Piggy Visa card!
IBM WebSphere
Local intranet

Welcome to PiggyBank, the web's friendliest bank!

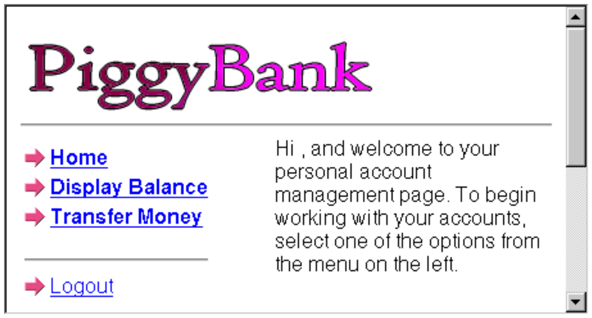
From here you can now access all of your account details online! Just click on the link on the left.

Figure 5-24 Detailed HTML screen

Chapter 10, “Development using WebSphere Studio” on page 237 shows how to perform screen composition with WebSphere Studio.

Navigation

Typically, the application offers a main menu presenting the list of the use cases the user can initiate, possibly in a hierarchical manner depending on the size of the application. Figure 5-25 shows the PiggyBank application main menu.



Hi , and welcome to your personal account management page. To begin working with your accounts, select one of the options from the menu on the left.

Figure 5-25 PiggyBank main menu

If login is required, as in most Web applications, the main menu can be accessed through the login screen. The submit buttons send the login request to the application server, which handles it, authenticates if necessary and establishes a new user session. Session management is covered in “Web application server maintained data flow sources” on page 48.

After the main menu allows the user to select a use case among a list, zero, one or several input screen can be presented to the user, who can then initiate the use case by submitting the complete request. That calls the corresponding use case, which usually performs OK and returns the result, typically a page with the summary of the transaction to the customer and a link to the main menu (Figure 5-26).

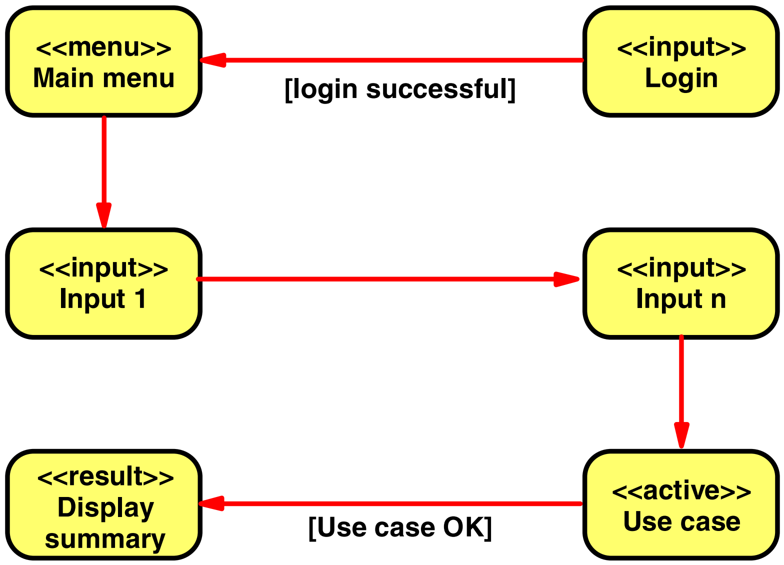
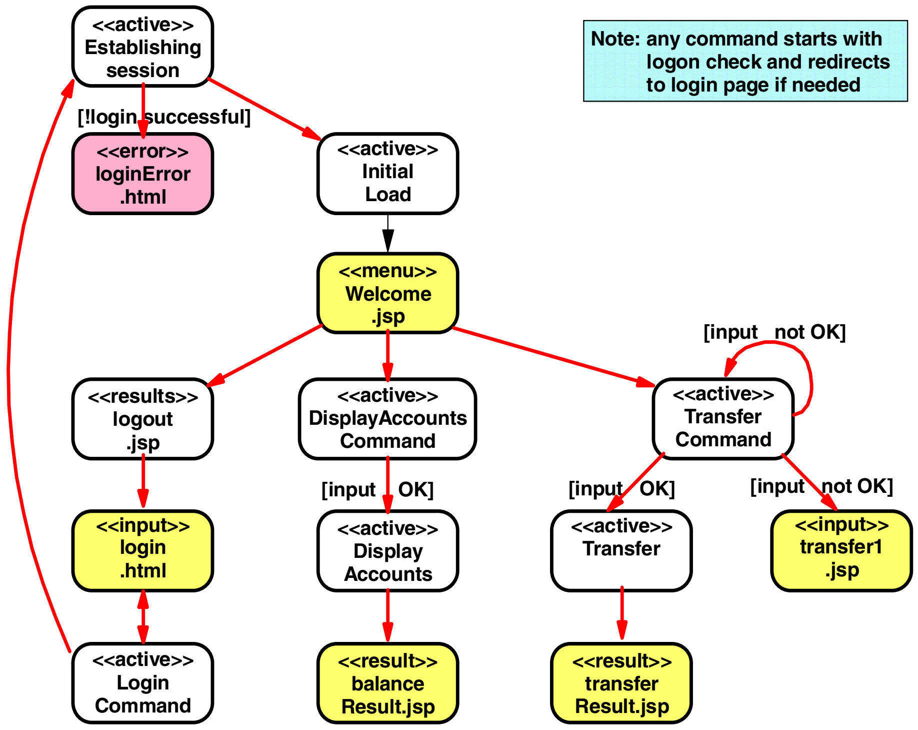


Figure 5-26 Typical navigation

To make the navigation easier, the main menu can be permanently presented to the user, as an HTML frame or a component included by every page. JSP specifications provides two types of include mechanisms to make that possible:

- `<jsp:include page="relativeURL" flush="true">`
- `<%@ include file="relativeURL" %>`

Figure 5-27 shows the PiggyBank screen navigation in the form of a UML state diagram.



<<active>> Establishing session
[!login successful]
<<error>> loginError .html
<<active>> Initial Load
<<menu>> Welcome .jsp
Note: any command starts with logon check and redirects to login page if needed
[input not OK]
<<results>> logout .jsp
<<active>> DisplayAccounts Command
<<active>> Transfer Command
[input OK]
[input OK]
[input not OK]
<<input>> login .html
<<active>> Display Accounts
<<active>> Transfer
<<input>> transfer1 .jsp
<<active>> Login Command
<<result>> balance Result.jsp
<<result>> transfer Result.jsp

Figure 5-27 PiggyBank navigation

Use case commands

Now, we introduce the command concept into the previous design principles. The basic integration rule is to provide one command per input screen.

We begin with the case where the input screen submission leads to a use case. We call such a command a *use case command*. Figure 5-28 represents the navigation state diagram adaptation for a single input screen.

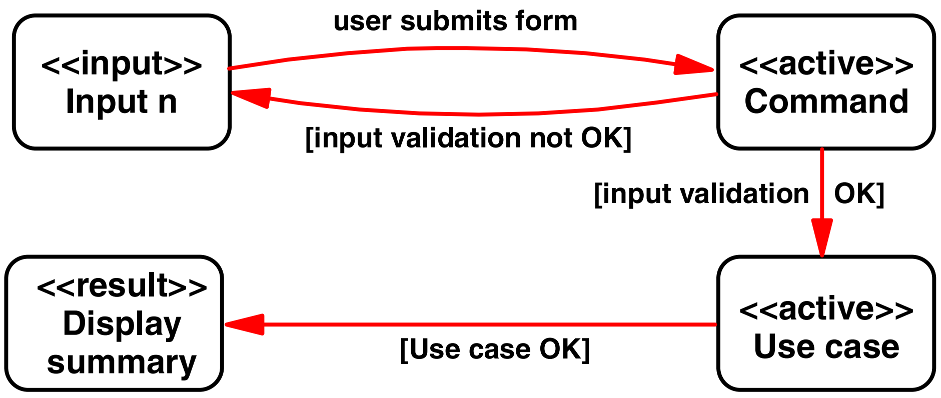


Figure 5-28 Use case command navigation

Intermediate commands

Now we consider the case where the input process leading to a use case is performed through several input screens, and an input screen submission leads to the next input screen.

We call such a command an *intermediate command*. In this particular case, the command just validates and forwards to the next input screen; it does not start a use case.

The navigation state diagram is therefore a little bit different (Figure 5-29).

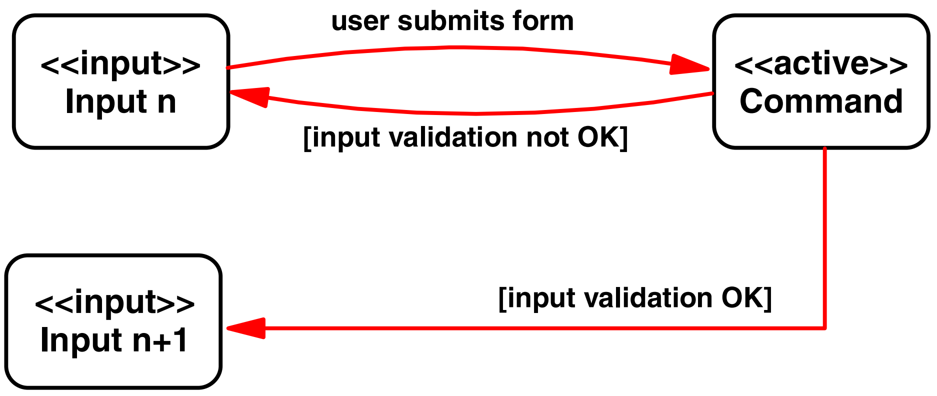


Figure 5-29 Intermediate command navigation

An interesting discussion can be held about the information transmission through the input screens up to the use case. There are several possibilities:

- The information is saved in the session context. The drawback is this can lead to problems if the user navigates with multiple browser windows.
- The information is stored into a stateful session bean. The drawback is this can lead to EJB instantiation overhead.
- The information is transmitted from page to page using hidden form fields. This is very simple and supports user navigation using multiple browser windows. The drawback is this can lead to network communication overhead.

For more information see Chapter 1, “WebSphere programming model” on page 3.

6

Modeling and code generation

In this chapter we discuss how we can use the model created from our design activities to generate code for our application using the Rational Rose product. We describe the concept of “round tripping” and discuss integration with other development tools described in this book, particularly VisualAge for Java.

We also introduce the J2EE support included with Rose, and describe how we can use it to model and then generate code for one of the EJBs from our PiggyBank application.

Code generation

Once we have a model in Rose that describes the initial design of our application, the next step is to implement the design in Java code. A naïve approach would be to create Java source files for each class identified in the Rose model by hand. Depending on the number of classes involved, this may turn out to be a rather tedious and somewhat error-prone activity.

Fortunately Rose allows us to avoid this drudgery by taking our model and automatically generating Java classes from it. The generated code directly reflects the model, including the associations between classes and the attributes and operations contained within them.

Round tripping

The next step is to take the code generated from our initial model and develop it into our application. During the course of development the classes will be modified substantially—often adding new methods and fields or modifying those that exist already. If the Rose model is to continue to be of use to us as a source of information about the application design it is essential that changes in the code are reflected in the model. To do this by hand requires discipline on the part of the developer—even if the developer does remember to update the model the interruption to his train of thought caused by starting up Rose and updating the model can adversely affect productivity.

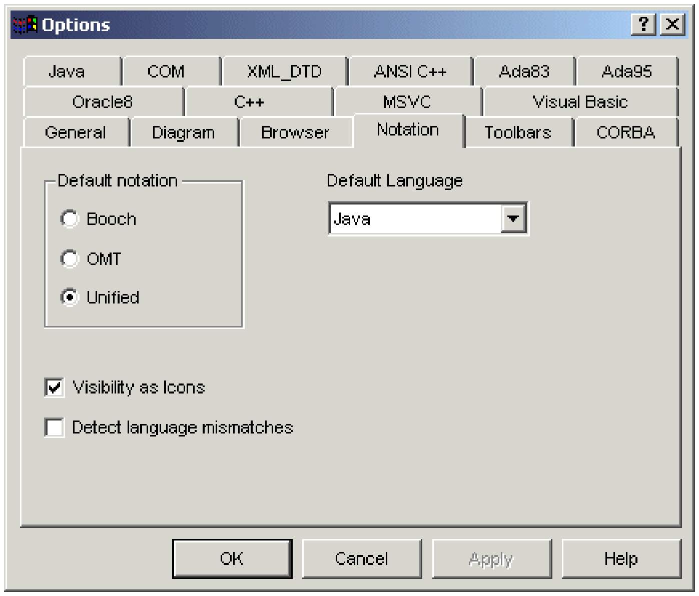
Fortunately there is a solution to this problem in the form of *round tripping*. This process combines the code generation features of Rose with its reverse-engineering capabilities—the ability to create a model by examining Java code. In a round tripping scenario code is generated from a model and modified by a developer. The developer's changes are then reverse-engineered, updating the original model in Rose. The term round tripping describes the round trip from model to code, and back to the model again.

In the rest of this chapter we discuss how we can use Rose to generate and reverse-engineer our application code. We also take a look at how to integrate Rose with VisualAge for Java, and how we can use the J2EE capabilities of Rose to model and generate code and deployment descriptors for EJBs.

Setting the default language for Rose

Rose supports many target languages for development. All of our WebSphere development is performed with Java, so we can alter the Rose options to specify Java as the default language. This enables some additional context menu options that we will use—this setting also enables Rose to automatically map classes in our logical view to Java components during code generation.

We change the default language using the options dialog, which we display by making the menu choice *Tools -> Options*. Once in the dialog, we select the *Notation* tab, which is illustrated in Figure 6-1. Change the default language to Java and click *OK* to save the changes.



Default notation: ( ) Booch ( ) OMT (●) Unified

Default Language: Java

[✓] Visibility as Icons

[ ] Detect language mismatches

Figure 6-1 Notations tab in the Rose options dialog

Code generation and reverse engineering

First we describe how to perform code generation to and reverse engineering from Java source files located in the file system. This basic path allows us to integrate with any development environment. Later on we describe a much more convenient way to integrate with the VisualAge for Java IDE.

Code generation

For this example we use some basic code from the data bean layer of our application. We start with the class diagram illustrated in Figure 6-2.

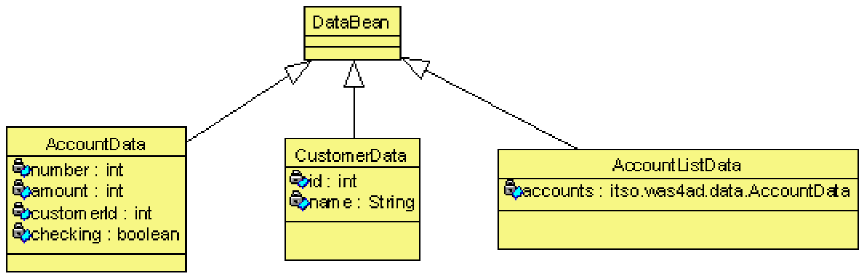
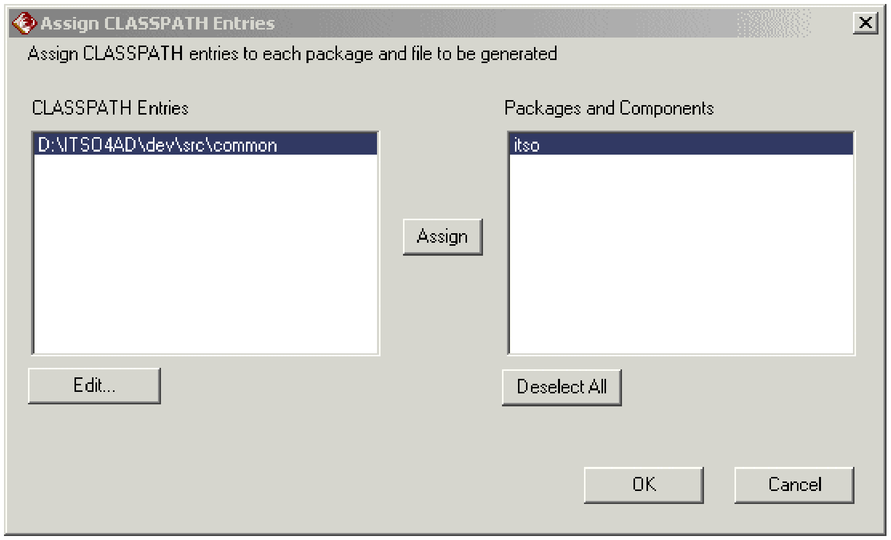


Figure 6-2 Initial data bean class diagram

We generate Java code from Rose by selecting the classes for which we want to generate code, and *J2EE / Java -> Generate Code* from the context menu. In this case we select all four classes in the diagram. Because this is the first time we have generated code for this component, the dialog shown in Figure 6-3 appears. This dialog allows us to specify the destination for the generated code. The destination must be an entry in the class path defined in the Rose project specification dialog—if the desired path is not included already, we can add it by clicking *Edit*.


Assign CLASSPATH Entries

Assign CLASSPATH entries to each package and file to be generated

CLASSPATH Entries

D:\ITSO4AD\dev\src\common

Packages and Components

itso

Figure 6-3 Assign class path entries dialog

We highlight the appropriate entries on both sides of the dialog, and click *Assign*. We then click *OK* to generate the code, shown in Figure 6-4.

Tip: This assignment is remembered for the entire Rose project. If you want to generate code for each module into separate directories, as described in Chapter 9, “Development using the Java 2 Software Development Kit” on page 183, you may find it easier to create a separate project in Rose for each module instead of having the entire model in one project.

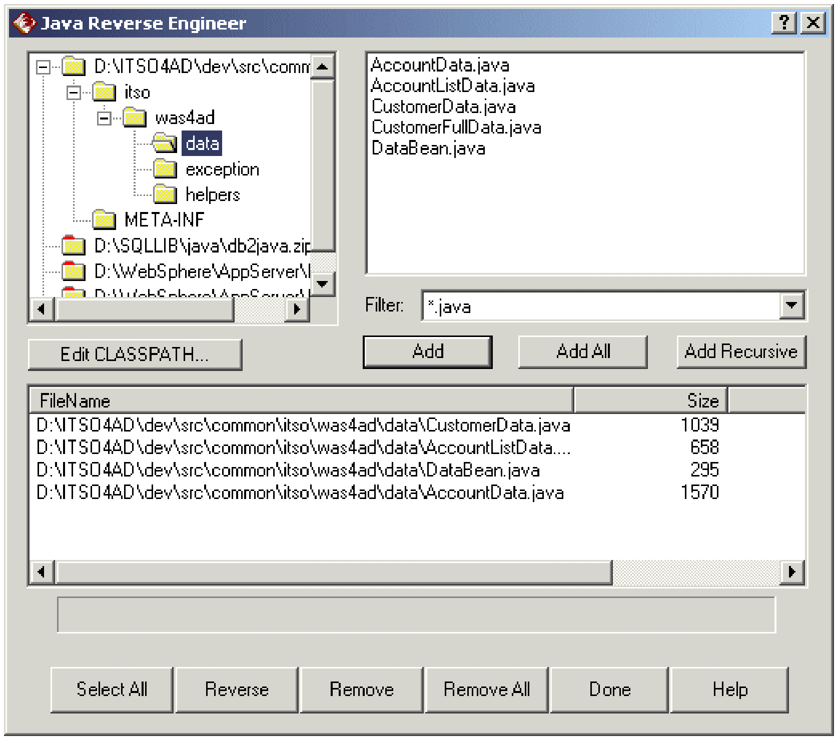
```
//Source file: D:\\ITSO4AD\\dev\\src\\common\\itso\\was4ad\\data\\CustomerData.java

package itso.was4ad.data;

import java.io.Serializable;

/**
 * Data object representing a customer
 */
public class CustomerData extends DataBean implements java.io.Serializable {
   private int id;
   private String name;

   /**
    * @roseuid 3B5DB6270126
    */
   public CustomerData() {
   }
   /**
    * Access method for the id property.
    * @return   the current value of the id property
    */
   public int getId() {
      return id;
   }
   /**
    * Access method for the name property.
    * @return   the current value of the name property
    */
   public String getName() {
      return name;
   }
}
```

Figure 6-4 CustomerData class generated by Rose

Note: The getter methods in the source file were generated because we defined the attributes in the Rose model as simple, read-only bean properties.

Reverse engineering

We reverse engineer Java code by selecting *Tools -> Java / J2EE -> Reverse Engineer* from the main Rose menu, or *Java / J2EE -> Reverse Engineer* from the context menu obtained by right-clicking on a component in the model.



FileName	Size
D:\ITSO4AD\dev\src\common\itso\was4ad\data\CustomerData.java	1039
D:\ITSO4AD\dev\src\common\itso\was4ad\data\AccountListData....	658
D:\ITSO4AD\dev\src\common\itso\was4ad\data\DataBean.java	295
D:\ITSO4AD\dev\src\common\itso\was4ad\data\AccountData.java	1570

Filter: *.java

Figure 6-5 Selecting code to reverse engineer

We browse through the project class path in the reverse engineer dialog (Figure 6-5) to select the source files we want to reverse engineer. We click *Add* to add files in the top right panel to the list at the bottom, then *Select All* and *Reverse* to reverse engineer the code.

Tip: If you select classes you want to reverse engineer in the model before you open the dialog, the selected classes are automatically added to the bottom panel.

We reverse-engineered the final version of the code included in the sample code to illustrate this point. When we updated the model in Rose, we ended up with the diagram shown in Figure 6-6.

The association between `AccountListData` and `AccountData` was not included in our original model. Reverse engineering the code added it to our model—we added it to our class diagram by dragging the association from the browser pane onto our diagram.

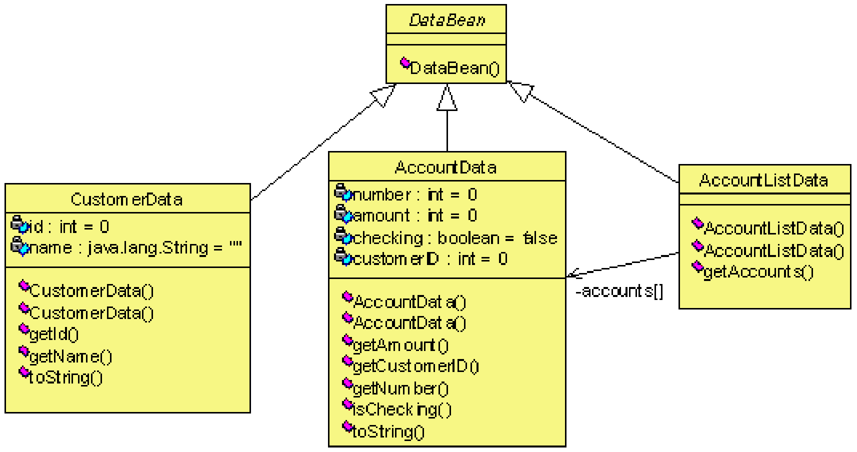


Figure 6-6 Updated data bean class diagram

Class paths and reverse engineering

If you attempt to reverse engineer the complete PiggyBank application you may find you encounter errors caused by the reverse engineering code not having access to supporting code required by the application. This includes the WebSphere logging code, as well as the J2EE libraries—although the Rose model knows about the EJB and servlet APIs the reverse engineering process does not.

To fix this problem you should open the Java project specification dialog (*Tools -> Java / J2EE -> Project Specification*) and add the following JARs to the class path:

```
%WAS_HOME%\lib\j2ee.jar                    <== J2EE library
%WAS_HOME\lib\ras.jar                      <== WebSphere logging library
```

In the examples above, `%WAS_HOME%` should be replaced by the directory where you installed the WebSphere Application Server software.

Integration with VisualAge for Java

There are several ways to exchange code between Rational Rose and VisualAge for Java:

- VisualAge for Java Rational Rose bridge
- XMI toolkit
- Plain Java files

VisualAge for Java Rational Rose bridge

The VIsualAge for Java Rational Rose bridge is a VisualAge tool extension that is included with the Rose product. The bridge allows code to be exported directly from Rose to VisualAge, and from VisualAge directly back into Rose, in a single operation.

Installing the Rose bridge

The Rose installation program checks to see if VisualAge for Java is installed, and offers the bridge as an installation option if VisualAge for Java is detected. Because of this, we recommend you install VisualAge for Java before installing Rose. If you install Rose before installing VisualAge for Java, you have two choices:

- Uninstall and reinstall Rose
- Download and execute a separate bridge installation program from Rational

When we were developing this book we discovered that Version 4.0 of VisualAge for Java—the version we were using—was so new that it was not recognized or officially supported by Rose. We used the following procedure to fool the Rose installer into believing that a supported version of VisualAge was installed:

- Export VisualAge for Java Version 4.0 registry entries
- Edit exported file to change Version 4.0 to Version 3.5
- Import edited file
- Install Rose
- Delete imported Version 3.5 settings

Although Rose does not at this time officially support VisualAge for Java Version 4.0, we encountered no problems using the bridge—we believe there are no significant changes to the VisualAge for Java tool API that would prevent the bridge from functioning in the new version. If you do encounter problems, however, you may have to wait until Rational officially supports the new version before you will be able to obtain support for the feature.

Important: These instructions involve editing Windows registry settings. This must be undertaken with caution, because a mistake can render your machine unusable. Follow these instructions at your own risk—we strongly recommend you do so only if you fully understand the procedure and the risks involved.

Export Version 4.0 registry entries

We used the Windows `regedit` tool to locate the VisualAge for Java registry entries at:

```
HKEY_LOCAL_MACHINE\SOFTWARE\IBM\VisualAge for Java for Windows
```

In the tree view you should see a single key in the tree view, named `4.0`. Select the key, then choose the menu option *Registry -> Export Registry File* to export the contents of the key to a file. We named our file `C:\TEMP\vaj.reg`.

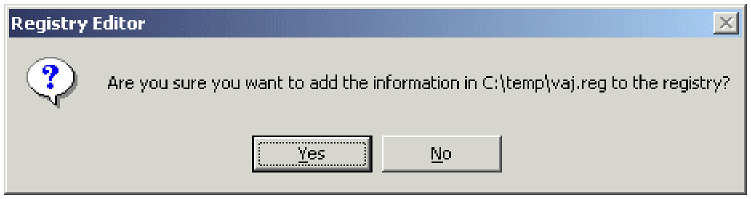
Edit exported file

We used Notepad to edit the exported file. **Do not** double click on the file. Edit the file by right-clicking and choosing *Edit* from the context menu, or by dragging it to your favorite text editor.

In the editor, perform a global find and replace, replacing **all** instances of the version ID `4.0` with `3.5`. Save the file.

Import edited file

We can now merge the edited file into the registry. Double click on the file, and answer *Yes* to the dialog in Figure 6-7.



Are you sure you want to add the information in C:\temp\vaj.reg to the registry?

Figure 6-7 Importing the edited registry file

If you now check the registry you should find an additional `3.5` key along with the `4.0` key we saw earlier. Do not start VisualAge for Java with this extra key in the registry.

Install Rose

Follow the normal Rose installation procedure—the installer should detect that VisualAge for Java is installed and automatically select the option to install the bridge.

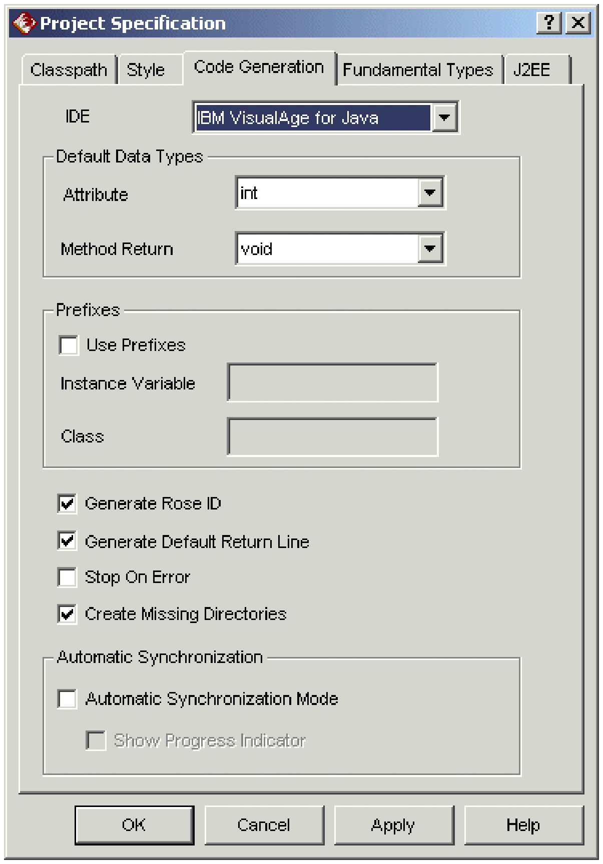
Delete imported version 3.5 settings

Once Rose is installed successful we can delete the fake version 3.5 key we created. Using the Windows `regedit` tool, we simply locate the key, highlight it and select *Delete* from the pop-up menu. Take care to only delete the key we created, and nothing else.

Configuring the Rose bridge

Before we can use the Rose bridge to transfer code between Rose and VisualAge we must configure Rose to use the bridge.

We do this in the Rose Java project specification dialog, accessed from the menu via *Tools -> Java / J2EE -> Project Specification* and clicking on the *Code Generation* tab (Figure 6-8). Change the *IDE* property to specify VisualAge for Java, rather than the internal editor.



IDE: IBM VisualAge for Java

Default Data Types
Attribute: int
Method Return: void

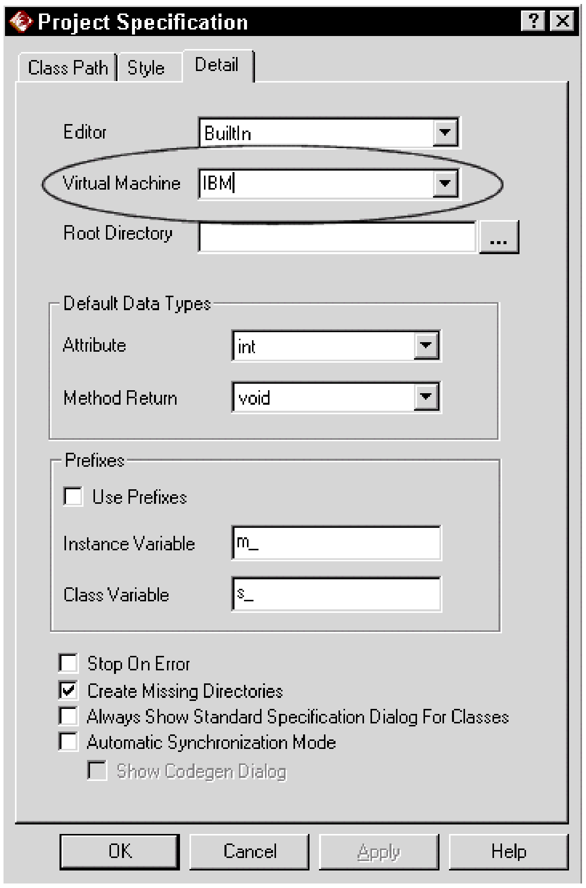
Prefixes
☐ Use Prefixes
Instance Variable:
Class:

☑ Generate Rose ID
☑ Generate Default Return Line
☐ Stop On Error
☑ Create Missing Directories

Automatic Synchronization
☐ Automatic Synchronization Mode
☐ Show Progress Indicator

Figure 6-8 Setting the IDE in the Rose project specification dialog

Earlier versions of Rose have a slightly different version of this dialog, where you enable integration with VisualAge for Java by altering the *JVM* property using the *Detail* tab (Figure 6-9).



Editor: BuiltIn
Virtual Machine: IBM
Root Directory:

Default Data Types
Attribute: int
Method Return: void

Prefixes
☐ Use Prefixes
Instance Variable: m_
Class Variable: s_

☐ Stop On Error
☑ Create Missing Directories
☐ Always Show Standard Specification Dialog For Classes
☐ Automatic Synchronization Mode
☐ Show Codegen Dialog

Figure 6-9 Setting the JVM in earlier versions of Rose

Starting the Rose plugin in VisualAge for Java

Rose communicates with VisualAge for Java via a plugin written to use the VisualAge for Java tool API. Before we can transfer code between Rose and VisualAge for Java we must first start the Rose plugin.

We can do this from the VisualAge for Java Quick Start dialog shown in Figure 6-10. Open the dialog by pressing F2 or selecting *File -> Quick Start* from the menus. In the dialog select *Basic -> Rational Rose VAJ Link Plugin Toggle* and click *OK* to start the plugin.

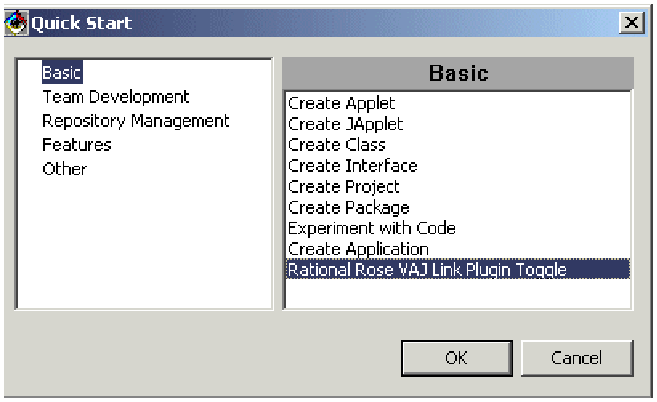
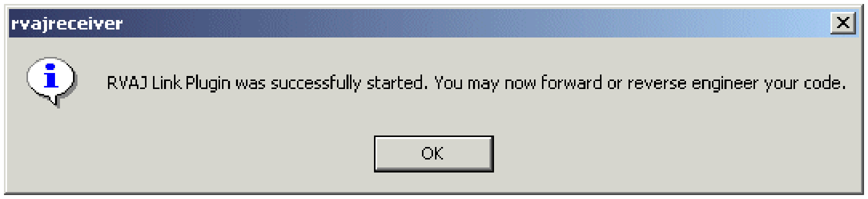


Figure 6-10 VisualAge for Java Quick Start dialog

After a short period the message illustrated in Figure 6-11 appears—VisualAge for Java is now ready to exchange code with Rose.



RVAJ Link Plugin was successfully started. You may now forward or reverse engineer your code.

Figure 6-11 Rose VisualAge for Java link plugin startup message

Generating code into VisualAge for Java

Now we are ready to generate some code from our Rose model and import it into VisualAge for Java. For this example we will use the same basic code from the data bean layer of our application that we used in our earlier code generation example. We start with the class diagram illustrated in Figure 6-12.

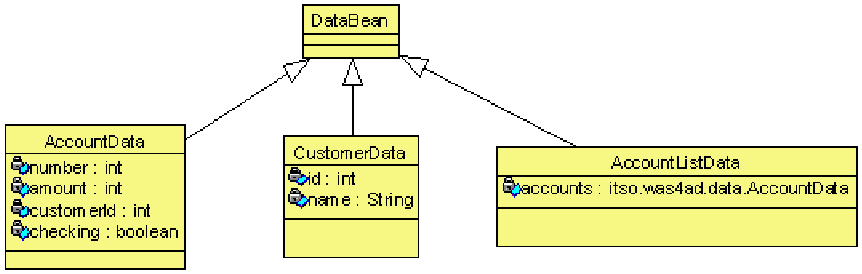


Figure 6-12 Initial data bean class diagram

We generate Java code from Rose by selecting the classes we want to generate code for and *J2EE / Java -> Generate Code* from the context menu. In this case we select all four classes in the diagram. Because this is the first time we are generating code into VisualAge for Java, we must specify the VisualAge for Java project we want to use in the dialog that appears (Figure 6-13).

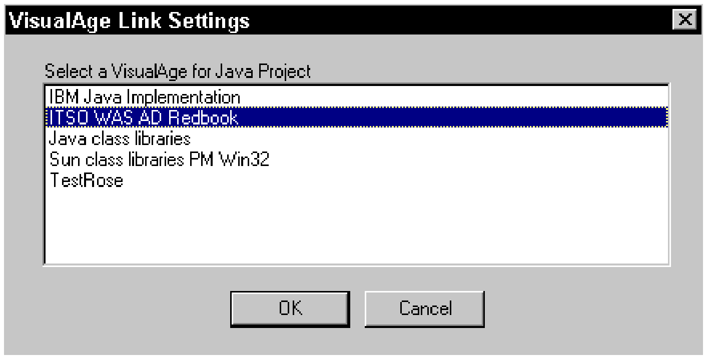


Figure 6-13 Selecting the VisualAge for Java project for the generated code

We select the appropriate project and click *OK*. The selected project is remembered for future use. The code for our classes is generated and imported into the appropriate project in VisualAge for Java.

Reverse engineering code from VisualAge for Java

As we develop our application we update the code generated by Rose in VisualAge for Java. As we make our changes, the code in VisualAge for Java diverges from the model we created in Rose. To prevent the Rose model from becoming out of date with respect to the code we must update the model with the changes we have made. Doing this by hand is a tedious process—fortunately we can use the bridge to reverse engineer our code changes and merge them back into the model.

We can do this from VisualAge for Java by selecting the classes we want to update in the model and *Tools -> Rational Rose Update Model* from the context menu.

We updated our code in VisualAge for Java with the final version included in the sample code to illustrate this point. When we reverse-engineered the updated code into Rose we ended up with the diagram shown in Figure 6-14.

The association between `AccountListData` and `AccountData` was not included in our original model. Reverse engineering the code added it to our model—we added it to our class diagram by dragging the association from the browser pane onto our diagram.

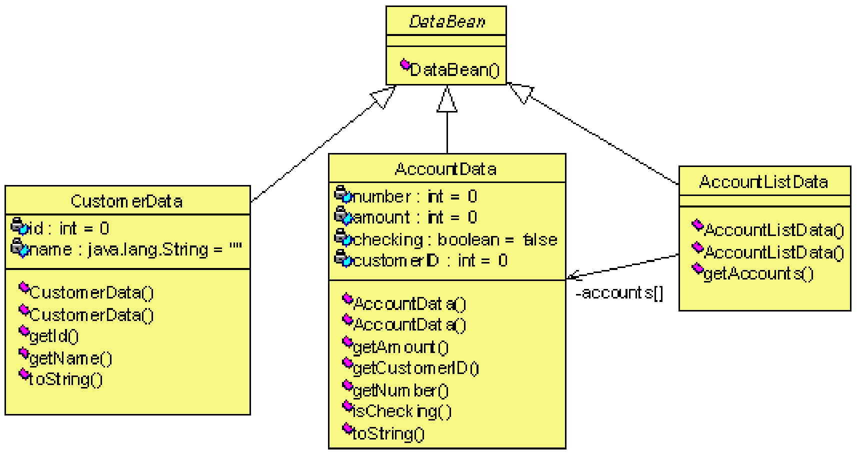


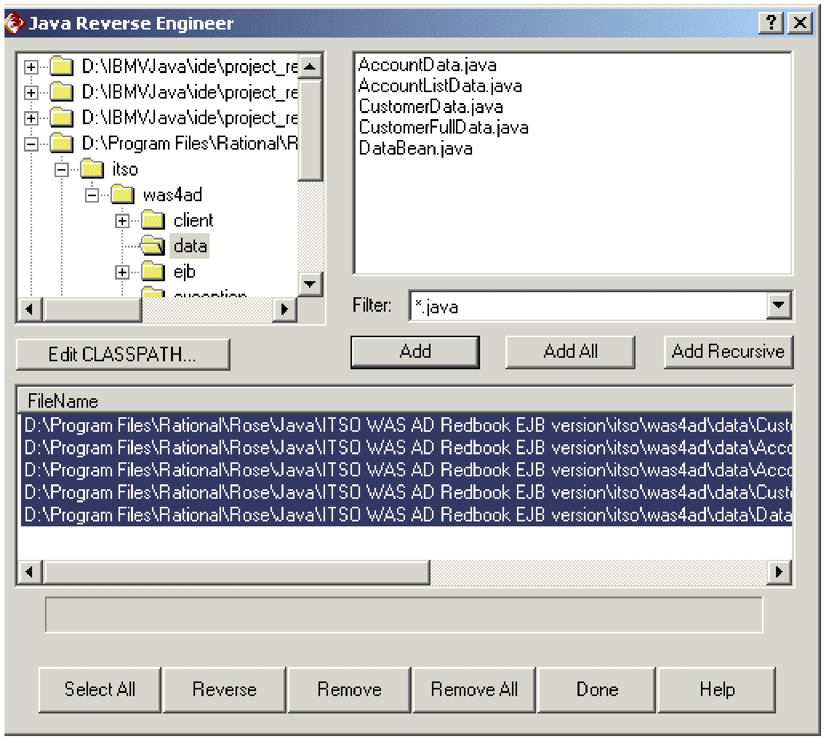
Figure 6-14 Updated data bean class diagram

We can also initiate the reverse-engineering of code from within Rose. First we select *Tools -> Java / J2EE -> Reverse Engineer* from the Rose menu.

In the reverse engineer dialog (Figure 6-15) expand the class path entry (shown on the top left) that corresponds to the VisualAge for Java project we are working with. Locate the package directory containing the files you want to reverse engineer—when you select the package directory the source files contained within are displayed in the panel at the top right.

Select the files you want to reverse engineer and click *Add* to add them to the list of files in the panel at the bottom of the dialog. Alternatively you can add all files in a package directory by clicking *Add All*, or recursively add all files in the directory and every sub-directory by clicking *Add Recursive.*

When all of the files you want to reverse engineer are included in the bottom panel click *Select All* to select all the files, then *Reverse* to start the process. When all files have been reverse-engineered, click *Done* to return to your updated model.



AccountData.java
AccountListData.java
CustomerData.java
CustomerFullData.java
DataBean.java

Filter: *.java

```
D:\Program Files\Rational\Rose\Java\ITSO WAS AD Redbook EJB version\itso\was4ad\data\Cust
D:\Program Files\Rational\Rose\Java\ITSO WAS AD Redbook EJB version\itso\was4ad\data\Acco
D:\Program Files\Rational\Rose\Java\ITSO WAS AD Redbook EJB version\itso\was4ad\data\Acco
D:\Program Files\Rational\Rose\Java\ITSO WAS AD Redbook EJB version\itso\was4ad\data\Cust
D:\Program Files\Rational\Rose\Java\ITSO WAS AD Redbook EJB version\itso\was4ad\data\Data
```

Figure 6-15 Rose reverse engineer Java dialog

XMI toolkit

The XMI toolkit is a component of VisualAge that you can use to maintain consistency between your model and your code. If you want to use the XMI toolkit you must first make sure that you selected it as an option when you installed VisualAge for Java.

The XMI toolkit supports the following operations:

- Perform an XMI conversion between Rose and Java
- Show the UML/Java mapping created during an XMI conversion between Rose and Java
- Show the differences between two versions of UML XMI
- Show the differences between two versions of Java XMI

The XMI toolkit provides a GUI interface as well as command-line tools for performing these operations.

You can start the XMI toolkit GUI from VisualAge for Java by selecting *Workspace -> Tools -> XMI Toolkit* from the main workspace menu. If the *XMI Toolkit* menu option is not visible, most likely you did not choose to install the component when you installed VisualAge for Java.

The XMI toolkit window is shown in Figure 6-16.

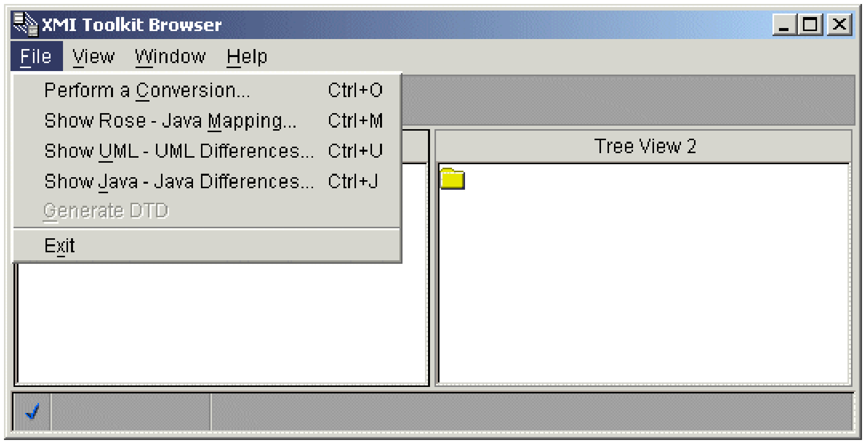

XMI Toolkit Browser
File
View
Window
Help
Perform a Conversion... Ctrl+O
Show Rose - Java Mapping... Ctrl+M
Show UML - UML Differences... Ctrl+U
Show Java - Java Differences... Ctrl+J
Generate DTD
Exit
Tree View 2

Figure 6-16 VisualAge for Java XMI toolkit window

We prefer and recommend the use of the Rose bridge to manage updates in your code and model, mainly because of the superior integration—information can be exchanged between the two tools on the fly, as opposed to having to save work to and from files.

The XMI toolkit can be used with other modeling tools, not just with Rose. If you want to learn more about the XMI toolkit, we suggest you consult the documentation available from the toolkit GUI's *Help* menu.

Plain Java files

The final alternative for integration between Rose and VisualAge is to use Java files exchanged via the file system. Java source code is generated into files by Rose, and imported into VisualAge using the *File -> Import* menu option. When code changes are complete, the code is re-exported from VisualAge for Java into the file system and reverse-engineered back into Rose to update the model.

This option is the least seamless and involves the most effort on the part of the developer—we recommend its use only as a last resort.

Designing EJBs with Rational Rose

Rational Rose provides tools to help us work with J2EE components such as servlets and EJBs. In this section we describe how we can use these tools to create a new EJB for the PiggyBank application.

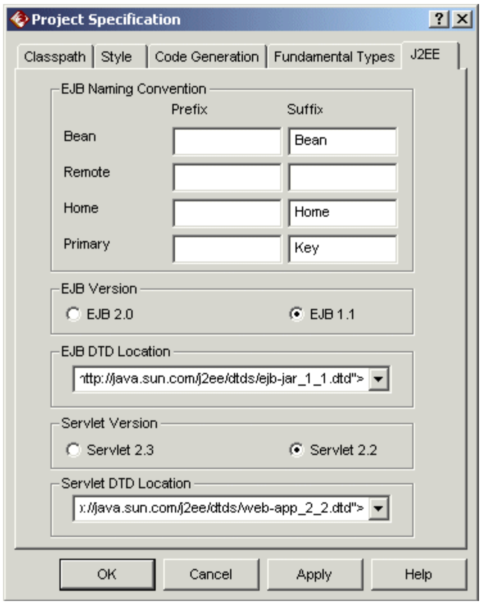
Defining the criteria that we use to define which of our business objects should be implemented as EJB components is beyond the scope of this book—we refer you to the many publications that discuss this subject.

Creating an EJB with Rose

This example describes how we can use Rose to create an entity EJB that enables us to represent and manage accounts in the PiggyBank application.

The version of Rose we are working with provides support for different levels of the J2EE specification. Before we start to create our EJB we must check the settings in Rose to make sure we generate code that is compatible with WebSphere Version 4.0.

We open our Java project specification in Rose, using the menu option *Tools -> Java / J2EE -> Project Specification*. This displays the project specification dialog. We click on the *J2EE* tab to display the dialog shown in Figure 6-17.



EJB Naming Convention

	Prefix	Suffix
Bean		Bean
Remote		
Home		Home
Primary		Key

EJB Version: EJB 2.0 [✓ EJB 1.1]

EJB DTD Location: ‹ttp://java.sun.com/j2ee/dtds/ejb-jar_1_1.dtd">

Servlet Version: Servlet 2.3 [✓ Servlet 2.2]

Servlet DTD Location: ›://java.sun.com/j2ee/dtds/web-app_2_2.dtd">

Figure 6-17 Rose Java project specification J2EE tab

We make the following changes and click *OK*:

- Change the EJB naming conventions as illustrated in the figure—this step is optional but convenient because our names now match the conventions used by VisualAge for Java.
- Change the EJB specification version to 1.1—the version supported by WebSphere Version 4.0.
- Change the servlet specification version to 2.2—this is also the version supported by WebSphere Version 4.0.

Creating a package

Next we create a package in our logical view in which to place the classes that make up our EJB. We name the package `itso.was4ad.ejb.account`. We create the hierarchy of packages in the Rose logical view using the *New -> Package* option from the logical view's context menu. We create each package in turn until we get the structure shown in Figure 6-18.

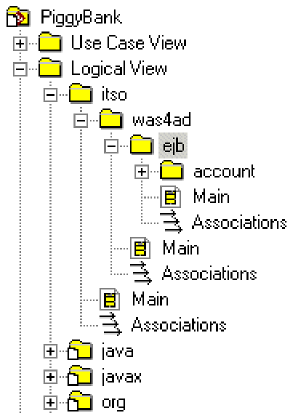


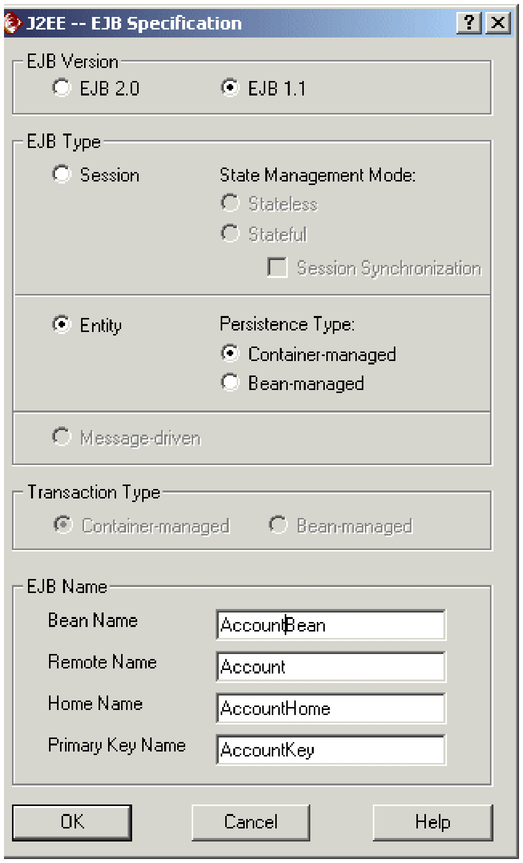
Figure 6-18 Logical view package structure

Create a class diagram

The next thing we do is create a new class diagram for the account EJB. We right-click in the browser window on the `itso.was4ad.ejb.account` package and select *New -> Class Diagram*. We call the diagram *Account EJB*, and double-click on it to open the empty diagram canvas.

Create the EJB

Next we create the account EJB. We select the *Tools -> Java / J2EE -> New EJB* menu option. This displays the EJB specification dialog (Figure 6-19), where we select the options to create a container managed entity bean. We can also display this dialog by selecting *Java / J2EE -> New EJB* from our class diagram's context menu, available by right-clicking the diagram's icon in the browser pane or in the background of the diagram itself.



J2EE -- EJB Specification

EJB Version
EJB 2.0 [✓ EJB 1.1]

EJB Type
Session — State Management Mode: Stateless / Stateful / Session Synchronization
[✓ Entity] — Persistence Type: [✓ Container-managed] / Bean-managed
Message-driven

Transaction Type
[✓ Container-managed] Bean-managed

EJB Name

Bean Name	AccountBean
Remote Name	Account
Home Name	AccountHome
Primary Key Name	AccountKey

Figure 6-19 Creating the account EJB in Rose

As we enter the bean name `Account` into the *Bean Name* field in the dialog Rose automatically fills in the other fields with names according to the convention we specified earlier. We click *OK* to create the EJB. Rose adds the components to our class diagram (Figure 6-20).

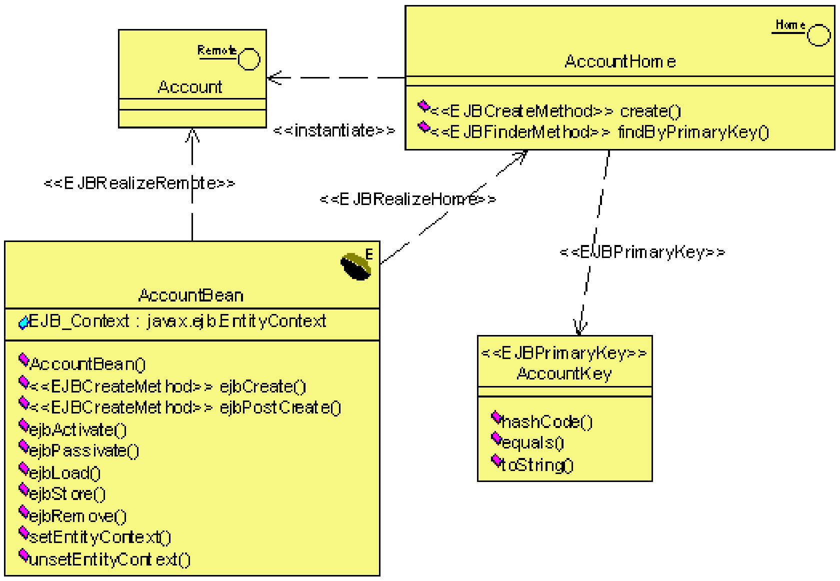
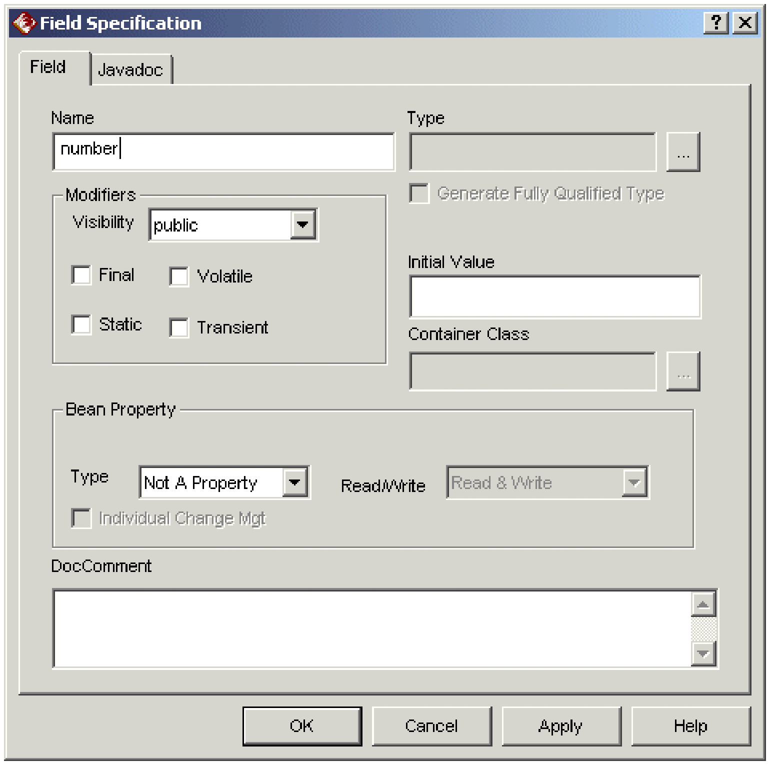


Figure 6-20 Customer EJB classes created by Rose

Adding CMP fields to the EJB

The next task is to add CMP fields to the EJB. We select the `AccountBean` class in the class diagram, right-click and select *Java / J2EE -> New EJB Method -> CMP Field*. This displays the field specification dialog shown in Figure 6-21.

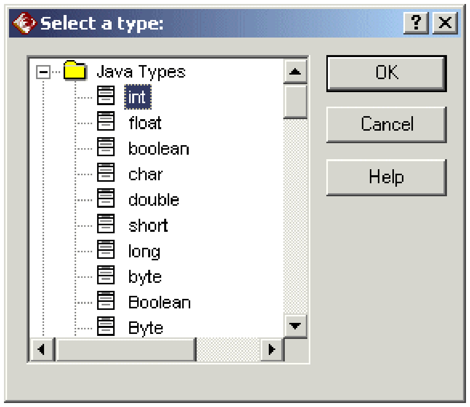
We enter the name of our CMP field, `number`, in the *Name* field, and click the button next to the greyed-out *Type* field to define the field type.



Name: number
Type:
Generate Fully Qualified Type
Modifiers
Visibility: public
Final
Volatile
Static
Transient
Initial Value:
Container Class:
Bean Property
Type: Not A Property
Read/Write: Read & Write
Individual Change Mgt
DocComment:

Figure 6-21 Field specification dialog

We expand the *Java Types* and select `int` from the list (Figure 6-22).



Java Types
int
float
boolean
char
double
short
long
byte
Boolean
Byte

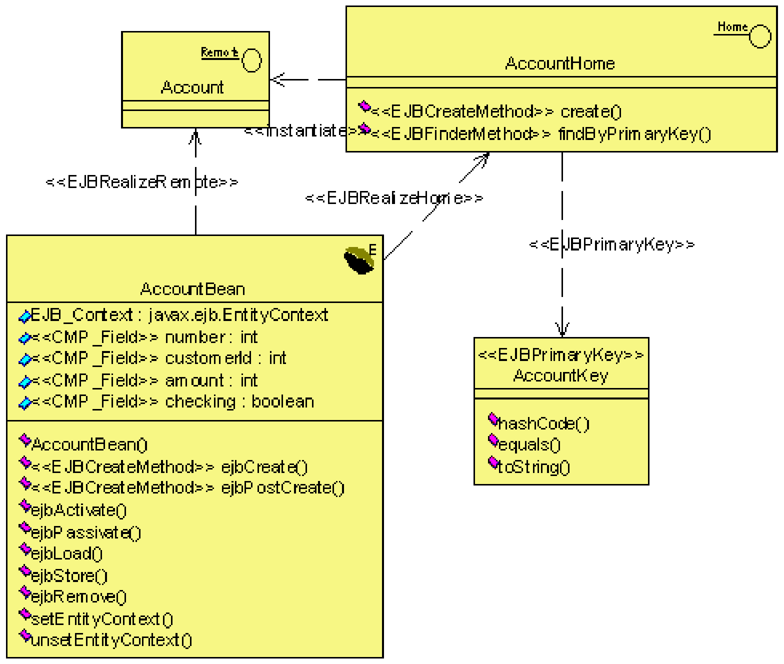
Figure 6-22 Selecting the CMP EJB field type

We repeat this procedure for the other CMP fields in our EJB. All four fields are listed in Table 6-1.

Table 6-1 CMP fields in the account EJB

Field name	Type	Description
`number`	`int`	Account number—primary key
`customerId`	`int`	Customer number of the customer that owns the account
`amount`	`int`	Current balance in the account
`checking`	`boolean`	`true` if a checking account, `false` if savings

The class diagram for our EJB now appears as shown in Figure 6-23.



Remote
Account
Home
AccountHome
<<EJBCreateMethod>> create()
<<EJBFinderMethod>> findByPrimaryKey()
<<instantiate>>
<<EJBRealizeRemote>>
<<EJBRealizeHome>>
<<EJBPrimaryKey>>
E
AccountBean
EJB_Context : javax.ejb.EntityContext
<<CMP_Field>> number : int
<<CMP_Field>> customerId : int
<<CMP_Field>> amount : int
<<CMP_Field>> checking : boolean
AccountBean()
<<EJBCreateMethod>> ejbCreate()
<<EJBCreateMethod>> ejbPostCreate()
ejbActivate()
ejbPassivate()
ejbLoad()
ejbStore()
ejbRemove()
setEntityContext()
unsetEntityContext()
<<EJBPrimaryKey>>
AccountKey
hashCode()
equals()
toString()

Figure 6-23 The updated EJB class diagram

Adding fields to the primary key class

The primary key for the EJB is the account number, which is an `int`. We must add a corresponding attribute to the primary key class `AccountKey`.

We add the attribute by selecting the primary key class in the logical view and *New -> Attribute* from the context menu (Figure 6-24).

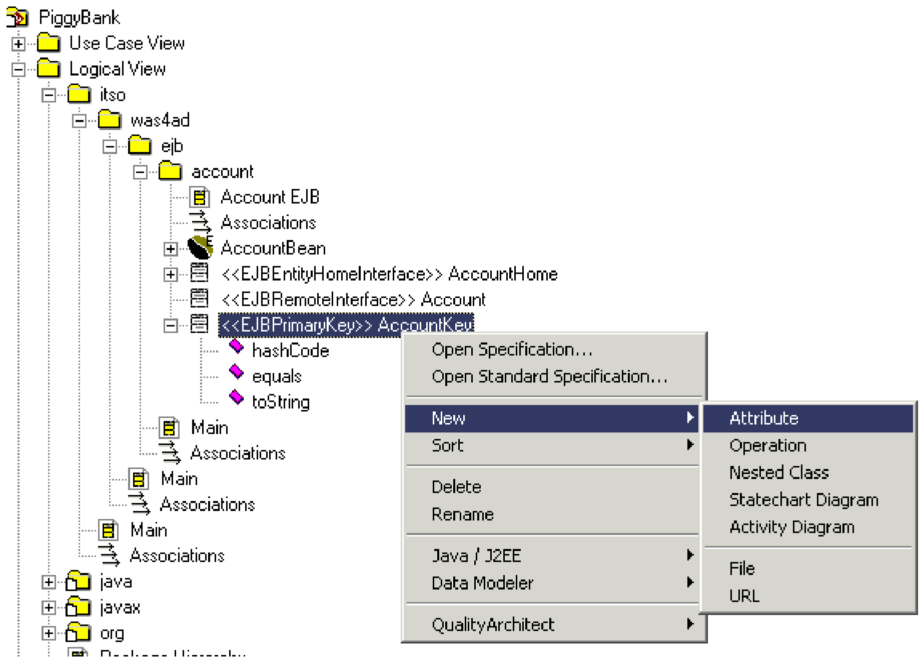


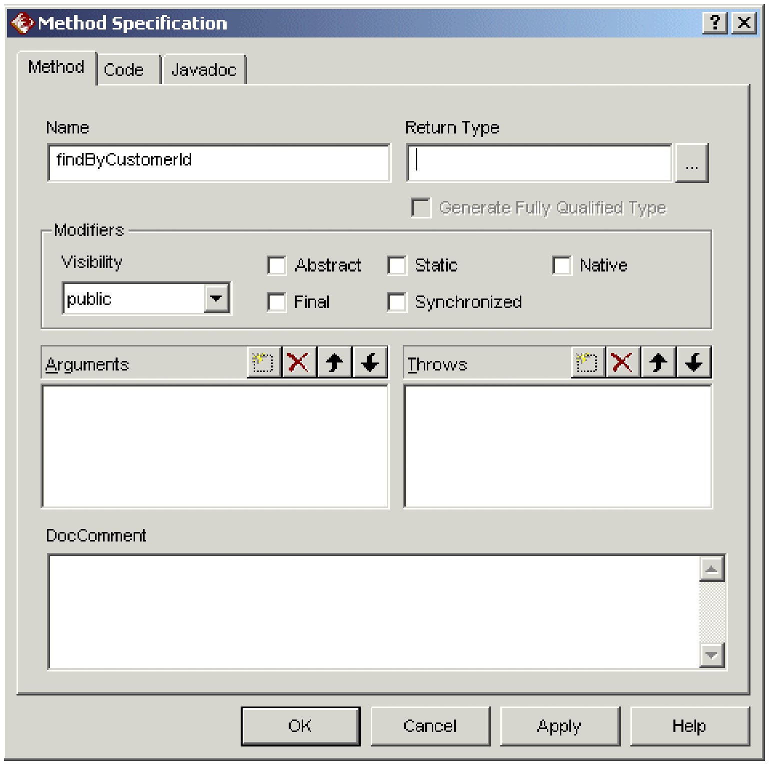
Figure 6-24 Adding an attribute to the primary key class

We name the attribute `number`, and double-click to open the *Field Specification* dialog. We change the field type to `int` and click *OK*.

Adding a finder method to the EJB

We want to add a custom finder to our EJB—we have to be able to find all accounts belonging to a particular customer—so we create a custom finder named `findByCustomerId`. We select the bean class and *Java / J2EE -> New EJB Method -> Finder Method* from the context menu.

In the dialog shown in Figure 6-25 we enter the finder name into the *Name* field, and click the button to the right of the *Return Type* field to select the return type.


Method Specification

Method | Code | Javadoc

Name: findByCustomerId

Return Type:

Generate Fully Qualified Type

Modifiers

Visibility: public

Abstract | Static | Native

Final | Synchronized

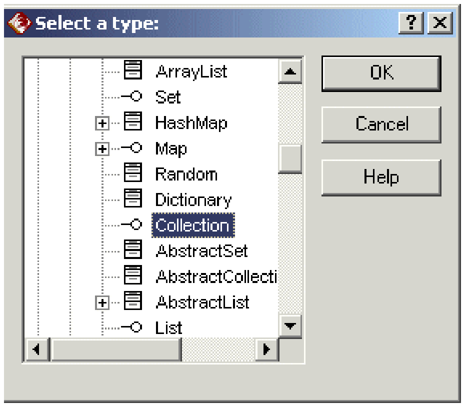
Arguments

Throws

DocComment

Figure 6-25 EJB method specification dialog

We select the `java.util.Collection` interface (Figure 6-26).


Select a type:

ArrayList
Set
HashMap
Map
Random
Dictionary
Collection
AbstractSet
AbstractCollecti
AbstractList
List

Figure 6-26 Selecting the finder return type

Next we click the button to the right of the *Arguments* label to add a new parameter to the finder. The customer ID is an `int`, so we enter that information in the next dialog (Figure 6-27).

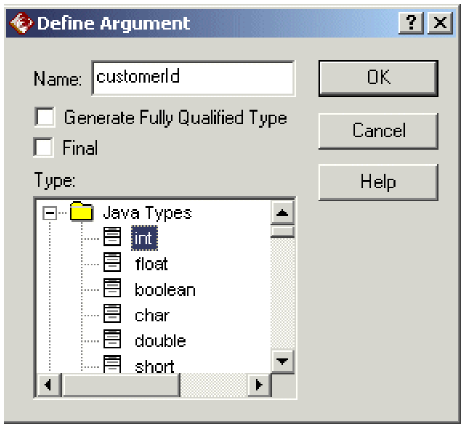


Figure 6-27   Specifying a finder parameter

Back in the original *Method Specification* dialog we click *OK* to add the finder to the EJB. The EJB home interface in the class diagram is updated (Figure 6-28).

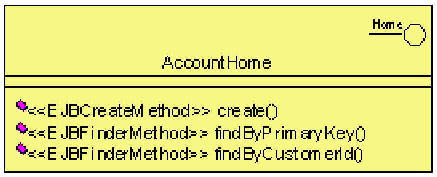

Home
AccountHome
<<EJBCreateMethod>> create()
<<EJBFinderMethod>> findByPrimaryKey()
<<EJBFinderMethod>> findByCustomerId()

Figure 6-28   EJB home interface with finder methods

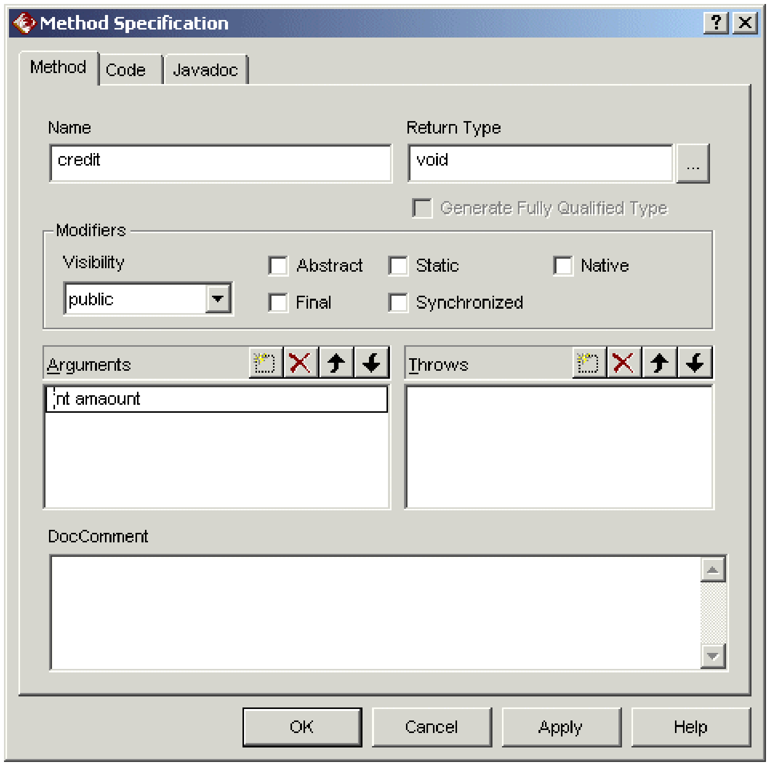
Adding business methods to the EJB

We want to add three business methods to the account EJB. The methods are listed in Table 6-2.

Table 6-2   Account EJB business methods

Method name	**Description**
`credit`	Credits money to the account
`debit`	Debits money from the account
`isOwnedBy`	Tests if an account is owned by a given customer

We add the methods by selecting the bean class in the class diagram and *Java / J2EE -> New EJB Method -> Business* from the context menu. We complete the *Field Specification* dialog shown in Figure 6-29 for each method in turn.



Name: credit
Return Type: void
Visibility: public
Arguments: int amaount

Figure 6-29 debit EJB method specification dialog

Having completed these updates, the class diagram now appears as shown in Figure 6-30.

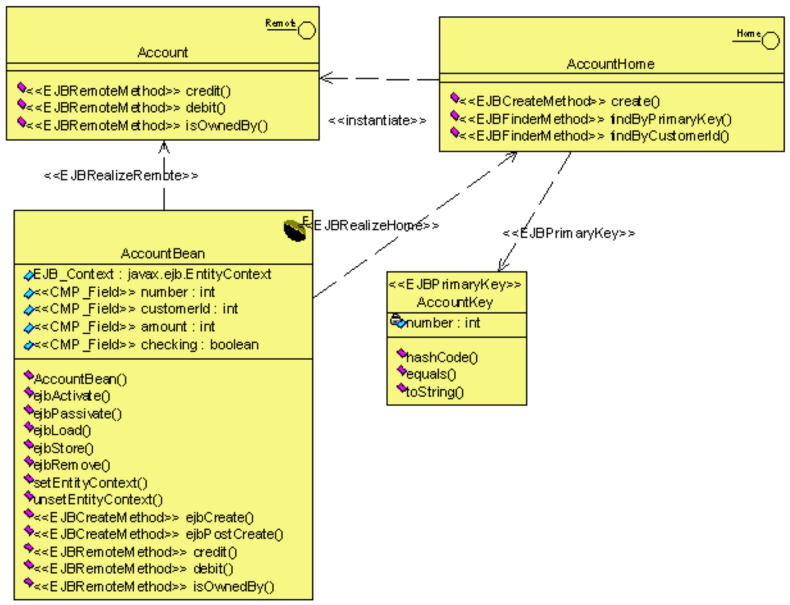

Remote
Account
<<EJBRemoteMethod>> credit()
<<EJBRemoteMethod>> debit()
<<EJBRemoteMethod>> isOwnedBy()
<<instantiate>>
Home
AccountHome
<<EJBCreateMethod>> create()
<<EJBFinderMethod>> findByPrimaryKey()
<<EJBFinderMethod>> findByCustomerId()
<<EJBRealizeRemote>>
<<EJBRealizeHome>>
<<EJBPrimaryKey>>
AccountBean
EJB_Context : javax.ejb.EntityContext
<<CMP_Field>> number : int
<<CMP_Field>> customerId : int
<<CMP_Field>> amount : int
<<CMP_Field>> checking : boolean
AccountBean()
ejbActivate()
ejbPassivate()
ejbLoad()
ejbStore()
ejbRemove()
setEntityContext()
unsetEntityContext()
<<EJBCreateMethod>> ejbCreate()
<<EJBCreateMethod>> ejbPostCreate()
<<EJBRemoteMethod>> credit()
<<EJBRemoteMethod>> debit()
<<EJBRemoteMethod>> isOwnedBy()
<<EJBPrimaryKey>>
AccountKey
number : int
hashCode()
equals()
toString()

Figure 6-30   Final class diagram for the account EJB

Generating EJB code

We can generate code for our EJB in exactly the same way we would generate Java code for any class in Rose, as we described earlier in "Code generation" on page 126. The steps in this example are fundamentally the same—we illustrate generating code to the file system.

First of all we go to our class diagram and select all four of the classes that make up our EJB—the home, remote, bean and primary key classes. We then right-click on our selection and select *Java / J2EE -> Generate Code* from the context menu.

The dialog shown in Figure 6-31 appears. We must tell Rose where to store the generated Java source files for the `itso` package hierarchy. We want to save our source in the directory `D:\ITSO4AD\dev\src\ejb`. We click on *Edit* to alter the class path to include this new directory.

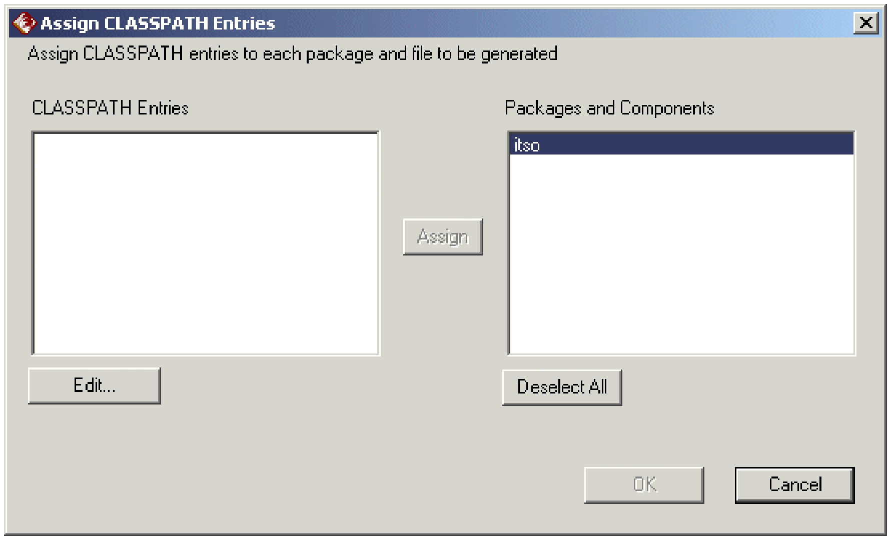
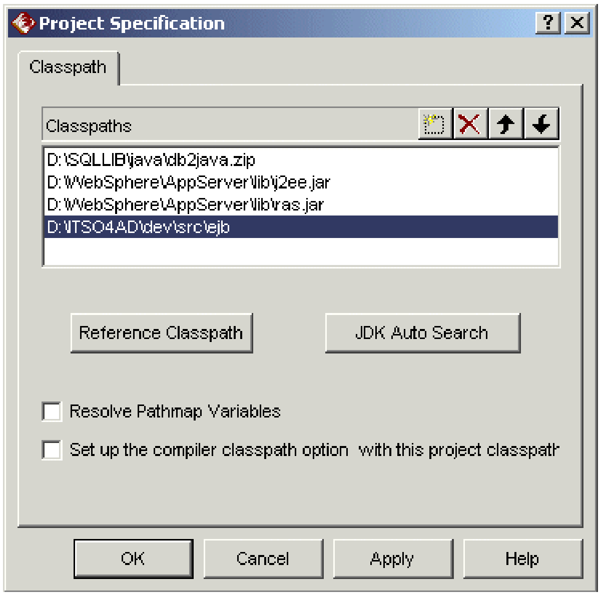


Figure 6-31 Assigning the class path

We add our new directory to the class path and click *OK* (Figure 6-32).

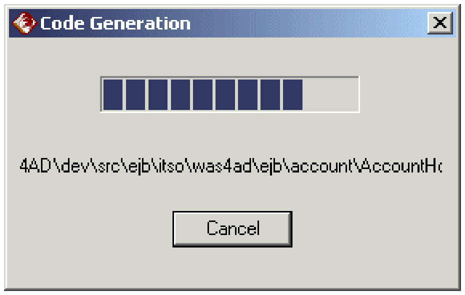


```
D:\SQLLIB\java\db2java.zip
D:\WebSphere\AppServer\lib\j2ee.jar
D:\WebSphere\AppServer\lib\ras.jar
D:\ITSO4AD\dev\src\ejb
```

- [ ] Resolve Pathmap Variables
- [ ] Set up the compiler classpath option with this project classpath

Figure 6-32 Adding the source directory to the class path

We then select the new class path entry in the *Assign Classpath* dialog, and click *Assign* to assign the package to that class path. We are now able to click the *OK* button to generate the code. Rose displays the dialog shown in Figure 6-33 while code generation is taking place.



4AD\dev\src\ejb\itso\was4ad\ejb\account\AccountH

Figure 6-33 Code generation dialog

When the dialog closes, code generation in complete. We can now look in the file system to examine the generated code. The code has been generated in the structure shown in Figure 6-34.

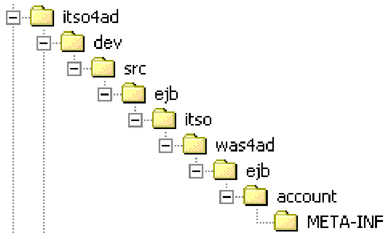


Figure 6-34 Directory structure of the generated code

Note that Rose has generated a `META-INF` directory, which contains a deployment descriptor for our new EJB. This deployment descriptor is shown in Figure 6-35.

```
<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE ejb-jar PUBLIC "-//Sun Microsystems, Inc.//DTD Enterprise
JavaBeans 1.1//EN" "http://java.sun.com/j2ee/dtds/ejb-jar_1_1.dtd">

<ejb-jar>
 <enterprise-beans>
  <entity>
   <ejb-name>AccountBean</ejb-name>
   <home>itso.was4ad.ejb.account.AccountHome</home>
   <remote>itso.was4ad.ejb.account.Account</remote>
   <ejb-class>itso.was4ad.ejb.account.AccountBean</ejb-class>
   <persistence-type>Container</persistence-type>
   <prim-key-class>itso.was4ad.ejb.account.AccountKey</prim-key-class>
   <reentrant>False</reentrant>
   <cmp-field>
    <field-name>number</field-name>
   </cmp-field>
   <cmp-field>
    <field-name>customerId</field-name>
   </cmp-field>
   <cmp-field>
    <field-name>amount</field-name>
   </cmp-field>
   <cmp-field>
    <field-name>checking</field-name>
   </cmp-field>
  </entity>
 </enterprise-beans>
</ejb-jar>
```

Figure 6-35 Account EJB deployment descriptor generated by Rose

Importing an EJB from Rose into VisualAge for Java

Instead of generating the code into a directory, you can import an EJB designed with Rose into VisualAge for Java.

Define an EJB group in VisualAge for Java, then select *EJB -> Add -> Import from Rose or XMI*.

In the Import SmartGuide select the Rose model file (.mdl), the EJB group, and enter the name of the package for the code. Skip the next page (virtual paths) and click *Finish*. The EJBs are added to the EJB group and can be tailored in VisualAge for Java.

7

Designing with frameworks

In this chapter we introduce the concepts of frameworks and explain what frameworks are good for. We provide a high-level vision of two major frameworks for Web application development and clarify how and when they should be chosen and applied:

- Jakarta Struts
- WebSphere Business Components Composer (WSBCC)

In Chapter 12, “Development with frameworks” on page 283 we provide the details on how to develop a Web application with these frameworks.

Introduction

As soon as an application development project starts up, one of the first concerns is the architecture to adopt. This includes the hardware as well as the software. In the software architecture domain, one of the first decisions to make is to define the scope of the application, which is made up by two elements:

- What the system should be at the end, what it should do.
- What the system should be at the beginning.

The first point can be left quite vague and is often determined late in the project, either according to time availability or because of an incremental development strategy, which mainly focuses on the next iteration dates, leaving the later schedule in the dark.

The second point is far more critical and it is surprisingly often decided with so little consideration, according to enterprise-wide standards, contractual agreements or just because of a hasty project start. And this is a shame, because choosing the right basis for an application is just like for a house: it will help its development and decide its robustness.

It is widely considered that starting any development from an existing basis is more productive than starting from scratch. This is the main idea behind all the re-use hype all around. And this is true and false because everyone tries to sell their basic product, and in an apparently random proportion the ongoing projects realize or not they entered a nightmare they cannot get out of. There is no random, no fate. The situation is purely rational although complicated—no surprise. To understand the ins and outs of it, we must introduce some shades in the utopian vision the marketing world tries to spread. Because the wonderful productivity promises can be achieved only by being conscious of what the stakes are.

Starting with a framework

Projects starting from frameworks work. This is a general principle with plenty of successful examples. And it could become a global reality if the frameworks were wisely chosen and used, and if some basic recommendations are followed.

In this section, we introduce some concepts and list some major nontechnical dangers when considering a framework. Avoiding these brings the necessary freedom that allows focus on the technical considerations.

What is a framework?

This term is very often debased. This adds to the confusion around frameworks.

A wise definition could be: *a framework is a software piece that an application development can start with; it is a basis*.

In the software architecture world, a framework is considered as *existing pieces of code in which your code fits*. It follows the well-known Hollywood principle, "don't call us we call you." The code you write is not called directly from the user intervention into the system. The runtime flow goes through the framework code then finally ends in your code.

In such a vision, a framework can be opposed to a component library, where the code you write calls existing pieces of code to perform a task. The UML sequence diagram in Figure 7-1, where the time goes by downwards, illustrates these considerations.

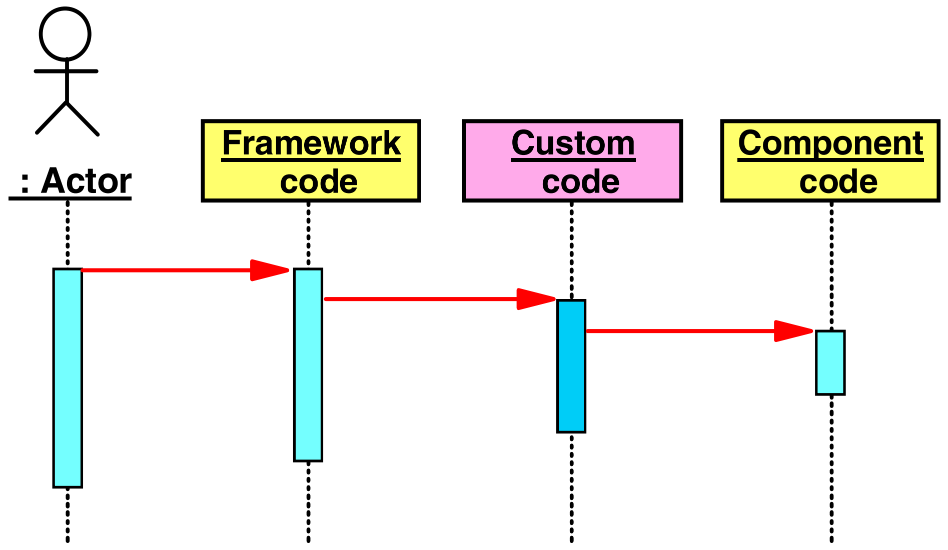


Figure 7-1 Framework and components sequence diagram

Sometimes the term framework is also used to refer to an object model that can be extended (mainly by inheritance) to suit the custom application needs. The backbone code then considers inherited objects just like framework parent objects through polymorphism.

Furthermore, all definitions can be used together. The “WebSphere Business Components Composer” on page 165 is a perfect example of this and the Java core classes are full of examples:

- Servlet API
- `addElement(Object element)` method in the `java.util.Vector` class
- `java.util.Calendar` class

Additional information about the Java core classes can be found at:

`http://java.sun.com/j2se/1.3/docs/api/index.html`

Frameworks drawbacks

Any medal has its reverse. Frameworks bring some advantages and introduce some limitations:

- Frameworks are not versatile

  They are good for what they have been designed for, and nothing else. Different things have to be done to work around the framework. This is what makes the framework choice critical. So many projects end up with an application that has replaced all the framework parts by custom code. In that case, the use of a framework can result in loss of time.

- Frameworks are not flexible

  If you want some minor change to it, it is often impossible to get it from the provider as any change would break other existing codes. This is particularly true with a widely-adopted framework where backward compatibility is critical.

- Frameworks impose a way of thinking

  Custom code has to stick to the framework jigsaw. Different ideas just do not fit. If the framework is well designed, this can be a good thing because it prevents from bad practices.

- Frameworks are specific

  If the team is new to it, the learning curve can be long. This needs to be qualified: the concepts behind are usually standard and the learning process is therefore limited to new terms.

Framework adoption

One of the major problems with frameworks, besides their own technical characteristics, is their adoption in the team. People hate frameworks.

After the first technical presentations or classes, the team usually rejects the potential or chosen framework as is. It is considered as “bad” and “useless.” Sometimes the comments are worse or even abusive. People do not want to use it. They have their own ideas “to do it better.” Wily programmers talk to their managers in terms of cost: “we can do it ourselves and it will be cheaper.” This is seldom true. For large repetitive projects, it is most of the time false.

So, what can we do with that? Actually, the solution falls within the competence of the project manager and his skills in the human relationships area. Forcing the team to adopt the framework just will not make it. This negotiation is a matter of honesty. Using a framework is often a win-win situation that the technical persons do not see. They know how things have to be done and they often feel the framework as a denial of their capacity to do what the framework does.

Actually, this is the contrary: they have to be aware that they are the best people for the job, because they already know the environment they will have to integrate with. They can add their own value to the framework in the fastest way. And if they are conscious of their value, which is usually not the case, they can leave their fear of doing new things and learn from a new experience.

Integration with the tools

There is a consideration common to frameworks and methodologies: they must fit the tools and vice versa.

There is more than coincidence if some tools fit and some do not. Some frameworks have been designed with or for some tools. Sometimes both.

For instance, WSBCC has been designed with and for VisualAge for Java. On the contrary, Struts has been developed totally independently and is more difficult to integrate with VisualAge for Java. This also shows the mismatch between two different development worlds: commercial and open-source. Actually, this can be solved as we describe in “Generating the WTE webapp file from a web.xml file” on page 287.

Anyway, any mismatch has to be considered *before* choosing the framework-tool combination.

Jakarta Struts

Struts is part of the Jakarta project, sponsored by the Apache Software Foundation.

Note: This section and its subsections contain documentation taken from the official Jakarta project Struts home page and from the official Struts user guide at:

```
http://jakarta.apache.org/struts
http://jakarta.apache.org/struts/userGuide/introduction.html
```

It also contains some quotes from Kyle Brown's articles on Struts in the VisualAge Developer Domain (VADD):

```
http://www.ibm.com/vadd
```

- Apache Struts and VisualAge for Java, Part 1: Building Web-based Applications using Apache Struts
- Apache Struts and VisualAge for Java, Part 2: Using Struts in VisualAge for Java 3.5.2 and 3.5.3

The goal of the Struts project is to provide an open source framework useful in building Web applications with Java Servlet and Java ServerPages (JSP) technology. Struts encourages application architectures based on the model-view-controller (MVC) design paradigm, colloquially known as Model II.

Struts includes the following primary areas of functionality:

- A controller servlet that dispatches requests to appropriate *Action* classes provided by the application developer.
- JSP custom tag libraries, and associated support in the controller servlet, that assists developers in creating interactive form-based applications.
- Utility classes to support XML parsing, automatic population of JavaBeans properties based on the Java reflection APIs, and internationalization of prompts and messages.

When to use Struts

Actually, most common Web applications can find some benefit in using Struts. As we have seen earlier, the MVC pattern allows to design the model (business logic) of the application in a traditional fashion. Adding a Web "controller + view" transforms this model into a Web application. Struts helps building the "controller + view" part, thus focusing on the presentation logic.

While J2EE APIs make it possible to develop Web-based applications that implement the MVC pattern, there are a number of common problems that must be solved in every servlet project (Ibid. Kyle Brown):

- Mapping HTTP parameters to JavaBeans—One of the most common tasks facing servlet programmers is to map a set of HTTP parameters (from the command line or from the POST of an HTML form) to a JavaBean for manipulation. This can be done using the `<jsp:useBean>` and `<jsp:setProperty>` tags, but this arrangement is cumbersome because it requires POSTing to a JSP, something that is not encouraged in a Model-II MVC architecture.
- Validation—There is no standard way in servlet/JSP programming to validate that an HTML form is filled in correctly. This leaves every servlet programmer to develop his own validation procedures, or not, as is too often the case.
- Error display—There is no standard way to display error messages in a JSP page or generate error messages in a servlet.
- Message internationalization—Even when developers strive to keep as much of the HTML as possible in JSPs, there are often hidden obstacles to internationalization spread throughout servlet and model code in the form of short error or informational messages. While it is possible to introduce internationalization with the use of Java resource managers, this is rarely done due to the complexity of adding these references.
- Hard coded JSP URIs—One of the more insidious problems in a servlet architecture is that the URIs of the JSP pages are usually coded directly into the code of the calling servlet in the form of a static string reference used in the `ServletContext.getRequestDispatcher` method. This means that it is impossible to reorganize the JSPs in a Web site, or even change their names, without updating Java code in the servlets.

The problem is that programmers are too often faced with "reinventing the wheel" each time they begin building a new Web-based application. Having a framework to do this work for them would make them more productive and let them focus more on the essence of the business problems they are trying to solve, rather than on the accidents of programming caused by the limitations of the technology (*No Silver Bullet: Essence and Accident in Software Engineering*. Fred Brooks. IEEE Computer, April 1987).

Simply put, Struts is an open-source framework for solving the kind of problems described above. Information on Struts, a set of installable JAR files, and the full Struts source code is available at the Struts framework Web site. Struts has been designed from the ground up to be easy to use, modular (so that you can choose to use one part of Struts without having to use all the others), and efficient. It has also been designed so that tool builders can easily write their tools to generate code that sits on top of the Struts framework (Ibid. Kyle Brown).

Servlet controller

True to the model-view-controller design pattern, Struts applications have three major components: a servlet controller, Java ServerPages (the "view"), and the application's business logic (the "model").

The controller bundles and routes HTTP requests from the client (typically a user running a Web browser) to framework objects and corresponding extended objects, deciding what business logic function is to be performed, then delegates responsibility for producing the next phase of the user interface to an appropriate view component like a JSP.

In Struts, the primary component of the controller is a servlet of class `org.apache.struts.action.ActionServlet`. When initialized, the controller parses a configuration resource file. The configuration resource defines, among other things, the action mappings for the application. The controller uses these mappings to turn HTTP requests into application actions.

At a minimum, a mapping must specify:

- ► A request path
- ► The object type to act upon the request

Each mapping defines a path that is matched against the request URI of the incoming request, and the fully qualified class name of an action class (that is, a Java class extending the `Action` class) which is responsible for performing the desired business logic, and then dispatching control to the appropriate View component to create the response.

The Struts `ActionServlet` basically plays the same role as the more simple `itso.was4ad.webapp.controller.ControllerServlet` we defined in the first version of the PiggyBank application.

Action objects

The action object can handle the request and respond to the client (usually a Web browser), or indicate that control should be forwarded to another action. For example, if a login succeeds, a `loginAction` object may want to forward control to a `mainMenu` action.

Action objects are linked to the application controller, and so have access to that servlet methods. When forwarding control, an object can indirectly forward one or more shared objects, including JavaBeans, by placing them in one of the standard collections shared by Java servlets.

An action object can for instance create a shopping cart bean, add an item to the cart, place the bean in the session collection, and then forward control to another action, which may use a JSP to display the contents of the user's cart. Because each client has their own session, they will each also have their own shopping cart. In a Struts application, most of the business logic can be represented using JavaBeans.

Following to the Struts guidelines, the PiggyBank application would contain several actions:

- Login
- Display accounts
- Transfer

This is represented in a class diagram (Figure 7-2).

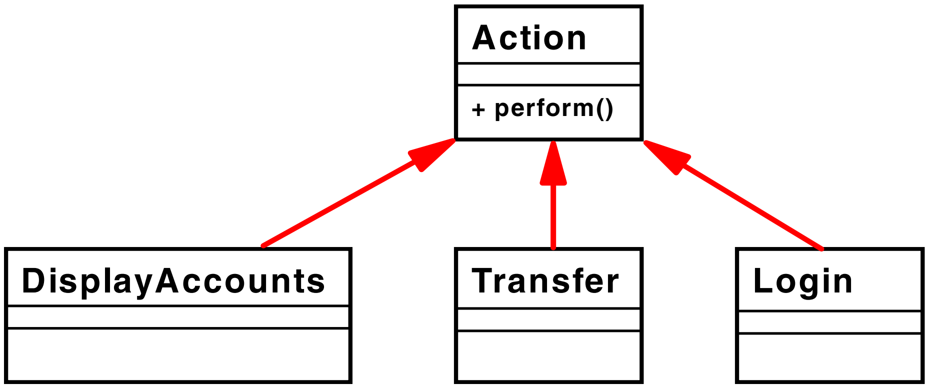


Figure 7-2 PiggyBank actions in Struts

The action set is very similar to the command set we defined in the first version of the PiggyBank application. A corollary of this observation is that the action classes to define in Struts correspond to the application design use cases in the same way.

Form beans

JavaBeans can also be used to manage input forms. A key problem in designing Web applications is retaining and validating what a user has entered between requests. With Struts, you can easily store the data for a input form in a form bean. The bean is saved in one of the standard, shared context collections, so that it can be used by other objects. The action object receives it as input to perform its task.

The form bean can be used:

- To collect data from the user
- To validate what the user entered
- By the JSP to re-populate the form fields.

In the case of validation errors, Struts has a shared mechanism for raising and displaying error messages. It automatically invokes the `ActionForm.validate` method whenever the JSP page containing the form corresponding to this `ActionForm` submits the form. Any type of validation can be performed in this method. The only requirement is that it returns a set of `ActionError` objects in the return value. Each `ActionError` corresponds to a single validation failure, which maps to a specific error message. These error messages are held in a properties file that the Struts application refers to.

A Struts form bean is defined in the configuration resource and linked to an action mapping using a common property name. When a request calls for an action that uses a form bean, the controller servlet either retrieves or creates the form bean, and passes it to the action object (Figure 7-3).

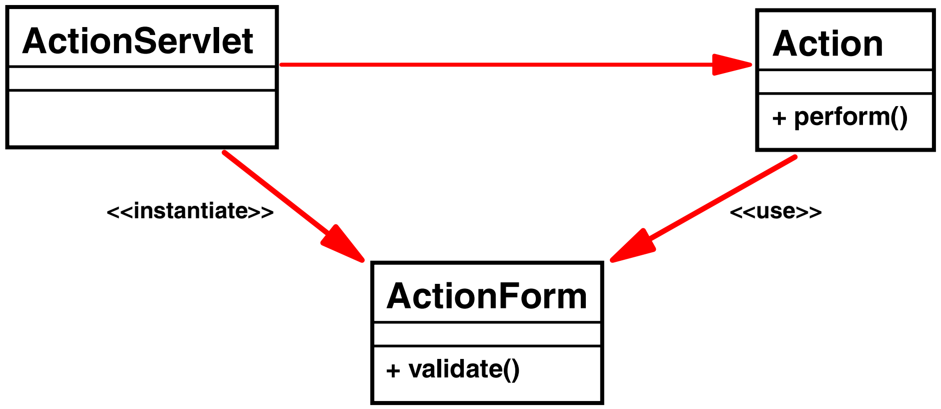


Figure 7-3 ActionForm handling

The action object can then check the contents of the form bean before its input form is displayed, and also queue messages to be handled by the form. When ready, the action object can return control with a forwarding to its input form, usually a JSP. The controller can then respond to the HTTP request and direct the client to the JSP. Figure 7-4 summarizes these operations.

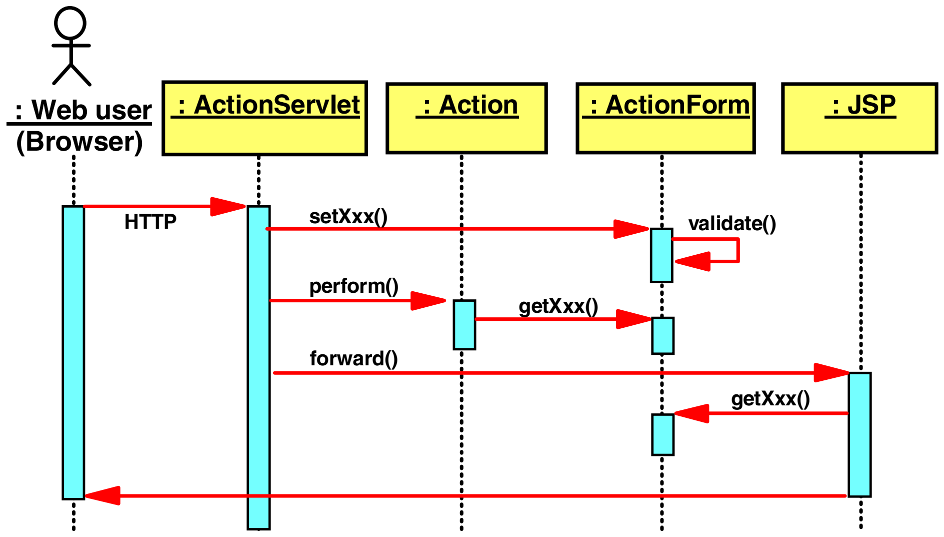


Figure 7-4 Struts request sequences

Custom tags

There are four JSP tag libraries that Struts includes:

1. The HTML tag library, which includes tags for describing dynamic pages, especially forms.
2. The beans tag library, which provides additional tags for providing improved access to Java beans and additional support for internationalization.
3. The logic tag library, which provides tags that support conditional execution and looping.
4. The template tag library for producing and using common JSP templates in multiple pages.

Using these custom tags, the Struts framework can automatically populate fields from and into a form bean, raising two advantages:

- The only thing most JSPs need to know about the rest of the framework is the proper field names and where to submit the form. The associated form bean automatically receives the corresponding value.
- If a bean is present in the appropriate scope, for instance after an input validation routine, the form fields will be automatically initialized with the matching property values.

Therefore, an HTML input field declaration code as:

```
<input type="text" name="amount" value="<%= bean.getFirstName() %>">
```

can be replaced in a JSP by a more elegant and efficient Struts tag:

```
<html:text property="amount"/>
```

Internationalization

Components such as the messages set by the action object can be output using a single custom tag. Other application-specific tags can also be defined to hide implementation details from the JSPs.

The custom tags in the Struts framework are designed to use the internationalization features built into the Java platform. All the field labels and messages can be retrieved from a message resource, and Java can automatically provide the correct resource for a client's country and language. To provide messages for another language, simply add another resource file.

Internationalism aside, other benefits to this approach are consistent labeling between forms, and the ability to review all labels and messages from a central location.

Code dependencies

For the simplest applications, an action object can handle the business logic associated with a request. However, in most cases, an action object should pass the request to another object, usually a JavaBean. To allow reuse on other platforms, business logic JavaBeans should not refer to any Web application objects. The action object should translate needed details from the HTTP request and pass those along to the business-logic beans as regular Java variables.

In a database application, the business logic beans might connect to and query the database and return the result set back to the action servlet to be stored in a bean and then displayed by the JSP. Neither the action servlet nor the JSP have to know or care where the result set comes from.

Downsides

No framework is perfect. Struts cannot be all things to all people, so you lose some things when using Struts that you can do when programming directly to the servlet API. Possibly the biggest downside is that with Struts you have only one servlet (the `ActionServlet`) serving up all of the dynamic pages of your Web application. Having only a single servlet per application is certainly a problem when you want to use a tool like WebSphere Resource Analyzer, which gives number of hits and average response time for a particular servlet. It is harder to obtain useful performance and load information when you have only a single servlet per application. In order to obtain this kind of load data, you would have to instrument your code (Ibid. Kyle Brown).

Another potential mismatch is in trying to apply Struts to a portal style of application. While you can use Struts in this style of application (especially considering its template support) you will find more appropriate support in other Apache projects, such as the Apache JetSpeed and Turbine portal servers (Ibid. Kyle Brown).

Development

See "Jakarta Struts" on page 284 for further information on application development with Jakarta Struts.

WebSphere Business Components Composer

The IBM WebSphere Business Components Composer, also known as WSBCC or "The Composer," is defined as a component-based framework for developing enterprise e-business applications. The framework components promote highly productive application development by supporting code reuse and the use of parametrization techniques to define business operations and their related objects externally.

WSBCC enables solution developers to focus on business functions and to avoid being slowed by technological issues such as communication protocols and message formatting. This permits faster time to market, especially for enterprise-wide project where it is possible to capitalize on human resource investment. A chain of projects starting from no Web application and using WSBCC typically see surprising time results from the second project.

The main technologies used are HTML, Java and XML. The skills required to develop with WSBCC are mainly Junior JSP and Senior Java for the framework extensions.

The development model creates a clear separation of roles that allows project team members to focus on their specific tasks. In a typical usage, WSBCC relies less on high programming skills because it provides components that are easy to understand and use, from back-end connectors to user interface building blocks.

When to use WSBCC

The WSBCC framework is well suited for building Web-based financial services applications, such as bank branch systems, as well as building solutions for a wide variety of retail delivery channels. WSBCC has been specifically designed to support multichannel architectures and extend the reach of a financial institution's information system services to all of its delivery channels. New channels include:

- Internet banking, or e-banking
- Call centers
- Stand-alone kiosks
- Automated teller machines (ATMs)
- Mobile access terminals, such as wireless access protocol (WAP) capable cellular phones

In this book, we will focus on the e-banking development.

Financial institution services are most often supported by applications whose core logic and data reside on large-scale host systems. Financial service delivery must access transaction functions on these systems, for example, to transfer funds between accounts. This all means the use cases are most of the time already existing and have been implemented in a host system. WSBCC provides a very complete and extensible infrastructure to interface such host systems with the Web, or other channels. To this end, it includes components designed to handle all aspects of transaction processing:

- Managing the user interface
- Assisting navigation
- Gathering and validating operation data
- Building host messages
- Processing host responses
- Logging transaction information
- Accessing financial devices

While WSBCC performs the best in financial domain, it is a potential solution to any transaction processing requirements no matter what the industry. Similar technical architectures can be found everywhere and WSBCC is highly customizable. Actually, like in any well-designed framework, it is potentially possible to build any kind of application. In "WebSphere Business Components Composer" on page 303 we illustrate with an example what levels of benefits can be achieved with WSBCC depending on how much you use of it.

Deployment and maintenance

After the application is deployed, its maintenance and operating efforts are small.

Deployment of a WSBCC-based application does not require changes in existing business logic or transactions run in back-end systems. The framework has been designed to provide communication systems between existing back-end systems and the Web application located on a middle-tier server. An extreme case is where the Web application has no business knowledge and just acts as a presentation layer.

WSBCC uses the standard TCP/IP network computing architecture for client administration, code distribution and server management. The deployed elements are modular and can be operated with minimal system administrator training.

Portability relies on the Java “Write Once Run Anywhere” paradigm. If operational conditions require that the application be moved to another platform, it can be quickly performed. This is also true for typical OS-mismatch between development and production: there is no particular difficulty to develop the Web application in a Windows NT distributed environment and to deploy it on a Unix server.

Ease of adaptation to back-end changes is achieved by externalizing the interface formats and/or protocols in XML configuration files. These changes can be made manually in the files or can be assisted through the Development Workbench tool (which is not covered in this book).

Architecture

The architecture of a WSBCC Web application is based on a logical three-tier model and standard communication protocols (Figure 7-5):

1. Back-end enterprise server
2. Middle-tier application server
3. Browser

The enterprise server, or back-end server, contains the existing core business logic of the institution. Such a system and its messaging interface remain untouched as the Web application uses the WSBCC set of back-end system connector components and message formatters.

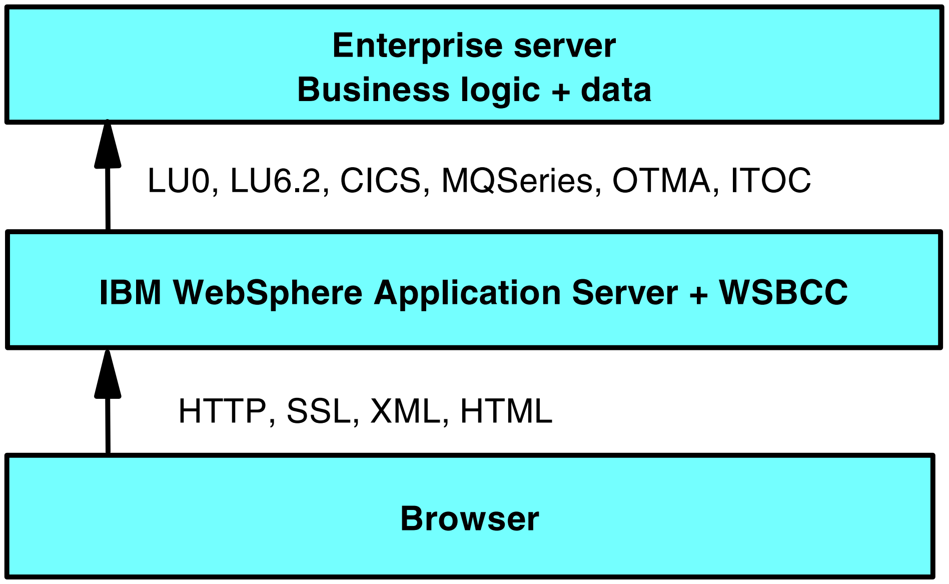

Enterprise server
Business logic + data
LU0, LU6.2, CICS, MQSeries, OTMA, ITOC
IBM WebSphere Application Server + WSBCC
HTTP, SSL, XML, HTML
Browser

Figure 7-5 WSBCC Web application architecture

The middle-tier server hardware communicates with the clients using a TCP/IP connection and the HTTP protocol. The WebSphere Application Server and its associated HTTP Server runs on the middle-tier server for this purpose. It processes requests from clients once the application is running. Handling client requests involves managing user navigation and interface, launching business operations that interact with back-end transactional systems, processing local transactions and sending HTML responses to the client running JSPs (or appropriate response for different channels).

A client is this architecture contains little or preferably no business logic and usually consists of presentation logic and first-level data validation. The code to execute the client logic is downloaded on demand from the middle-tier and does not reside on the client machine. In Internet banking, users want to access the financial services through a Web browser running on any device, typically a PC, connected to the Internet. The user interface is based on HTML and associated technologies. The HTML pages are almost never static and are dynamically generated on the middle-tier server using JSP technology. Data validation can be performed on the client-side by embedding JavaScript code in the HTML. This code remote execution cannot be trusted and acts as a client-side facility running inside the browser. Real data validation should be performed on the middle-tier server.

WSBCC elements

WSBCC runs on the middle-tier server. Its integration with the other tiers is based on the component model shown in Figure 7-6.

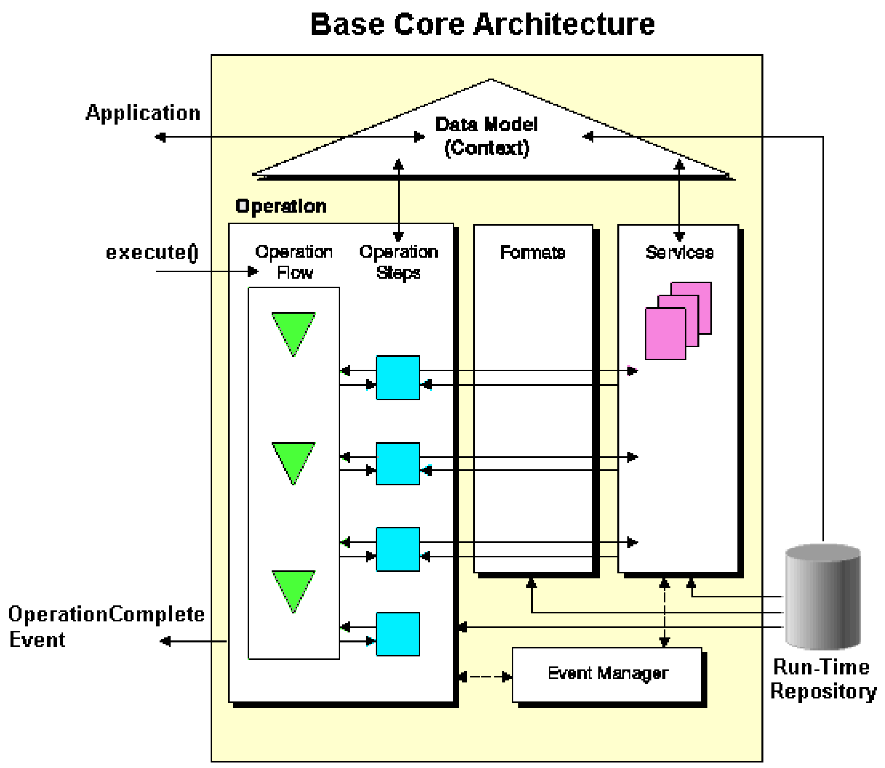


Figure 7-6 WSBCC component model

This model presents five key elements, or entities:

- Operations
- Data elements
- Formats
- Contexts
- Services

All these entities can be externalized in XML configuration files. Here we present them, what concepts they carry and what information is parametrized in the standard XML files.

Operation

An operation is the entity responsible for performing the set of tasks required to complete a basic financial operation, including data input and validation, interaction with external services, and management of the results and data received. An operation has a client that requests its execution, provides input data, and eventually receives the results.

An operation step is an entity that represents the set of interactions with the services that are required for a specific operation. Operation steps are managed by operations, and each operation's definition specifies the operation steps it will use.

Operations manage two basic attributes:

- Operation context—the context from which the operation flow requests the services to be used; requests that the data be formatted and sent to the services; and requests that the data received from the services is stored after it is interpreted. We explain more about the contexts below.
- Formatting services—the entities that are used to build the formatted data that is interchanged with the services and to interpret the data received from them.

An operation flow is normally common to many different operations, with the only differences being the data elements and the formats that are used in the interaction with the services. Because these differences can be handled by the formatting services, an operation flow is by nature a highly reusable part. Tasks inside the operation flow result in one or many operation steps. Operation steps are also highly reusable pieces of code that can be used by many different operations.

Context

When an operation is being performed, all the global data and services required by the application can be grouped into different sets of related information. Each of these sets of information logically belongs to a different type of banking entity: some related to the user, some to the branch, some to the client, some to the server, some to the whole banking institution, and so forth.

Each of these sets of related data and services makes up a context. The data used by an application can be considered as a context hierarchy, where each context level is able to provide the information it contains or the information belonging to contexts in upper levels. Figure 7-7 shows an example of context hierarchy.

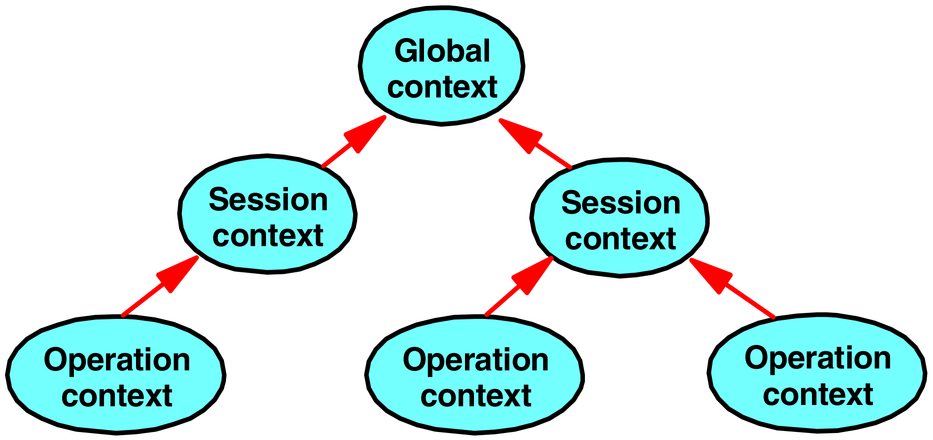


Figure 7-7 Context hierarchy

Each operation model has its own context, the operation context, with a specific set of operation data that includes elements for data input and for data received from external sources (for example, host or local DBM). Because the operation context is part of the context structure, the operation can access data at different levels in the context chain. When a service is requested, the operation will use the more specific service associated with the identification defined in the context chain.

An operation performed in a client/server environment has an operation context on the client and an operation context on the server. Each operation context is chained to one of the existing contexts when the operation is initiated.

Services

Services acts like connectors. They are defined in the context. When an operation wants to interact with a service, it uses a service alias to get a reference to the service.

Please note that the context hierarchy can contain more than one service with the same alias. In that case, the service with the requested alias which is closest to the operation context is returned.

Data elements

Each operation manages a set of data items, whose values are input from the client operation, shared from the contexts chain, received from external services, and so forth. These data elements may be used in various ways, such as being sent to the host, written as an electronic journal record, printed on a form, or passed to the client operation as results. For each operation step, data elements can be formatted differently, depending on the interacting service requirements.

The framework provides five base classes for dealing with data elements (Figure 7-8).

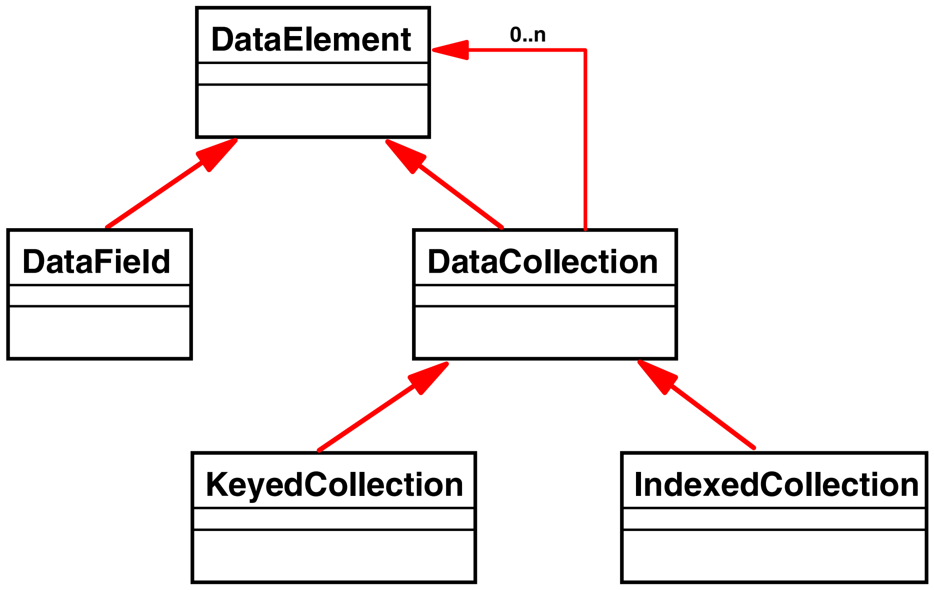


Figure 7-8 Data elements hierarchy

The data element hierarchy is extensible, and new classes can be derived easily when more functionality is needed. The classes that conform to the data hierarchy do not have exactly the same interface: only data fields have value, and only collections have `add`, `remove`, or `at` methods.

However, they have common instance variables such as name, and they share a common base class to be included inside collections (generally, collections deal with data elements). Methods for adding, retrieving, and deleting data elements are provided. There are also methods for setting and getting the value of the data elements contained in a collection. To maximize reusability of code, the `DataElement` class follows the composite design pattern, which is one in which any element of the collection can itself be a collection.

The five base data element classes are described as follows:

`DataElement`	This is the abstract base class for either single data items or data collections. It introduces a mandatory id instance variable for identification, and an optional description instance variable for description. Data elements can be typed or untyped, which depends on the value of its `PropertyDescriptor` attribute.
`DataField`	This is the representation of a single data item. It has a value instance variable for storing the data item value. Data fields are on the same conceptual level as a Java object attribute.
`DataCollection`	This is the abstract base class for composite data elements. Its subclasses are `KeyedCollection` and `IndexedCollection`.
`KeyedCollection`	This holds an ordered collection of data elements identified by their name or by their position in the collection. It is not possible to have two elements with the same name inside a collection, but elements can be other collections. It is possible to have another element with the same name in an inner collection. Keyed collections and their elements can be defined in the framework data definition file, or created and updated dynamically. Keyed collections are on the same conceptual level as Java objects. The term ‘collection’ is sometime confusing here.
`IndexedCollection`	This holds a data element that is repeated a number of times and identified by its position in the collection. Indexed collections are on the same conceptual level as Java collections.

To refer to a data element in an inner collection, you must provide the full path. For example, if the data field `dataField1` is inside `keyedCollection1`, which is inside `keyedCollection2`, then to access `dataField1`, you would specify: `keyedCollection1.dataField1`. Note that specifying `keyedCollection2` is not required because you are asking for its element called `keyedCollection1.dataField1`.

Another option is to use the `*` modifier, for example, `*.dataField1`. In this case, the first data element named `dataField1` is returned. The use of the `*` modifier is not recommended when there are different data elements with the same name in the same structure, or when the structure is very complex, because of the performance impact.

Typed data elements

The data elements described above are not aware of the type of the business objects they represent. Typed data elements can represent business objects, such as date, product, money, and have behavior that reflects the business rules that pertain to the business object they represent. The implementation of some business operations may require typed information in the data elements.

The framework provides the ability to work with or without typed data. Typed and untyped data elements can coexist at run time, and this allows each operation to be designed and implemented in the appropriate data typing mode. For example, a typed data element knows how to format itself for display, how to clone itself, and the nature of any validation required when requested to change its value.

The information that a data element knows about itself is made available by associating it with an object of the `PropertyDescriptor` class. Each `PropertyDescriptor` in turn is associated with a `Validator` and a `Converter`. A typed data element is an instance of a `DataElement` class in which the `PropertyDescription` property is not `null`.

One of the benefits of type-awareness is the ability to exploit object identity. Data elements that are type-aware can dynamically construct an identifier, which distinguishes them from other data elements of the same type. Note that this is object identity of a business object, not a Java object. For example, two instances of customer `123` are distinct Java objects, but are the same customer because their identifiers are equal.

Formats

Each operation manages a set of data items, whose values may be taken from input screens, other devices, shared data repositories (branch data, user data), host replies to a transaction, and so forth. This data must be formatted and combined to build the messages that are used in various ways, such as to send a transaction to the host, write a journal record, print a form, and so forth. For each of these steps, the data items can be formatted differently depending on the interacting object requirements (such as a host, electronic journal, financial printer), making the formatting process complex.

The objective of the hierarchies of format classes is to automate the formatting process as much as possible. The provided set of classes handles a large number of formatting situations. In addition, format classes are designed with extensibility as one of their main objectives because extending a class is the usual way of adding new required functionality.

The format classes are examples of the composite design pattern. They implement the concept of collections of elements that can themselves be collections.

The format classes also implement the decorator design pattern. Format decorators can be applied to (can "decorate") any format element to add functionality without requiring you to change it or subclass it.

The `FormatElement` hierarchy is similar in structure to the `DataElement` hierarchy. There are subclasses:

- `FieldFormat`, which are applied to format `DataField`
- `IndexedCollectionFormat`, which are applied to format `IndexedCollection`
- `KeyedCollectionFormat`, which are applied to format `KeyedCollection`

Development

See “WebSphere Business Components Composer” on page 303 for further information on application development with WebSphere Business Components Composer.

Part 3

Coding the application

In this part we discuss how to set up a development environment that will allow you to develop J2EE applications for WebSphere Application Server Version 4.0.

First we discuss development using the standard Java 2 software development kit (SDK), as well as using open-source build tools and tools from IBM.

We then provide some guidelines to assist in coding applications for WebSphere, and provide practical examples that show how to incorporate frameworks into your project.

Finally, we introduce software configuration management (SCM) and describe how Rational ClearCase can be used to provide SCM for a WebSphere development project.

8 Setting up a development environment

In this chapter we present some of the issues that you should consider when setting up a development environment for a J2EE application using WebSphere. We outline some of the decisions you will need to make, and highlight issues that you should consider before you start writing your application code.

Planning for development

It is always tempting at the start of a project to sit down and start coding straight away. We strongly recommend, however, that you take the time to plan out your environment and put in place the infrastructure to support your development effort. This is true now more than ever in the environments where WebSphere is typically deployed, with rapidly changing goals, high staff turnover and constrained deadlines.

A well-designed environment will save you time and money, and allow you to cope with the demands of developing applications today. In particular you should aim to:

- Plan for productivity
  - Provide tools to simplify and speed-up common tasks
  - Make use of frameworks and off-the-shelf components where appropriate
  - Reduce ramp-up time for new staff by using standard tools and processes wherever possible, and by documenting the complete environment
  - Automate wherever possible
- Plan for flexibility
  - Structure your code into stand-alone modules that can be re-used if requirements change or the project grows
- Plan for deployment
  - Make sure you can build and deploy your code quickly and easily
  - Include configurable logging and tracing in your code from day one
  - Consider application performance during every activity

The chapters that follow present some ideas you can use to plan and build a development environment suitable for your project's needs.

Defining the deliverables

The PiggyBank application is comprised of a number of subcomponents, each with its own deliverable module. The deliverables we consider here are:

EJB JAR file	This is a Java archive containing the code that implements the EJBs for the PiggyBank application. These EJBs are reusable components that provide functionality that may be reused by other Enterprise Java applications.
WAR file	This is a Web application archive containing all of the code and Web content that implements the Web-based front

	end for the PiggyBank application. It includes the servlets, JSPs and their supporting classes, as well as HTML and other static content.
Client JAR file	This is a Java archive that contains the code implementing a thick client front-end to the PiggyBank application. This client is a standalone Java application that uses the services provided by the PiggyBank EJBs.
Use case JAR file	This is a Java archive that contains our use case, or business logic access bean classes. This code is common to every client component that wishes to make use of the services provided by our application model.
Common JAR file	This is a separate Java archive that contains utility code used by all components in our application, for example data-only beans and helper classes for message logging and locating EJB homes in JNDI.

In terms of the J2EE specification, we should consider ourselves as developers to be *module* providers, that is to say we create the J2EE modules listed above for assembly into a complete application, stored in an enterprise archive file (EAR), by an application assembler (see Chapter 15, “Assembling the application” on page 389).

In reality, however, our work will stray into the assembly area, because we will need to create EAR files in order to deploy and unit test our code, and depending on the size and structure of our development organization, application assembly may well be a responsibility of the development team.

Because this book is concerned primarily with application development, and not assembly and deployment (in the true J2EE sense of the words), we focus mainly on the creation of J2EE modules. We do however cover the creation and deployment of a single EAR containing all of our modules for unit testing purposes.

Choosing your tools

In this redbook we discuss development for WebSphere Application Server Version 4.0 using three different sets of tools:

- The Java 2 SDK and Ant, an open source build tool from the Apache Jakarta project
- IBM VisualAge for Java Version 4.0
- WebSphere Studio Version 4.0

Although these topics are presented as three separate strands, you may find that elements of each solution are suitable for your environment.

This part of the redbook also discusses a number of other tools:

- WebSphere command framework
- Struts, from the Apache Jakarta project
- WebSphere Business Components Composer
- Logging and tracing with the WebSphere JRas facility and the Apache Jakarta project framework Log4J
- Rational ClearCase for Software Configuration Management (SCM)

Automation opportunities

Look for opportunities to automate tasks and processes in your development environment. Automation can deliver significant advantages to a development project. The potential benefits include:

- Improved developer productivity
- Reduced turnaround time for builds and code fixes
- Better consistency in application code
- Improved quality
- Reinforcement of development standards and policies

Within each discussions in this publication, we highlight areas where tooling and automation may pay dividends, and suggest ways in which you can leverage automation to improve your development environment.

9

Development using the Java 2 Software Development Kit

In this chapter we discuss how to develop and build a WebSphere application using the basic tools that come with the Java 2 Software Development Kit (SDK) and the base WebSphere Application Server product.

First we describe how the various files that make up a project may be organized in a directory structure, and how to use the SDK tools to compile and build J2EE modules for assembly into an application.

Next we investigate how Ant, a popular open source build tool from the Apache Jakarta project, can be used to automate these build tasks. It describes how to install and configure Ant to build the sample PiggyBank application.

The final part of this chapter discusses how to work with meta-data files, and provides hints and tips to assist you with developing your own J2EE applications.

If you plan to use more sophisticated tools during your development project, you may still find the discussions in this chapter give a useful overview of the low-level activities involved in building a J2EE application for WebSphere.

Organizing the project directory structure

It is well worth spending some time at the beginning of a project to consider how best to organize the source files that make up your application. When doing this, you should consider a number of factors, including:

- The number of developers working on the project, and their roles and experience
- How the deliverables from development will be structured into artifacts such as JAR and WAR files
- The tools you plan to use to assist with development
- How you expect the project to evolve as it moves from initial development into production and beyond

Just as no two projects are exactly alike, no one scheme fits every situation. In this chapter we present solutions for our example application, and attempt to justify and explain the decisions we have made.

All of the files required to develop and build the PiggyBank application are managed under a single directory structure. Under that top level directory we created separate directories for source code, intermediate code produced while building the application, and the deliverable application modules. We also created a directory for documentation, which includes the output from the analysis and design activities described in Part 2, "Analysis and design" on page 81, as well as that generated from the source code using the `javadoc` tool, and other documentation created during the course of the project.

We decided to further split the source code directory along the lines of deliverables, creating five separate source subtrees, one for each deliverable. The justifications for the split along these lines are:

- There are clear boundaries between code that will be deployed into different parts of the infrastructure
- We separate project specific code from the code that makes up reusable components
- Illegal dependencies between modules can be enforced at compile time—for example EJB code that depends upon a class in the servlet tree will not compile

We created a separate directory in the Web application part of the source tree in which to manage the Web content for the Web application, in which we include the JSPs as well as static files such as HTML and images.

Finally, within each source directory we created a `META-INF` directory to hold meta-data for each component. This meta-data includes manifest files used to create Java archives, and deployment descriptor information for the application client and EJB modules. We also created a `WEB-INF` directory in the Web application source tree to hold the J2EE Web application deployment descriptor.

Note: Windows NT does not make it easy to specify the case of directory names using Explorer. The `META-INF` and `WEB-INF` directory names must be in upper case in J2EE archive files. You can create the directories in upper case using Explorer or using the command line `mkdir` command, but Explorer may display the names in the incorrect case.

The final directory structure is illustrated in Figure 9-1.

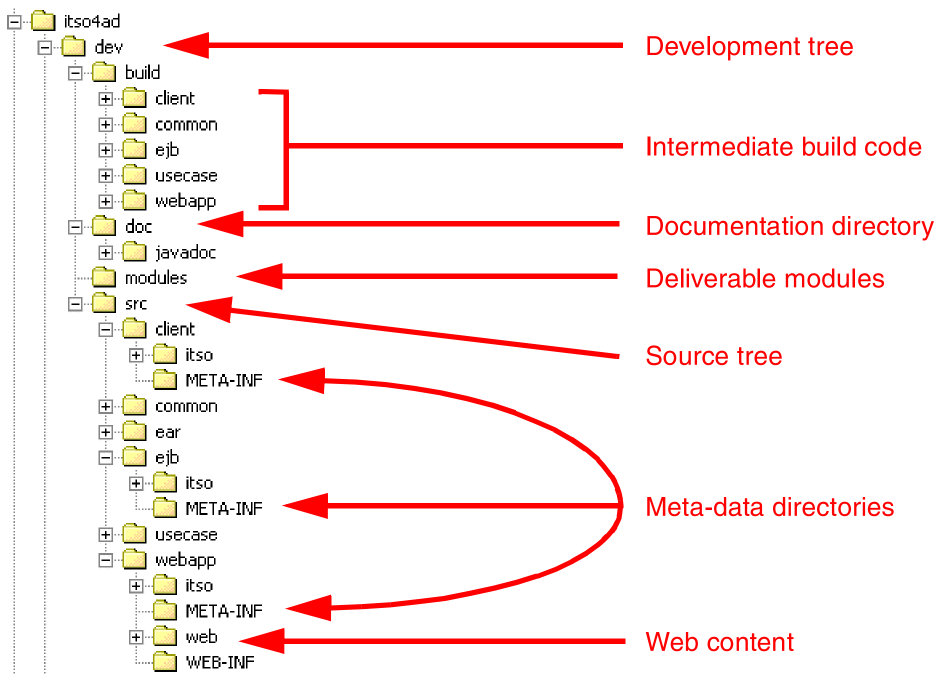


Figure 9-1 Development directory structure

Using the Java 2 SDK to build the application

It is entirely possible to build a complete J2EE application ready for assembly and deployment into WebSphere manually, using only the native operating system command-line and the basic tools supplied with the Java 2 SDK. Although we do not recommend this approach due to the tedious and error-prone nature of the steps involved, it is nevertheless a useful exercise to walk through the steps in order to better understand how to automate the task using scripts or a tool such as Ant, as discussed later in this chapter in “Using Ant to build a WebSphere application” on page 197.

The steps we discuss in more detail are as follows:

- Set up the environment
- Compile the source code
- Create JAR file containing the common code
- Create the EJB JAR file
- Create the use case JAR file
- Create the WAR file
- Create the client JAR file
- Generate documentation from the source code

Tools in the Java 2 SDK

First of all we briefly describe the tools that we use to build the application. These tools are all shipped with the standard Java 2 SDK, which is documented in full on the Sun Java Web site. We are using the Version 1.3 SDK that is shipped with WebSphere Application Server, and documented at

`http://java.sun.com/j2se/1.3/docs/tooldocs/tools.html`

For this example we use the following tools in conjunction with the Windows NT command line:

javac	This is the Java compiler. It takes Java source files and compiles them into class files containing Java byte code.
jar	This tool is used to manage Java archives, which are collections of multiple files rolled up into a single archive file.
javadoc	This tool processes Java source files looking for specially formatted comments that contain documentation about the nearby code. All of the Java API reference documentation is generated using the `javadoc` tool.

We are using the 1.3 SDK that is shipped with WebSphere Application Server Version 4.0. In our examples the WebSphere software is installed in `D:\WebSphere\AppServer`. The application source files in the development directory structure described in “Organizing the project directory structure” on page 184 are installed in `D:\ITSO4AD\dev`.

Setting up the environment

Before we can start to build the application, we must first set up our command line environment so that we are able to locate and run the tools in the SDK, and compile and package up the application code.

We must update our `PATH` so that we can locate the SDK tools. The SDK is installed in the `java` subdirectory of the WebSphere Application Server directory, and the tools are located in the SDK `bin` directory. We must also update our class path so that the compiler can locate the J2EE runtime libraries, as well as dependent application classes. The commands to update the `PATH` and `CLASSPATH` are shown in Figure 9-2.

```
set PATH=%PATH%;D:\WebSphere\AppServer\java\bin
set CLASSPATH=%CLASSPATH%;D:\WebSphere\AppServer\lib\j2ee.jar
set CLASSPATH=%CLASSPATH%;D:\ITSO4AD\dev\build\common
set CLASSPATH=%CLASSPATH%;D:\ITSO4AD\dev\build\ejb
set CLASSPATH=%CLASSPATH%;D:\ITSO4AD\dev\build\usecase
```

Figure 9-2 Setting environment variables for building on the command line

Note: We do not have to include the client and Web application build directories in the class path, because no other code should have dependencies upon classes in these directories. By not including them in the class path, we ensure that erroneous dependencies are discovered at compile time, and not during deployment.

Compiling the source code

We use the Java compiler, `javac`, to compile our source code into .class files. Before we can run the compiler we must first consider:

- Dependencies in our source code
- Compiler options we want to use

Code dependencies

We must take care to compile the sources in our directory structure in the correct order, taking into account the dependencies between our high-level code components. If we had simply placed all the source files in the same directory structure this would not be a concern, because `javac` searches for and automatically compiles dependent classes as required.

In our example, however, we want to restrict this behavior in order to reinforce boundaries in our architecture, and ensure that the reusable components we deliver really are reusable.

The dependencies in our code are illustrated in Figure 9-3.

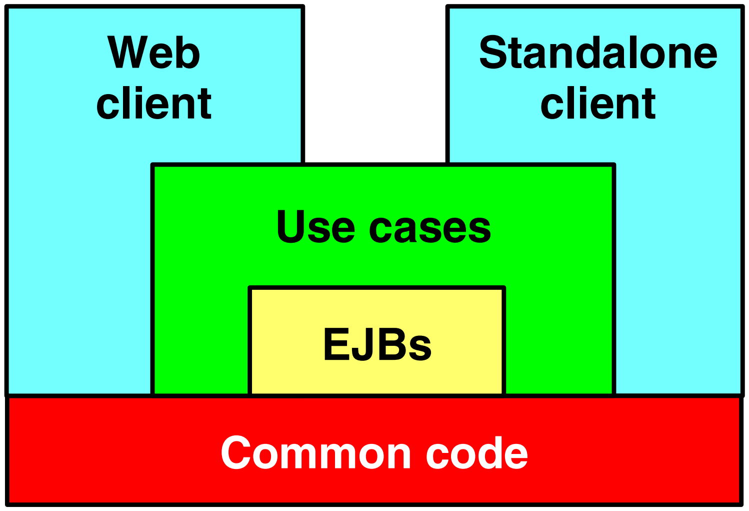


Figure 9-3 Code dependencies

The code dependencies dictate the compilation sequence:

- The utility code in our common directory structure is used by all our other code. This code must therefore be compiled first.
- The use case code also accesses the EJBs, so they must be compiled before the use cases.
- The two client modules use both the common code and the use cases, so these must be left until last.

Compiler options

`javac` accepts a number of options on the command line. In this example we use two in particular:

-d	Specifies the directory structure in which to write generated .class files. We specify this to make `javac` write its output to the directory structure under the build directory.
-g	Generates .class files with complete debugging information. This option is required to enable full debugging support. When compiling for a production system we may choose to use the `-g:none` option to remove the debugging information.

We do not have to specify the `-classpath` parameter because `javac` will pick this up from the environment.

Running the compiler

`javac` allows us to specify multiple files to compile in a single command. We can do this either by listing the files one by one on the command line, or by providing a list of files in another file. We choose this second option, using the `dir` command to create the list to ensure we do not miss any source files. Figure 9-4 shows the sequence of commands required to compile the source files that make up the common components.

```
D:\>cd ITSO4AD\dev\src\common

D:\ITSO4AD\dev\src\common>dir /s /b *.java > filelist

D:\ITSO4AD\dev\src\common>javac -g -d D:\ITSO4AD\dev\build\common @filelist
```

Figure 9-4 Compiling the source files for the common code

This procedure must be repeated for the EJB source files, then the use case, standalone client and servlet sources. For each subtree we change to the base directory where the source is located, create the file list and execute the Java compiler. Once we have completed all five sets of compilations we have a full set of .class files in the directory structure under `D:\ITSO4AD\dev\build`.

Creating the common JAR file

We place all of the common code in a single Java archive file. Because this is a regular JAR file, as opposed to an EJB JAR or WAR file, there are no deployment descriptors include. We have to include information from our own manifest file, however. This manifest information includes an entry that we use to specify the class path for the common code.

When the common code module is eventually packaged into a J2EE enterprise archive (EAR) file, WebSphere will use this class path to locate classes that our common code needs. We use the `m` option to specify the manifest file, which is located in the `META-INF` directory of the common source tree. We archive all of the class files under the `build\common` directory, and place the archive in the `modules` directory (Figure 9-5).

```
D:\ITSO4AD\dev\build>cd common

D:\ITSO4AD\dev\build\common>jar cvmf D:\ITSO4ad\dev\src\common\META-INF\MANIFEST.MF
                                D:\ITSO4AD\dev\modules\piggybank-common.jar *
added manifest
adding: itso/(in = 0) (out= 0)(stored 0%)
adding: itso/was4ad/(in = 0) (out= 0)(stored 0%)
adding: itso/was4ad/data/(in = 0) (out= 0)(stored 0%)
adding: itso/was4ad/data/CustomerData.class(in = 805) (out= 394)(deflated 51%)
adding: itso/was4ad/databean/(in = 0) (out= 0)(stored 0%)
adding: itso/was4ad/databean/AccountData.class(in = 1257) (out= 569)(deflated 54%)
adding: itso/was4ad/databean/CustomerData.class(in = 860) (out= 468)(deflated 45%)
adding: itso/was4ad/exception/(in = 0) (out= 0)(stored 0%)
adding: itso/was4ad/exception/BusinessException.class(in = 909) (out= 495)(deflated 45%)
adding: itso/was4ad/exception/CustomerNotAuthorized.class(in = 638) (out= 357)(deflated 44%)
adding: itso/was4ad/exception/InsufficientFunds.class(in = 626) (out= 350)(deflated 44%)
adding: itso/was4ad/exception/InvalidCheque.class(in = 614) (out= 349)(deflated 43%)
adding: itso/was4ad/exception/InvalidOperation.class(in = 623) (out= 350)(deflated 43%)
adding: itso/was4ad/exception/NonExistentAccount.class(in = 629) (out= 354)(deflated 43%)
adding: itso/was4ad/exception/NonExistentCustomer.class(in = 632) (out= 354)(deflated 43%)
adding: itso/was4ad/helpers/(in = 0) (out= 0)(stored 0%)
adding: itso/was4ad/helpers/HomeHelper.class(in = 2814) (out= 1366)(deflated 51%)
adding: itso/was4ad/helpers/LogHelper.class(in = 1930) (out= 770)(deflated 60%)
```

Figure 9-5 Creating the common JAR file

Note: The `jar` command in Figure 9-5 must be entered on a single line.

Creating the EJB JAR file

We create the EJB JAR file in a similar manner to the common JAR file, packaging up all of the class files in the `build\ejb` directory.

The EJB JAR file differs from a regular JAR file, however, in that it also requires deployment descriptor information that describes the EJBs found in the JAR file. Our EJB deployment descriptor is named `ejb-jar.xml` and is located in the `src\ejb\META-INF` directory. The same directory also contains our manifest information, and files that describe WebSphere-specific deployment information; see “Working with meta-data” on page 226 to learn more about these additional meta-data files.

Figure 9-6 shows how we use the -C option on the jar command to change to a new directory before adding a selection of files. The -C option is used to add the deployment descriptor information stored in the source tree. The resulting EJB JAR file is stored in the modules directory.

```
D:\ITSO4AD\dev\build\common>cd ..\ejb

D:\ITSO4AD\dev\build\ejb>jar cvmf D:\ITSO4ad\dev\src\ejb\META-INF\MANIFEST.MF
              D:\ITSO4AD\dev\modules\piggybank-ejb.jar -C D:\ITSO4AD\dev\src\ejb META-INF *
added manifest
ignoring entry META-INF/
adding: META-INF/ejb-jar.xml(in = 4634) (out= 814)(deflated 82%)
adding: META-INF/ibm-ejb-jar-bnd.xmi(in = 1307) (out= 408)(deflated 68%)
adding: META-INF/ibm-ejb-jar-ext.xmi(in = 4756) (out= 695)(deflated 85%)
ignoring entry META-INF/MANIFEST.MF
adding: META-INF/Map.mapxmi(in = 2936) (out= 590)(deflated 79%)
adding: META-INF/Schema.rdbxmi(in = 2688) (out= 597)(deflated 77%)
adding: META-INF/Table.ddl(in = 384) (out= 223)(deflated 41%)
adding: itso/(in = 0) (out= 0)(stored 0%)
adding: itso/was4ad/(in = 0) (out= 0)(stored 0%)
adding: itso/was4ad/ejb/(in = 0) (out= 0)(stored 0%)
adding: itso/was4ad/ejb/account/(in = 0) (out= 0)(stored 0%)
adding: itso/was4ad/ejb/account/Account.class(in = 370) (out= 250)(deflated 32%)
adding: itso/was4ad/ejb/account/AccountBean.class(in = 4544) (out= 2100)(deflated 53%)
adding: itso/was4ad/ejb/account/AccountBeanFinderHelper.class(in = 254) (out= 201)(deflated 20%)
adding: itso/was4ad/ejb/account/AccountHome.class(in = 507) (out= 285)(deflated 43%)
adding: itso/was4ad/ejb/account/AccountKey.class(in = 852) (out= 533)(deflated 37%)
adding: itso/was4ad/ejb/account/AccountManager.class(in = 604) (out= 351)(deflated 41%)
adding: itso/was4ad/ejb/account/AccountManagerBean.class(in = 8037) (out= 3514)(deflated 56%)
adding: itso/was4ad/ejb/account/AccountManagerHome.class(in = 314) (out= 209)(deflated 33%)
adding: itso/was4ad/ejb/customer/(in = 0) (out= 0)(stored 0%)
adding: itso/was4ad/ejb/customer/Customer.class(in = 339) (out= 226)(deflated 33%)
adding: itso/was4ad/ejb/customer/CustomerBean.class(in = 5756) (out= 2598)(deflated 54%)
adding: itso/was4ad/ejb/customer/CustomerBeanFinderHelper.class(in = 154) (out=120)(deflated 22%)
adding: itso/was4ad/ejb/customer/CustomerHome.class(in = 465) (out= 258)(deflated 44%)
adding: itso/was4ad/ejb/customer/CustomerKey.class(in = 849) (out= 531)(deflated 37%)
adding: itso/was4ad/ejb/customer/CustomerManager.class(in = 563) (out= 307)(deflated 45%)
adding: itso/was4ad/ejb/customer/CustomerManagerBean.class(in = 7629) (out= 3213)(deflated 57%)
adding: itso/was4ad/ejb/customer/CustomerManagerHome.class(in = 319) (out= 208)(deflated 34%)
```

Figure 9-6 Creating the EJB JAR file

Note: We have to make sure that the META-INF directory name in the archive is correctly specified in upper case, even though the Windows file system is not case sensitive. We do this by the using upper case name in the parameters to the jar command.

Creating the use case JAR file

The use case JAR file is just like the common JAR file in that is a regular Java archive, and thus does not have any J2EE deployment descriptors. We do, however, require our own manifest so that we can specify a class path that reflects the dependencies the use case code has upon the common and EJB JAR files. We place the manifest and the compiled use case .class files in the archive `piggybank-usecase.jar` in the `modules` directory (Figure 9-7).

```
D:\ITSO4AD\dev\build\usecase>jar cvmf D:\ITSO4AD\dev\src\usecase\META-INF\MANIFEST.MF
                                D:\ITSO4AD\dev\modules\piggybank-usecase.jar *
added manifest
adding: itso/(in = 0) (out= 0)(stored 0%)
adding: itso/was4ad/(in = 0) (out= 0)(stored 0%)
adding: itso/was4ad/usecase/(in = 0) (out= 0)(stored 0%)
adding: itso/was4ad/usecase/CashCheck.class(in = 1276) (out= 607)(deflated 52%)
adding: itso/was4ad/usecase/CreateCustomer.class(in = 2014) (out= 989)(deflated 50%)
adding: itso/was4ad/usecase/DisplayAccount.class(in = 1680) (out= 843)(deflated 49%)
adding: itso/was4ad/usecase/DisplayBalance.class(in = 832) (out= 457)(deflated 45%)
adding: itso/was4ad/usecase/DisplayCustomer.class(in = 1695) (out= 846)(deflated 50%)
adding: itso/was4ad/usecase/DisplayCustomerAccounts.class(in = 1742) (out=
859)(deflated 50%)
adding: itso/was4ad/usecase/DisplayCustomerDetail.class(in = 1733) (out=
855)(deflated 50%)
adding: itso/was4ad/usecase/OpenAccount.class(in = 1026) (out= 549)(deflated 46%)
adding: itso/was4ad/usecase/Transfer.class(in = 1301) (out= 601)(deflated 53%)
adding: itso/was4ad/usecase/UseCase.class(in = 2481) (out= 1119)(deflated 54%)
```

Figure 9-7 Creating the use case JAR file

Creating the WAR file

The WAR file needs to contain the following items:

- The compiled servlet code, in the `WEB-INF\classes` directory in the archive
- The Web content from the `src\webapp\web` directory
- The Web application deployment descriptor, `web.xml`, also stored in the `WEB-INF` directory in the archive
- Additional WebSphere-specific deployment information, also in the `WEB-INF` directory (see “Working with meta-data” on page 226 to find out about the WebSphere-specific meta-data)
- Manifest information from the `src\webapp\META-INF\MANIFEST.MF` file

In order to create this archive, we must first copy some of the files we require into a temporary directory structure, because the command line parameters for the `jar` command do not offer us enough flexibility to create the required structure.

Instead we must create a temporary directory named `D:\temp\wartemp`, and copy the servlet code and meta-data files to the appropriate place under the temporary structure (Figure 9-8).

```
D:\ITSO4AD\dev\build\ejb>cd ..\webapp

D:\ITSO4AD\dev\build\webapp>mkdir D:\temp\wartemp\WEB-INF\classes

D:\ITSO4AD\dev\build\webapp>xcopy * D:\temp\wartemp\WEB-INF\classes /s
D:itso\was4ad\webapp\command\CommandConstants.class
D:itso\was4ad\webapp\command\Login.class
D:itso\was4ad\webapp\command\Logout.class
D:itso\was4ad\webapp\command\MainMenu.class
D:itso\was4ad\webapp\controller\Command.class
D:itso\was4ad\webapp\controller\ControllerServlet.class
D:itso\was4ad\webapp\controller\Error.class
7 File(s) copied

D:\ITSO4AD\dev\build\webapp>xcopy D:\ITSO4AD\dev\src\webapp\WEB-INF D:\temp\wartemp\WEB-INF /s
D:\ITSO4AD\dev\src\webapp\WEB-INF\ibm-web-bnd.xmi
D:\ITSO4AD\dev\src\webapp\WEB-INF\ibm-web-ext.xmi
D:\ITSO4AD\dev\src\webapp\WEB-INF\web.xml
3 File(s) copied
```

Figure 9-8   Creating temporary directory structure used for building the WAR file

Figure 9-9 shows how we can now create the WAR file in the `modules` directory, using the `-C` parameter on the `jar` command to pull in the files from the other locations, before removing the temporary directory.

```
D:\ITSO4AD\dev\build\webapp>jar cvmf D:\ITSO4AD\dev\src\webapp\META-INF\MANIFEST.MF
                            D:\ITSO4AD\dev\modules\piggybank-webapp.war -C D:\temp\wartemp WEB-INF
                            -C D:\ITSO4AD\dev\src\webapp\web .
added manifest
adding: WEB-INF/(in = 0) (out= 0)(stored 0%)
adding: WEB-INF/classes/(in = 0) (out= 0)(stored 0%)
adding: WEB-INF/classes/itso/(in = 0) (out= 0)(stored 0%)
adding: WEB-INF/classes/itso/was4ad/(in = 0) (out= 0)(stored 0%)
adding: WEB-INF/classes/itso/was4ad/webapp/(in = 0) (out= 0)(stored 0%)
adding: WEB-INF/classes/itso/was4ad/webapp/command/(in = 0) (out= 0)(stored 0%)
adding: WEB-INF/classes/itso/was4ad/webapp/command/CommandConstants.class(in = 645) (out=
397)(deflated 38%)
adding: WEB-INF/classes/itso/was4ad/webapp/command/Login.class(in = 1864) (out = 930)(deflated
50%)
adding: WEB-INF/classes/itso/was4ad/webapp/command/Logout.class(in = 871) (out = 449)(deflated
48%)
adding: WEB-INF/classes/itso/was4ad/webapp/command/MainMenu.class(in = 801) (out = 417)(deflated
47%)
adding: WEB-INF/classes/itso/was4ad/webapp/controller/(in = 0) (out= 0)(stored 0%)
adding: WEB-INF/classes/itso/was4ad/webapp/controller/Command.class(in = 325) (out = 212)(deflated
34%)
adding: WEB-INF/classes/itso/was4ad/webapp/controller/ControllerServlet.class(in = 6826) (out =
3043)(deflated 55%)
adding: WEB-INF/classes/itso/was4ad/webapp/controller/Error.class(in = 865) (out = 481)(deflated
44%)
adding: WEB-INF/ibm-web-bnd.xmi(in = 289) (out= 178)(deflated 38%)
adding: WEB-INF/ibm-web-ext.xmi(in = 468) (out= 265)(deflated 43%)
adding: WEB-INF/web.xml(in = 2471) (out= 753)(deflated 69%)
adding: error.jsp(in = 2931) (out= 921)(deflated 68%)
adding: images/(in = 0) (out= 0)(stored 0%)
adding: images/b_lis019.gif(in = 138) (out= 122)(deflated 11%)
adding: images/logo.gif(in = 4895) (out= 4815)(deflated 1%)
adding: images/PoweredByWebSphere.gif(in = 1533) (out= 1190)(deflated 22%)
adding: index.html(in = 2373) (out= 750)(deflated 68%)
adding: login.jsp(in = 2863) (out= 816)(deflated 71%)
adding: loginfail.jsp(in = 2919) (out= 845)(deflated 71%)
adding: logout.jsp(in = 2298) (out= 710)(deflated 69%)
adding: theme/(in = 0) (out= 0)(stored 0%)
adding: theme/Master.css(in = 253) (out= 157)(deflated 37%)
adding: welcome.jsp(in = 3246) (out= 1004)(deflated 69%)

D:\ITSO4AD\dev\build\webapp>rmdir /s D:\temp\wartemp
D:\temp\wartemp, Are you sure (Y/N)? y
```

Figure 9-9 Creating the WAR file

Note: The `WEB-INF` directory name must be spelled in upper case in the parameters to the `jar` command.

Creating the client JAR file

The steps to create the client JAR file are essentially the same as for the EJB JAR file, pulling in instead the compiled classes from the `build\client` directory, the manifest information, and the deployment descriptor files from the `src\client\META-INF` directory. The commands to run are illustrated in Figure 9-10.

```
D:\ITSO4AD\dev\build\webapp>cd ..\client

D:\ITSO4AD\dev\build\client>jar cvmf D:\ITSO4AD\dev\src\client\META-INF\MANIFEST.MF
        D:\ITSO4AD\dev\modules\piggybank-client.jar -C D:\ITSO4AD\dev\src\client META-INF *
added manifest
ignoring entry META-INF/
adding: META-INF/application-client.xml(in = 1007) (out= 359)(deflated 64%)
adding: META-INF/ibm-application-client-bnd.xmi(in = 720) (out= 270)(deflated 62%)
ignoring entry META-INF/MANIFEST.MF
adding: itso/(in = 0) (out= 0)(stored 0%)
adding: itso/was4ad/(in = 0) (out= 0)(stored 0%)
adding: itso/was4ad/client/(in = 0) (out= 0)(stored 0%)
adding: itso/was4ad/client/StandaloneClient.class(in = 10398) (out= 4829)(deflated 53%)
```

Figure 9-10 Creating the client JAR file

Note: `WEB-INF` must be spelled in upper case in the `jar` command.

Generating documentation

`javadoc` provides a large number of options that control the output from the tool. In this example we use a simple subset to generate the documentation for our code, using the standard doclet supplied with the Java 2 SDK. For a more complete description of the `javadoc` tool see the SDK documentation at:

`http://java.sun.com/j2se/1.3/docs/tooldocs/javadoc/index.html`

javadoc options

The options that we use to invoke javadoc are described below:

-private	This option causes the tool to generate documentation for all classes and members, regardless of their visibility. We chose this option because the intended readers of the documentation will be maintaining the application code, not just using an implemented API.
-d	This option defines the target directory for the generated HTML. We decided to generate the documentation for all of the code into a single directory, the `D:\ITSO4AD\dev\doc\javadoc` directory.
-use	This option causes pages describing class and package usage to be generated
-windowtitle	This option defines title of the browser window in which the documentation is displayed
-doctitle	This option defines the title on the documentation index page

Running javadoc

The commands we used to execute the `javadoc` tool are shown in Figure 9-11. The tool allows the list of files to process to be specified in the same way as the Java compiler, so we re-used the files we created in “Compiling the source code” on page 187. The tool writes a substantial amount of status information to the console while it is running—we have not included this output.

```
D:\ITSO4AD\dev\build\client>cd D:\ITSO4AD\dev\src

D:\ITSO4AD\dev\src>javadoc -private -d D:\ITSO4AD\dev\doc\javadoc -use
                           -windowtitle "PiggyBank Documentation"
                           -doctitle "PiggyBank Documentation"
                           @common/filelist @ejb/filelist @usecase/filelist
                           @client/filelist @webapp/filelist
```

Figure 9-11 Running the javadoc tool

Viewing the generated documentation

Once the `javadoc` tool has completed we can use a Web browser to view the generated documentation (Figure 9-12). Simply open a browser on the `index.html` file in the destination directory, which in our case is `D:\ITSO4AD\dev\doc\javadoc`.

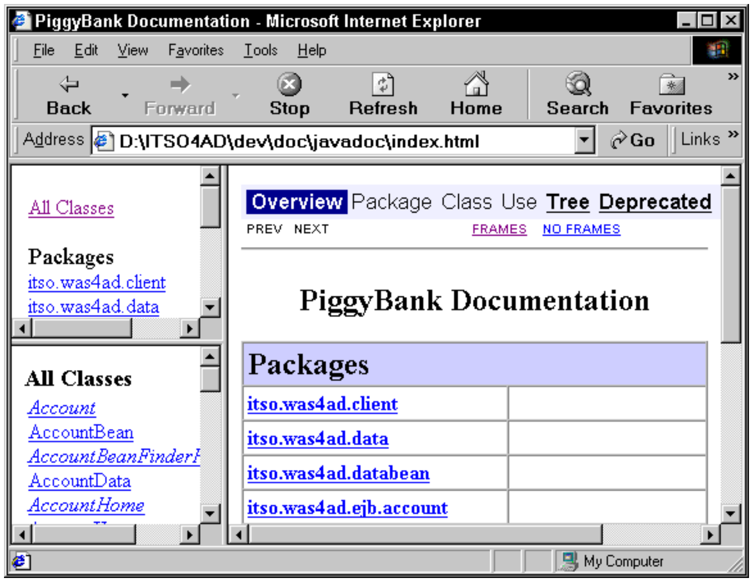


Figure 9-12 Viewing the generated documentation

Using Ant to build a WebSphere application

It is common practice in all but the most trivial projects to employ tools to simplify the task of building the application. The reasons are clear—the tasks are generally simple but may be long-winded, and during the course of development will be repeated many times by each developer. Tools speed up the process and reduce the risk of errors, leading to a repeatable, reliable process that is essential through multiple development cycles.

Traditionally these tasks have been performed by shell scripts or batch files in UNIX or Windows environments, or by using tools such as `make`. While these approaches are still valid, developing Java applications—especially in a heterogeneous environment—introduces new challenges. A particular limitation is the close-coupling to a particular operating system inherent in using these tools.

What is Ant?

Ant attempts to solve some of these issues by providing a framework that implements extensions in Java, instead of issuing shell commands to perform build tasks. The base Ant package comes with a comprehensive set of standard extensions (known as *tasks* in Ant) for performing common actions such as compiling source code and manipulating files. If a project requires a more specialized task, and a suitable task is not already available in the standard optional library, it is possible to write your own tasks in Java.

Ant is a subproject of the Apache Jakarta project, part of the Apache Software Foundation. This is the organization responsible for the open source Apache Web server, the basis of the IBM HTTP Server shipped with WebSphere Application Server. The goal of the Jakarta project is to *“provide commercial-quality server solutions based on the Java Platform that are developed in an open and cooperative fashion.”*

To find out more about the Jakarta project visit the Jakarta Web site at:

`http://jakarta.apache.org/`

The Ant Web site is located at:

`http://jakarta.apache.org/ant/`

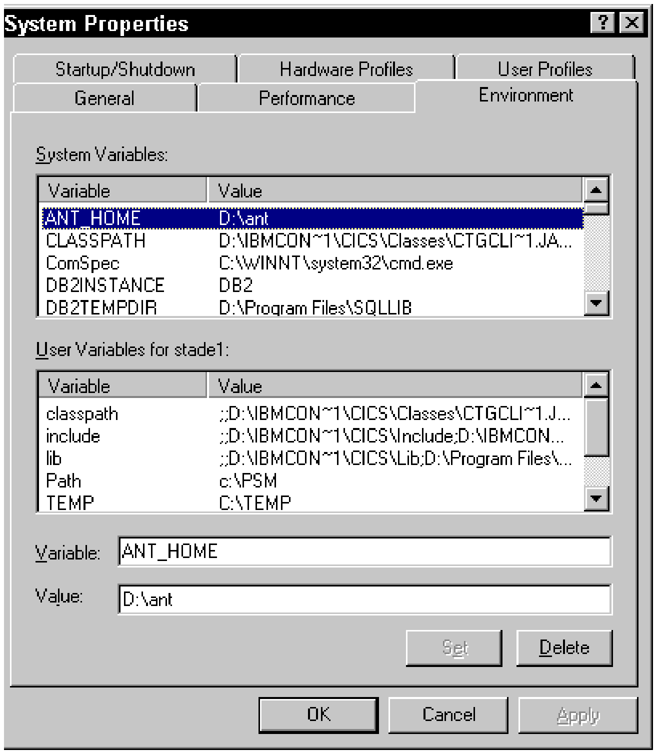
This section provides a basic outline of the features and capabilities of Ant. For complete information you should consult the Ant documentation included in the Ant distribution or available on the Web at:

`http://jakarta.apache.org/ant/manual/`

Installing and configuring Ant

First of all we downloaded the Ant binary distribution from the Ant Web site. All of the examples in this chapter have been developed and tested using the latest release of Ant available at the time of writing, Version 1.3.

We unpacked the binary distribution file into a temporary directory and copied all of the files into `D:\ant`. Following the Ant installation guide we then set the environment variables `ANT_HOME` and `JAVA_HOME`, and updated our `PATH`. We did this for the entire machine by selecting *Start -> Setting -> Control Panel -> System -> Environment*, and editing the values as shown in Figure 9-13.



System Variables:

Variable	Value
ANT_HOME	D:\ant
CLASSPATH	D:\IBMCON~1\CICS\Classes\CTGCLI~1.JA...
ComSpec	C:\WINNT\system32\cmd.exe
DB2INSTANCE	DB2
DB2TEMPDIR	D:\Program Files\SQLLIB

User Variables for stade1:

Variable	Value
classpath	;;D:\IBMCON~1\CICS\Classes\CTGCLI~1.J...
include	;;D:\IBMCON~1\CICS\Include;D:\IBMCON...
lib	;;D:\IBMCON~1\CICS\Lib;D:\Program Files\...
Path	c:\PSM
TEMP	C:\TEMP

Variable: ANT_HOME

Value: D:\ant

Figure 9-13 Setting environment variables

We set `ANT_HOME` to the base location of the Ant software, in our case `D:\ant`. `JAVA_HOME` is used by Ant to determine the location of the JDK used to run the tool; in our case we decided to use the JDK shipped with WebSphere, which we have installed in `D:\WebSphere\AppServer\java`. We also added `D:\ant\bin` to the `PATH` environment variable, so we could pick up the `Ant` executable.

Tip: We also edited our existing `CLASSPATH` environment variable to remove spaces from it—the batch file that starts the `ant` command does not work when there are spaces in the class path. In our case we found that DB2 had added paths including `D:\Program Files`—we changed them to `D:\Progra~1`.

The final configuration step was to install the optional Ant tasks that are also available on the Web site in a separate package from the main Ant distribution. We downloaded the JAR containing the optional tasks and installed it into the `D:\ant\lib` directory.

Tip: The script that starts the `ant` command automatically adds any JAR files in the `lib` directory under `ANT_HOME` to the class path when running Ant.

Ant build files

Ant uses XML *build files* to describe what tasks must be performed in order to build a project. The main components of a build file are:

project	A build file contains build information for a single project. It may contain one or more *targets*.
target	A target describes the *tasks* that must be performed to satisfy a goal, for example compiling source code into .class files may be one target, and packaging the .class files into a JAR file may be another target. Targets may depend upon other targets, for example the .class files must be up to date before you can create the JAR file. Ant will resolve these dependencies.
task	A task is a single step that must be performed in order to satisfy a target. Tasks are implemented as Java classes that are invoked by Ant, passing parameters defined as attributes in the XML. Ant provides a set of standard tasks, a set of optional tasks, and an API which allows you to write your own tasks.
property	A property has a name and a value. Properties are essentially variables that can be passed to tasks via task attributes. Property values can be set inside a build file, or obtained externally from a property file or from the command line. A property is referenced by enclosing the property name inside `${}`, for example `${basedir}`.
path	A path is a set of directories or files. Paths can be defined once and referred to multiple times, easing the development and maintenance of build files. For example, a Java compilation task may use a path reference to determine the class path to use.

Note: On Windows systems Ant recognizes both forward slash (/) and back slash (\) characters as directory name separators. In order to maintain portability of our build files so they may run unaltered on UNIX systems we always use the forward slash character in our build files. Where we have to specify drive letters, we do this only in property files and not in the build files themselves.

Built-in tasks

A comprehensive set of built in tasks are supplied with the Ant distribution. The tasks that we use in this example are described below:

ant	Invokes Ant using another build file
copy	Copies files and directories
delete	Deletes files and directories
echo	Outputs messages
jar	Creates Java archive files
javac	Compiles Java source
javadoc	Generates documentation from Java source
mkdir	Creates directories
tstamp	Sets properties containing date and time information
war	Creates WAR files

Creating build files for the PiggyBank application

We take the development tree structure described in “Organizing the project directory structure” on page 184, and develop Ant build files that can be used to create the same deliverables we created in “Using the Java 2 SDK to build the application” on page 186, namely:

- A JAR file containing common code
- An EJB JAR file containing EJB code
- A JAR file containing the use case code
- A client JAR file containing application client code
- A WAR file containing the code and content for the Web client
- Generated documentation for all Java source

We split the PiggyBank application into subprojects, each with its own build file, and invoke the subprojects from a master build file that resides in the base source directory, as illustrated in Figure 9-14.

This strategy emphasizes the modular nature of the application architecture. Internal changes in one subproject should not necessitate rebuilding every deliverable, and reusable components, particularly the use cases, EJBs and common code, may be separated out from the rest of the application with minimal pain.

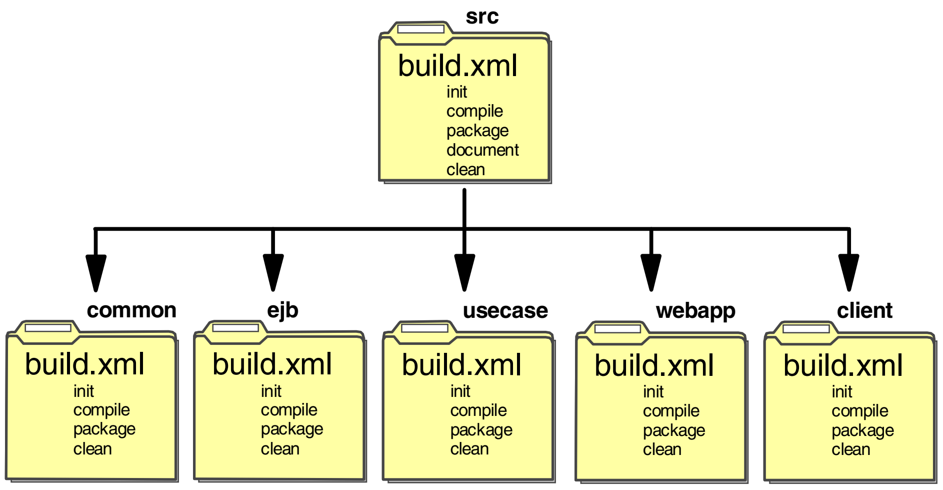


Figure 9-14 Organizing Ant build files into subprojects

The build files are all named `build.xml`. This is the default name assumed by Ant if no build file name is supplied. This allows a build of the entire project or a single subproject to be performed by simply changing to the appropriate directory and issuing the `ant` command with no arguments. Each build file has the following common targets:

init	Performs build initialization tasks—all other targets depend upon this target
compile	Compiles Java source into .class files
package	Creates the deliverables for each module—depends upon the `compile` target
clean	Removes all generated files—used to force a full build

The master build file has additional targets that are appropriate for the entire project, for example to generate documentation from the Java source code.

Each Ant build file may have a default target. This target is executed if Ant is invoked on a build file and no target is supplied as a parameter. In all cases the default target for our build files is `package`. The dependencies between targets are illustrated in Figure 9-15.

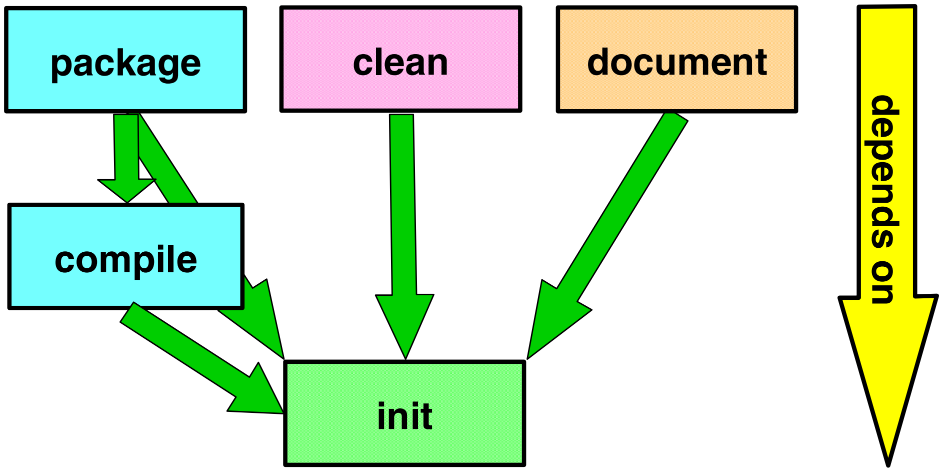


Figure 9-15 Dependencies between build targets

Master build file

The master build file sets global properties and delegates responsibility for actually building code to the individual subprojects' build files.

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

<project name="itso4ad" default="package">

  <!-- Properties and targets are defined here -->

</project>
```

Figure 9-16 Master build file outer tags

Figure 9-16 shows the outer tags from the master build file. As we progress through this section of the book we add XML fragments inside the project tags, building up the file as we go. See "Using the Web material" on page 558 for the location of the complete master build file.

Note: Although Ant build files use XML, there is no single document type definition (DTD) that can define the document structure. This is because each task adds its own tags, and new tasks can easily be written and added to Ant.

Setting global properties

We use properties in our build files to provide information that may change over time or from machine to machine. We have to provide a sensible default set of values for each property we use, while also allowing individuals to override those defaults where appropriate.

A good example is the location of the WebSphere Application Server software. This information is required in order to locate libraries that we have to compile our code against, but it may differ from machine to machine, depending on where the software is installed.

The solution is to use a hierarchy of property files. Ant is able to read properties from files that use the format recognized by the Java `java.util.Properties` class. Once a property has been set, however, it may not be changed. We first check the current user's home directory for a properties file and use any properties defined there, and then read any remaining properties from a file called `global.properties` that is stored with the master build file.

The XML that implements this scheme is shown in Figure 9-17. The `user.home` property is provided by Ant and is set to the home directory of the user who started Ant. On our Windows NT system, for example, this resolves to `C:\WinNT\Profiles\resident`. The `basedir` property is also set by Ant, and defaults to the location of the build file.

```
<!-- Set up global properties -->
<property name="global.dev.dir" value="${basedir}/.."/>
<property file="${user.home}/override.properties"/>
<property file="${global.dev.dir}/global.properties"/>
```

Figure 9-17 Setting global properties in the master build file

The `global.dev.dir` property is used to make the build location independent. No matter where the development tree is actually installed, the build will always work, because all paths are calculated relative to this property. We use almost identical XML to set global properties in the build files for each subproject. The only difference is that the `global.dev.dir` property will be defined as being two directory levels up. This is shown in Figure 9-18.

```
<!-- Set up global properties -->
<property name="global.dev.dir" value="${basedir}/../.."/>
<property file="${user.home}/override.properties"/>
<property file="${global.dev.dir}/global.properties"/>
```

Figure 9-18 Setting global properties in a subproject build file

This allows the subproject build files to locate the global properties in the case where Ant is executed against the subproject build file directly, as well as when invoked from the master build file.

The initial global properties in `global.properties` are shown in Figure 9-19. Note that unlike in regular Java property files we are able to reference other properties in property values. Ant resolves references at the time the property is set, so properties must be defined in the correct order.

```
#
# Global build properties file
#
# If you need to override anything in here create an
# override.properties file in your home directory
#

#
# Software locations
#
global.was.dir=D:/WebSphere/AppServer

#
# Destination directories
#
global.build.dir=${global.dev.dir}/build
global.module.dir=${global.dev.dir}/modules
global.javadoc.dir=${global.dev.dir}/doc/javadoc
```

Figure 9-19 Global properties file global.properties

As we work though the examples we add additional properties to this file as required.

If we have to override one of the default settings, we can create an `override.properties` file in our home directory. An example is shown in Figure 9-20.

```
#
# Override the WebSphere software location
#
global.was.dir=C:/WebSphere/AppServer
```

Figure 9-20 Overriding default properties in override.properties

It is also possible to override properties on the ant command line using the `-D` parameter of the `ant` command, for example:

```
ant -Dglobal.was.dir=C:/WebSphere/AppServer
```

Build targets

The master build file contains a number of build targets; these are our four standard targets that we define in all our build files, and a number of project-wide targets that are unique to the master build file.

Initialization target

The first target we describe is the `init` target. All other targets in the build file depend upon this target.

In the `init` target we execute the `tstamp` task to set up properties that include timestamp information. These properties are available throughout the whole build. We also write out a message indicating that the build is starting. The XML for the init target is shown in Figure 9-21.

```
<target name="init">
  <tstamp/>
  <echo>Build of ${ant.project.name} started at ${TSTAMP} on ${TODAY}</echo>
</target>
```

Figure 9-21 Master build file init target

The `ant.project.name` property used in the starting message is another standard property supplied by Ant—it is set to the name of the project specified in the outer `project` tag in the current build file. When we execute the `init` target we see output similar to that shown in Figure 9-22.

```
D:\ITSO4AD\dev\src>ant init
Buildfile: build.xml

init:
     [echo] Build of itso4ad started at 1211 on May 23 2001

BUILD SUCCESSFUL

Total time: 0 seconds
```

Figure 9-22 Output from the master build file init target

Compilation and packaging targets

The `compile` and `package` targets in the master build file look very similar. They simply use the `ant` task to delegate to each of the subproject build files in turn, then output a message indicating that the target has completed. Figure 9-23 shows the XML that describes these targets.

```
<target name="compile" depends="init">
  <echo>Compiling ${ant.project.name}</echo>
  <ant dir="common" target="compile"/>
  <ant dir="ejb" target="compile"/>
  <ant dir="usecase" target="compile"/>
  <ant dir="client" target="compile"/>
  <ant dir="webapp" target="compile"/>
  <echo>Finished compiling ${ant.project.name}</echo>
</target>

<target name="package" depends="init">
  <echo>Packaging ${ant.project.name}</echo>
  <ant dir="common" target="package"/>
  <ant dir="ejb" target="package"/>
  <ant dir="usecase" target="package"/>
  <ant dir="client" target="package"/>
  <ant dir="webapp" target="package"/>
  <echo>Finished packaging ${ant.project.name}</echo>
</target>
```

Figure 9-23 Master build file compile and package targets

The `package` target in the master build file does not have to depend upon the `compile` target. This is because the `package` targets in all the subproject build files already depend upon `compile` in their own build files.

Cleanup targets

The last of our standard targets in the master build file is the `clean` target. This target delegates responsibility to the `clean` targets of the subproject build files, just as `compile` and `package` do. We also create two other cleanup-related targets in the master build file:

clean-document	Removes all the generated documentation from the `doc` directory
clean-all	Removes both compiled and packaged code and the generated documentation

The XML for the three cleanup targets is shown in Figure 9-24.

```
<target name="clean" depends="init">
  <echo>Cleaning ${ant.project.name}</echo>
  <ant dir="common" target="clean"/>
  <ant dir="ejb" target="clean"/>
  <ant dir="usecase" target="clean"/>
  <ant dir="client" target="clean"/>
  <ant dir="webapp" target="clean"/>
  <echo>Finished cleaning ${ant.project.name}</echo>
</target>

<target name="clean-document" depends="init">
  <echo>Cleaning documentation for ${ant.project.name}</echo>
  <delete dir="${global.javadoc.dir}"/>
  <echo>Finished cleaning documentation for ${ant.project.name}</echo>
</target>

<target name="clean-all" depends="init,clean,clean-document"/>
```

Figure 9-24 Master build file cleanup targets

Documentation target

We generate documentation from the source code using `javadoc` at the project level, rather than in each subproject. We do this because we want to have a single common set of documentation for all the code, with a single index and cross references that work between sub projects.

The `document` target in the master build file uses the built in `javadoc` task provided by Ant. This task has attributes that map to options of the `javadoc` tool provided by the Java SDK described in “javadoc options” on page 195. We obtain the values for many attributes from global properties—we added the properties shown in Figure 9-25 to the global properties file.

```
#
# javadoc settings
#
global.javadoc.public=
global.javadoc.protected=
global.javadoc.package=
global.javadoc.private=on
global.javadoc.use=on
global.javadoc.windowtitle=PiggyBank Documentation
global.javadoc.doctitle=PiggyBank Documentation
```

Figure 9-25 Javadoc properties in the global properties file

We also use a path to define the list of directories containing source code to be scanned by the `javadoc` task. The path is defined near the beginning of the master build file, after the XML that defines the global properties. The XML is shown in Figure 9-26. The path may come in useful if we have to add any new targets that must process all of the source files.

```
<!-- Set up local properties and paths-->
 <path id="itso4ad.source.path">
   <pathelement path="${basedir}/common"/>
   <pathelement path="${basedir}/ejb"/>
   <pathelement path="${basedir}/usecase"/>
   <pathelement path="${basedir}/client"/>
   <pathelement path="${basedir}/webapp"/>
 </path>
```

Figure 9-26 Defining a path containing all source files

The XML for this target is shown in Figure 9-27. Note how we use the `TODAY` property set by the `tstamp` task to include the date at the bottom of each page generated. The output generated when we use Ant to generate the `javadoc` documentation is shown in Figure 9-28.

```
<target name="document" depends="init">
  <echo>Generating documentation for ${ant.project.name}</echo>
  <mkdir dir="${global.javadoc.dir}"/>
  <javadoc sourcepathref="itso4ad.source.path"
           packagenames="itso.*"
           classpath="${global.was.dir}/lib/j2ee.jar;${global.was.dir}/lib/ras.jar"
           destdir="${global.javadoc.dir}"
           public="${global.javadoc.public}"
           protected="${global.javadoc.protected}"
           package="${global.javadoc.package}"
           private="${global.javadoc.private}"
           use="${global.javadoc.use}"
           windowtitle="${global.javadoc.windowtitle}"
           doctitle="${global.javadoc.doctitle}"
           bottom="&lt;i&gt;Generated on ${TODAY}&lt;/i&gt;"/>
  <echo>Finished generating documentation for ${ant.project.name}</echo>
</target>
```

Figure 9-27 Master build file document target

```
Buildfile: build.xml

init:
     [echo] Build of itso4ad started at 1356 on July 16 2001

document:
     [echo] Generating documentation for itso4ad
  [javadoc] Generating Javadoc
  [javadoc] Javadoc execution
  [javadoc] Loading source files for package itso.was4ad.data...
  [javadoc] Loading source files for package itso.was4ad.exception...
  [javadoc] Loading source files for package itso.was4ad.helpers...
  [javadoc] Loading source files for package itso.was4ad.ejb.account...
  [javadoc] Loading source files for package itso.was4ad.ejb.customer...
  [javadoc] Loading source files for package itso.was4ad.usecase...
  [javadoc] Loading source files for package itso.was4ad.client.swing...
  [javadoc] Loading source files for package itso.was4ad.webapp.command...
  [javadoc] Loading source files for package itso.was4ad.webapp.controller...
  [javadoc] Loading source files for package itso.was4ad.webapp.view...
  [javadoc] Constructing Javadoc information...
  [javadoc] Building tree for all the packages and classes...
  [javadoc] Building index for all the packages and classes...
  [javadoc] Building index for all classes...
     [echo] Finished generating documentation for itso4ad

BUILD SUCCESSFUL

Total time: 13 seconds
```

Figure 9-28 Output from the Ant document target

Building the common code

The Ant build file that describes how to build the common code is located at `src\common\build.xml`. We start with a basic skeleton that we use for all our subproject build files (Figure 9-29).

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

<project name="common" default="package">

  <!-- Set up global properties -->
  <property name="global.dev.dir" value="${basedir}/../.."/>
  <property file="${user.home}/override.properties"/>
  <property file="../global.properties"/>

  <!-- If we were invoked by the master file TSTAMP will be set already -->
  <target name="init" unless="TSTAMP">
    <tstamp/>
    <echo>
      Build of ${ant.project.name} started at ${TSTAMP} on ${TODAY}
    </echo>
  </target>

  <target name="compile" depends="init">
    <echo>Compiling ${ant.project.name}</echo>
    <echo>Finished compiling ${ant.project.name}</echo>
  </target>

  <target name="package" depends="init,compile">
    <echo>Packaging ${ant.project.name}</echo>
    <echo>Finished packaging ${ant.project.name}</echo>
  </target>

<target name="clean" depends="init">
    <echo>Cleaning ${ant.project.name}</echo>
    <echo>Finished cleaning ${ant.project.name}</echo>
  </target>

</project>
```

Figure 9-29 Skeleton build file for subprojects

The only text in this skeleton that is specific to a subproject is the `name` attribute of the outer `project` tag.

Compiling the common code

Before we compile the code we first ensure that the destination directory exists. We achieve this using the `mkdir` task. We then use the `javac` task to compile the code, specifying the following attributes:

srcdir	The location of the Java source files
destdir	The directory in which to write generated .class files
classpathref	A reference to the class path to use during compilation
debug	Whether to generate debugging information in .class files
optimize	Whether to turn on compiler optimization
deprecation	Whether to output deprecation warnings

The complete XML for the `compile` task is shown in Figure 9-30.

```
<target name="compile" depends="init">
   <echo>Compiling ${ant.project.name}</echo>
   <mkdir dir="${common.build.dir}"/>
   <javac srcdir="${basedir}"
          destdir="${common.build.dir}"
          classpathref="common.classpath"
          debug="${global.javac.debug}"
          optimize="${global.javac.optimize}"
          deprecation="${global.javac.deprecation}"
   />
   <echo>Finished compiling ${ant.project.name}</echo>
 </target>
```

Figure 9-30 Common code compile target

The values for the `debug`, `optimize` and `deprecation` attributes are obtained from global properties. Figure 9-31 shows the information we add to the `global.properties` file to support this.

```
#
# javac settings
#
global.javac.debug=on
global.javac.optimize=off
global.javac.deprecation=on
```

Figure 9-31 Compiler settings in global properties file

The destination directory and class path used during the compile are specific to this subproject, but may be referred to many times in the build file, so we define these by adding the XML in Figure 9-32 near the start of the build file, after the XML defining the global properties.

```
<!-- Set up local properties and paths-->
<property name="common.build.dir" value="${global.build.dir}/common"/>
<path id="common.classpath">
  <pathelement location="${global.was.dir}/lib/j2ee.jar"/>
  <pathelement location="${global.was.dir}/lib/ras.jar"/>
</path>
```

Figure 9-32 Setting up local properties and class path for the common code

We can test the compilation of the common code in isolation by changing to the `src\common` directory and issuing the command:

```
ant compile
```

This generates output similar to that shown in Figure 9-33.

```
D:\ITSO4AD\dev\src\common>ant compile
Buildfile: build.xml

init:
     [echo] Build of common started at 1644 on May 23 2001

compile:
     [echo] Compiling common
    [mkdir] Created dir: D:\ITSO4AD\dev\src\common\..\..\build\common
    [javac] Compiling 12 source files to
                       D:\ITSO4AD\dev\src\common\..\..\build\common
     [echo] Finished compiling common

BUILD SUCCESSFUL

Total time: 3 seconds
```

Figure 9-33 Output from compiling the common code

Ant is able to determine whether compiles have to be performed or not by comparing the timestamps on the generated .class files with those on the Java source files. If we immediately repeat the command to compile the common code we get a different output (Figure 9-34).

```
D:\ITSO4AD\dev\src\common>ant compile
Buildfile: build.xml

init:
     [echo] Build of common started at 1646 on May 23 2001

compile:
     [echo] Compiling common
     [echo] Finished compiling common

BUILD SUCCESSFUL

Total time: 0 seconds
```

Figure 9-34 Compilation output when no source files are out of date

Packaging the common code

Packaging the common code involves creating a JAR file containing the compiled .class files generated by the `compile` target.

The JAR file must be created in the modules directory. We use the `mkdir` task again to make sure the modules directory exists, and then the `jar` task to create the JAR file. We specify the manifest information to include using the `manifest` attribute. This is illustrated in Figure 9-35.

```
<target name="package" depends="init,compile">
  <echo>Packaging ${ant.project.name}</echo>
  <mkdir dir="${global.module.dir}"/>
  <jar jarfile="${common.jar.file}"
       basedir="${common.build.dir}"
       manifest="META-INF/MANIFEST.MF"/>
  <echo>Finished packaging ${ant.project.name}</echo>
</target>
```

Figure 9-35 Common code package target

This task uses a new local property, `client.jar.file`, in the build file to define the name of the JAR file to create. We updated the XML that defines the local properties as shown in Figure 9-36 to include the new property.

```
<!-- Set up local properties and paths-->
<property name="common.build.dir" value="${global.build.dir}/common"/>
<property name="common.jar.file"
          value="${global.module.dir}/piggybank-common.jar"/>
<path id="common.classpath">
  <pathelement location="${global.was.dir}/lib/j2ee.jar"/>
  <pathelement location="${global.was.dir}/lib/ras.jar"/>
  <pathelement location="${global.log4j.lib.dir}/log4j.jar"/>
</path>
```

Figure 9-36 Updated local properties and path for the common build file

When we use Ant to execute the `package` target we get output similar to that shown in Figure 9-37.

```
D:\ITSO4AD\dev\src\common>ant package
Buildfile: build.xml

init:
     [echo] Build of common started at 1701 on May 23 2001

compile:
     [echo] Compiling common
     [echo] Finished compiling common

package:
     [echo] Packaging common
    [mkdir] Created dir: D:\ITSO4AD\dev\src\common\..\..\modules
      [jar] Building jar:
                D:\ITSO4AD\dev\src\common\..\..\modules\piggybank-common.jar
     [echo] Finished packaging common

BUILD SUCCESSFUL

Total time: 1 second
```

Figure 9-37 Output from packaging the common code

Because `package` depends on `compile` Ant first makes sure that all the compiled code is up to date before creating the JAR file.

Cleaning up common code files

The `clean` target in the common build file removes the directory containing the compiled .class files generated by the compiler and the packaged JAR file. The XML is shown in Figure 9-38.

```
<target name="clean" depends="init">
  <echo>Cleaning ${ant.project.name}</echo>
  <delete dir="${common.build.dir}"/>
  <delete file="${common.jar.file}"/>
  <echo>Finished cleaning ${ant.project.name}</echo>
</target>
```

Figure 9-38 Common code clean target

Building the EJBs

The Ant build file that describes how to build the EJBs is located at `src\ejb\build.xml`. We start again with the basic skeleton and add in the local properties and path that we need (Figure 9-39).

```
<!-- Set up local properties and paths-->
<property name="ejb.build.dir" value="${global.build.dir}/ejb"/>
<property name="ejb.jar.name" value="piggybank-ejb.jar"/>
<property name="ejb.jar.file" value="${global.module.dir}/${ejb.jar.name}"/>
<path id="ejb.classpath">
  <pathelement location="${global.was.dir}/lib/j2ee.jar"/>
  <pathelement path="${global.build.dir}/common"/>
</path>
```

Figure 9-39 Local properties and path for the EJB code

The main difference from the build file for the common code is that we have added the common code to the class path, because the EJB code uses classes from the common code.

Compiling the EJBs

The `compile` target for the EJBs is nearly identical to that for the common code. The only difference is in the property names that define the class path to use and the destination directory for the compiled .class files. The XML is shown in Figure 9-40.

```
<target name="compile" depends="init">
  <echo>Compiling ${ant.project.name}</echo>
  <mkdir dir="${ejb.build.dir}"/>
  <javac srcdir="${basedir}"
         destdir="${ejb.build.dir}"
         classpathref="ejb.classpath"
         debug="${global.javac.debug}"
         optimize="${global.javac.optimize}"
         deprecation="${global.javac.deprecation}"
  />
  <echo>Finished compiling ${ant.project.name}</echo>
</target>
```

Figure 9-40 EJB compile target

Packaging the EJBs

We also use the built in Ant `jar` task to package the EJB JAR file. The XML for the `package` target is shown in Figure 9-41.

```
<target name="package" depends="init,compile">
  <echo>Packaging ${ant.project.name}</echo>
  <mkdir dir="${global.module.dir}"/>
  <jar jarfile="${ejb.jar.file}"
       manifest="META-INF/MANIFEST.MF"
  >
    <fileset dir="${ejb.build.dir}"/>
    <fileset dir="${basedir}">
      <include name="META-INF/*"/>
      <exclude name="META-INF/MANIFEST.MF"/>
    </fileset>
  </jar>
  <echo>Finished packaging ${ant.project.name}</echo>
</target>
```

Figure 9-41 EJB package target

We specify the output file name and manifest to use with attributes, and the files to include in the archive using nested `fileset` elements. The first nested element adds the compiled .class files to the archive. The second nested element pulls in the deployment descriptor files, but excludes the manifest file, which was specified earlier.

Figure 9-42 shows the output from Ant when it is used to package the EJBs.

```
D:\ITSO4AD\dev\src\ejb>ant package
Buildfile: build.xml

init:
     [echo] Build of ejb started at 1240 on June 6 2001

compile:
     [echo] Compiling ejb
     [echo] Finished compiling ejb

package:
     [echo] Packaging ejb
      [jar] Building jar:
            D:\ITSO4AD\dev\src\ejb\..\..\modules\piggybank-ejb.jar

     [echo] Finished packaging ejb

BUILD SUCCESSFUL

Total time: 1 second
```

Figure 9-42 Output from packaging the EJBs

Packaging EJBs and generating deployed code

The previous section described how to generate an *undeployed* EJB JAR file. When an undeployed EJB JAR or an EAR containing an undeployed EJB JAR file is installed into WebSphere the application server must deploy the EJB code before the EJBs can be installed. This deployment process involves examining the EJB code and deployment information and generating and compiling WebSphere-specific code that links the EJBs with the WebSphere EJB container implementation.

While it is quite acceptable to deliver EJBs and generate code in this manner, you may prefer to generate the deployed code earlier in the cycle, at the point where we create the EJB JAR file. The main benefits of this approach are:

- Deployment issues caused by problems in the code such as non-conformance with the EJB specification are highlighted when the code is built, rather than when it is deployed.
- The code generation is a relatively lengthy process—we can speed up the development code and unit test cycle if we only generate code when absolutely necessary, rather than every time we redeploy an EAR file that contains EJBs.

Figure 9-43 shows the XML we developed to create the EJB JAR file and generated deployed code for WebSphere.

```
<target name="package" depends="init,compile">
   <echo>Packaging ${ant.project.name}</echo>
   <uptodate property="ejb.deploy.uptodate">
     <srcfiles dir= "${ejb.build.dir}"/>
     <srcfiles dir= "${basedir}/META-INF"/>
     <mapper type="merge" to="${ejb.jar.file}"/>
   </uptodate>
   <mkdir dir="${global.module.dir}"/>
   <jar jarfile="${ejb.jar.file}"
        manifest="META-INF/MANIFEST.MF"
   >
     <fileset dir="${ejb.build.dir}"/>
     <fileset dir="${basedir}">
       <include name="META-INF/*"/>
       <exclude name="META-INF/MANIFEST.MF"/>
     </fileset>
   </jar>
   <antcall target="deploy"/>
 </target>

 <target name="deploy" unless="ejb.deploy.uptodate">
   <echo>Deploying EJB JAR file</echo>
   <exec executable="${global.was.dir}/bin/ejbdeploy.bat">
     <arg value="${ejb.jar.file}"/>
     <arg value="${global.temp.dir}"/>
     <arg value="${global.temp.dir}/${ejb.jar.name}"/>
     <arg value="-quiet"/>
   </exec>
   <move file="${global.temp.dir}/${ejb.jar.name}"
         tofile="${ejb.jar.file}"
   />
   <echo>Finished deploying EJB JAR file</echo>
 </target>
```

Figure 9-43 Package and deploy targets for generating a deployed EJB JAR file

We made two new additions to the original package target. The first of these uses the Ant built-in `uptodate` task to determine whether or not we need to regenerate the deployed EJB code.

This task sets a property depending on the time stamps on files—in our case we compare the dates on the contents of the EJB build tree and the meta-data in the source tree with the date of the current EJB JAR file, before we run the `jar` task to create it. We have to do this because although the `jar` task is able to determine whether or not the target file is up to date by examining the source files, the `exec` task we use to generate the deployed code is not.

The second addition to the `package` target uses the built-in `antcall` task to invoke a new `deploy` target in our build file. This new target executes only if the EJB JAR file was not already up to date, making the decision based upon the property set by the `uptodate` task in the `package` target.

The `deploy` target runs the WebSphere `ejbdeploy` tool on the undeployed JAR file, generating a new JAR file containing the deployed code in a temporary directory. It then moves the generated file from the temporary directory into the `modules` directory.

The output from this version of the package target is shown in Figure 9-44.

```
Buildfile: build.xml

init:
     [echo] Build of ejb started at 1712 on July 15 2001

compile:
     [echo] Compiling ejb
     [echo] Finished compiling ejb

package:
     [echo] Packaging ejb
      [jar] Building jar:
                     D:\itso4ad\dev\src\ejb\..\..\modules\piggybank-ejb.jar


deploy:
     [echo] Deploying EJB JAR file
     [exec] 0 Errors, 0 Warnings, 0 Informational Messages
     [move] Moving 1 files to D:\itso4ad\dev\src\..\modules
     [echo] Finished deploying EJB JAR file

BUILD SUCCESSFUL

Total time: 1 minute 0 seconds
```

Figure 9-44 Output from packaging the EJBs and generating deployed code

Cleaning up EJB files

The `clean` target in the EJB build file removes the directory containing the compiled .class files generated by the compiler and the packaged JAR file. The XML is shown in Figure 9-45.

```
<target name="clean" depends="init">
  <echo>Cleaning ${ant.project.name}</echo>
  <delete dir="${ejb.build.dir}"/>
  <delete file="${ejb.jar.file}"/>
  <echo>Finished cleaning ${ant.project.name}</echo>
</target>
```

Figure 9-45 EJB clean target

Building the use cases

The Ant build file that describes how to build the use case code is located at `src\usecase\build.xml`. It compiles and packages the use case code into the `piggybank-usecase.jar` archive, which is stored with the other archives in the `modules` directory.

The build file is almost identical to that described for the common code in "Building the common code" on page 210, except that the use case code requires the common and EJB build directories on the class path in order to compile. For this reason we do not describe it further here.

Building the standalone client application

The Ant build file that describes how to build the client code is located at `src\client\build.xml`. We start again with the basic skeleton and add in the local properties and path that we need (Figure 9-46).

```
<!-- Set up local properties and paths-->
<property name="client.build.dir" value="${global.build.dir}/client"/>
<property name="client.jar.file"
          value="${global.module.dir}/piggybank-client.jar"/>
<path id="client.classpath">
  <pathelement location="${global.was.dir}/lib/j2ee.jar"/>
  <pathelement path="${global.build.dir}/common"/>
  <pathelement path="${global.build.dir}/usecase"/>
</path>
```

Figure 9-46 Local properties and path for the client code

Compiling the client code

The `compile` target for the client code is nearly identical to that for the common code. The only difference is in the property names that define the class path to use and the destination directory for the compiled .class files. The XML is shown in Figure 9-47.

```
<target name="compile" depends="init">
  <echo>Compiling ${ant.project.name}</echo>
  <mkdir dir="${client.build.dir}"/>
  <javac srcdir="${basedir}"
         destdir="${client.build.dir}"
         classpathref="client.classpath"
         debug="${global.javac.debug}"
         optimize="${global.javac.optimize}"
         deprecation="${global.javac.deprecation}"
  />
  <echo>Finished compiling ${ant.project.name}</echo>
</target>
```

Figure 9-47 Client compile target

Packaging the client code

We use the built in `jar` task to create the JAR file for the standalone client application. Figure 9-48 shows how we use two nested `fileset` elements inside the `JAR` element to include the compiled .class files from the build tree, and the client meta-data from the source tree. The second nested element excludes the manifest file which we specify explicitly in an attribute of the `jar` task.

```
<target name="package" depends="init,compile">
  <echo>Packaging ${ant.project.name}</echo>
  <mkdir dir="${global.module.dir}"/>
  <jar jarfile="${client.jar.file}"
       manifest="META-INF/MANIFEST.MF"
  >
    <fileset dir="${client.build.dir}"/>
    <fileset dir="${basedir}">
      <include name="META-INF/*"/>
      <exclude name="META-INF/MANIFEST.MF"/>
    </fileset>
  </jar>
  <echo>Finished packaging ${ant.project.name}</echo>
</target>
```

Figure 9-48 Client package target

Cleaning up client files

The `clean` target in the client build file removes the directory containing the compiled .class files generated by the compiler and the packaged JAR file. The XML is shown in Figure 9-49.

```
<target name="clean" depends="init">
  <echo>Cleaning ${ant.project.name}</echo>
  <delete dir="${client.build.dir}"/>
  <delete file="${client.jar.file}"/>
  <echo>Finished cleaning ${ant.project.name}</echo>
</target>
```

Figure 9-49 Client clean target

Building the Web application

The Ant build file that describes how to build the Web application is located at `src\webapp\build.xml`. We start again with the basic skeleton and add in the local properties and path that we need (Figure 9-50).

```
<!-- Set up local properties and paths-->
<property name="webapp.build.dir" value="${global.build.dir}/webapp"/>
<property name="webapp.war.file"
          value="${global.module.dir}/piggybank-webapp.war"/>
<path id="webapp.classpath">
  <pathelement location="${global.was.dir}/lib/j2ee.jar"/>
  <pathelement path="${global.build.dir}/common"/>
  <pathelement path="${global.build.dir}/usecase"/>
</path>
```

Figure 9-50 Local properties and path for the Web application code

Compiling the Web application code

The `compile` target for the Web application code is nearly identical to that for the common code. The only difference is in the property names that define the class path to use and the destination directory for the compiled .class files. The XML is shown in Figure 9-51.

```
<target name="compile" depends="init">
  <echo>Compiling ${ant.project.name}</echo>
  <mkdir dir="${webapp.build.dir}"/>
  <javac srcdir="${basedir}"
         destdir="${webapp.build.dir}"
         classpathref="webapp.classpath"
         debug="${global.javac.debug}"
         optimize="${global.javac.optimize}"
         deprecation="${global.javac.deprecation}"
  />
  <echo>Finished compiling ${ant.project.name}</echo>
</target>
```

Figure 9-51 Web application compile target

Packaging the Web application

Ant has a built in `war` task that can create packaged WAR files. We use this task to create the WAR file for the Web application (Figure 9-52).

```
<target name="package" depends="init,compile">
  <echo>Packaging ${ant.project.name}</echo>
  <mkdir dir="${global.module.dir}"/>
  <war warfile="${webapp.war.file}"
       webxml="WEB-INF/web.xml"
       basedir="web"
       manifest="META-INF/MANIFEST.MF"
  >
    <classes dir="${webapp.build.dir}"/>
    <webinf dir="WEB-INF">
      <exclude name="web.xml"/>
    </webinf>
  </war>
  <echo>Finished packaging ${ant.project.name}</echo>
</target>
```

Figure 9-52 Web application package target

We use the following attributes and nested elements with the `war` task:

warfile	The name of the generated archive file
webxml	The location of the Web application deployment descriptor
basedir	The base location of the Web content that is to be included in the archive
manifest	The file containing the manifest information to use

classes This nested element that specifies the files that should be included in the `WEB-INF/classes` directory in the archive. We include all the .class files compiled into the `build\webapp` directory.

webinf This nested element specifies other files to be included in the `WEB-INF` directory in the archive. We include all of the files in the `WEB-INF` directory in the source tree, excluding the `web.xml` which is specified as an attribute of the `war` task.

Figure 9-53 shows the output generated when Ant is used to package the Web application WAR file.

```
D:\ITSO4AD\dev\src\webapp>ant package
Buildfile: build.xml

init:
     [echo] Build of webapp started at 1615 on May 24 2001

compile:
     [echo] Compiling webapp
     [echo] Finished compiling webapp

package:
     [echo] Packaging webapp
      [war] Building war:
                D:\ITSO4AD\dev\src\webapp\..\..\modules\piggybank-webapp.war
     [echo] Finished packaging webapp

BUILD SUCCESSFUL
Total time: 1 second
```

Figure 9-53 Output from packaging the Web application

Cleaning up Web application files

The `clean` target in the Web application build file removes the directory containing the compiled .class files generated by the compiler and the packaged war file. The XML is shown in Figure 9-54.

```
<target name="clean" depends="init">
  <echo>Cleaning ${ant.project.name}</echo>
  <delete dir="${webapp.build.dir}"/>
  <delete file="${webapp.war.file}"/>
  <echo>Finished cleaning ${ant.project.name}</echo>
</target>
```

Figure 9-54 Web application clean target

Further automation opportunities using Ant

So far we have seen how we can use the basic features of Ant to build the separate application modules that make up the PiggyBank application. Depending on your environment you may find it useful to build upon the ideas presented here and create build files for Ant that can be used to further enhance your development environment. Some ideas you may want to consider involve using Ant to:

- Package application modules into J2EE enterprise application archive (EAR) files
- Build and rebuild specific application release versions by extracting versioned application source files from a software configuration management tool such as Rational ClearCase
- Extract and build source files from VisualAge for Java using the VisualAge tasks supplied in the package of optional Ant tasks
- Deploy application modules into WebSphere ready for testing
- Initiate automated testing suites such as JUnit (see “Automating unit testing using JUnit” on page 517).
- Insert build version information into source files before building the application

If you examine the build files that come with the sample application code you can see some of these ideas put into practice. See “Using the Web material” on page 558.

To gain a better understanding of the standard features available with Ant consult the Ant documentation that is available on the Web and in the Ant distribution. If the standard features do not suit your purposes, remember that Ant provides a mechanism for you to implement your own tasks in Java.

Automatic builds

A common practice in many development environments is the use of daily builds. These automatic builds are usually initiated in the early hours of the morning by a scheduling tool such as `cron`, which is standard on UNIX systems. Similar tools are available for Windows environments. The daily builds usually attempts to build a complete system, based upon the latest checked-in versions of the application source files.

A daily build benefits a development team by automatically highlighting compile-time issues in a timely manner. It also assists developers by providing a baseline against which to develop and unit test their code. Developers can refresh their individual development environment from the latest daily build before starting work on a development task. Compilation times are improved because each developer has to build only the components he or she is currently working on. Integration issues are reduced because developers work against the most recent version of the application.

Using Ant you can extend the daily build concept to perform additional tasks as part of the automatic nightly process. For example, if a nightly build completes successfully you could then have Ant automatically deploy the latest build into a test environment and execute all of your unit test cases against it, then e-mail the results to the appropriate team members.

Ant is especially well suited if you have a heterogeneous environment, perhaps with Windows-based developer desktops and UNIX for test and deployment.

Working with meta-data

During the earlier discussions in this chapter we mention meta-data files a number of times. This section describes the various meta-data files involved in the development and deployment of a WebSphere application, and describes ways in which the WebSphere tools can be used to create and modify the meta-data files, while still managing the individual files in the source tree, which is especially useful in an environment using a software configuration management (SCM) tool.

This section also describes how we use Ant build files to create an enterprise archive (EAR) file, complete with WebSphere binding information for a specific environment, that can be installed directly into WebSphere without user intervention.

Meta-data in WebSphere

We have to consider three categories of meta-data files when developing and building WebSphere applications. The three categories are:

- J2EE deployment descriptors
- WebSphere deployment information
- Java archive (JAR) manifest information

J2EE deployment descriptors

The J2EE specification defines a number of deployment descriptors. These are files that contain XML that describes the components supplied in an application module and define the resources they require in order to be deployed into a container.

WebSphere Application Server provides tools that process these deployment descriptors and map the requirements of an application module to services provided by the WebSphere containers. Table 9-1 lists the four XML deployment descriptor files defined in Version 1.2 of the J2EE specification.

Table 9-1 J2EE deployment descriptors

Module type	Descriptor file name
EJB	`ejb-jar.xml`
Web application	`web.xml`
Client application	`client-application.xml`
Enterprise application	`application.xml`

WebSphere deployment information

When you use WebSphere tools such as the Application Assembly Tool (AAT) to assemble and deploy J2EE modules, the tools store information specific to the WebSphere environment in separate files in the J2EE archive files. The information is stored separately from the J2EE deployment information so that tools from other vendors are still able to process the standard descriptors.

There are two classes of WebSphere-specific information:

- Extensions describe additional information specific to the WebSphere J2EE implementation, but applicable to every WebSphere installation, for example EJB methods that should be considered read-only.
- Bindings describe how components should behave in a particular WebSphere installation, for example the name of the DataSource or the global JNDI name an EJB should use.

WebSphere-specific meta-data files are listed in Table 9-2. The EJB schema and map files are described in more detail in “Customizing CMP persistence mapping” on page 420.

Table 9-2 WebSphere deployment meta-data files

Module type	File name	Purpose
EJB	`ibm-ejb-jar-bnd.xmi`	Binding information for EJBs including JNDI names and data source properties
	`ibm-ejb-jar-ext.xmi`	Describes security and transaction information for EJBs, including defining read-only methods
	`ibm-ejb-access-bean.xmi`	Defines EJB access beans
	`Map.mapxmi`	Defines mappings between CMP EJB fields and the DB schema
	`Schema.dbxmi`	Describes the database schema used by CMP EJBs
Web application	`ibm-web-bnd.xmi`	Binding information for the Web application, including WebSphere virtual host name
	`ibm-web-ext.xmi`	Settings for WebSphere extensions, for example default error page and whether to serve servlets by class name
Client application	`ibm-application-client-bnd.xmi`	J2EE client binding information, for example mapping local EJB references to global JNDI names
Enterprise application	`ibm-application-bnd.xmi`	Binding information for the enterprise application
	`ibm-application-ext.xmi`	WebSphere extension information for the enterprise application

Manifest information

When we build our application modules we specify information that we want to include in the manifest file included in the Java archive (JAR) file. All of the PiggyBank modules use the JAR manifest to specify the class path to search in order to find Java classes that the code in the module needs in order to deploy and run. The contents of the manifest file for the Web application module are shown in Figure 9-55.

```
Manifest-Version: 1.0
Class-Path: piggybank-common.jar piggybank-usecase.jar
```

Figure 9-55 Manifest information for the PiggyBank Web application module

The `Class-Path` entry in the manifest indicates that a class loader should search the common and use case JAR files for classes that the Web application module needs. The locations are specified relative to the location from which the JAR that includes the manifest was loaded. When the PiggyBank application is packaged into an enterprise archive (EAR) file our application modules are all placed in the same location, the base directory of the archive file.

Creating and editing meta-data files

Because the J2EE deployment descriptors are well documented, and the XML document type definitions (DTDs) are freely available, it is certainly possible to create and manage your J2EE deployment descriptors using only a simple text editor. This is not an easy or enjoyable task however, even with an XML-aware editor to help you match up tags and highlight syntax errors.

All editions of WebSphere Application Server include the Application Assembly Tool (AAT), which provides a GUI interface that greatly simplifies the task of creating and assembling modules and the deployment descriptors that describe them. It also is able to manage the WebSphere specific binding and extension files. The AAT is described in more detail in Chapter 15, “Assembling the application” on page 389.

The problem in an iterative development cycle is that you do not want to go through the laborious and potentially error-prone process of assembling your application from scratch before you can test a new version of your code. This is especially true in an environment using daily or more frequent builds. Most code changes do not require any modification to the deployment descriptors. Changes that do—the addition of a new EJB or servlet, for example, are relatively rare.

The solution is to use AAT to create and edit your deployment descriptors, then extract and save the descriptors in the source tree. When we want to introduce new code we can rebuild our modules using the saved descriptors. If we have to edit the descriptors we simply build the module using the old descriptors, load it into AAT for editing, then extract the new descriptors from the module file saved by the tool.

This is the mechanism we used to create all of the deployment descriptors used by the PiggyBank application described in this chapter.

Extracting deployment descriptors

All J2EE archives can be manipulated using the `jar` tool included with the Java 2 SDK. To create deployment descriptors in our source tree from a module saved using AAT, we simply move to the appropriate directory and extract the files we need. This is illustrated in Figure 9-56, which shows how the various deployment descriptors can be extracted from the `piggybank-webapp.war` archive.

```
D:\ITSO4AD\dev\src\webapp>jar xvf
    D:\ITSO4AD\dev\modules\piggybank-webapp.war
    META-INF WEB-INF/web.xml WEB-INF/ibm-web-bnd.xmi WEB-INF/ibm-web-ext.xmi
  created: META-INF/
extracted: META-INF/MANIFEST.MF
extracted: WEB-INF/ibm-web-bnd.xmi
extracted: WEB-INF/ibm-web-ext.xmi
extracted: WEB-INF/web.xml
```

Figure 9-56 Extracting meta-data files from a Web application archive

Automation using Ant

We decided to extend our existing Ant build files to allow us to manage our EAR file and all of the meta-data for the PiggyBank application. We created a new `ear` directory in the source tree in which to store the meta-data for the enterprise application, and the Ant build file that we use to manage it.

There are three targets of interest in the build file:

package	Creates the EAR file
edit-ear	Helps us edit the deployment descriptors
install	Installs the packaged EAR file into WebSphere Application Server, Single Server Edition, ready for testing

Packaging the EAR file

For simplicity we create a single EAR file which contains all five modules that make up the PiggyBank application. In a more realistic scenario we would probably create two separate files:

- The first would be installed into WebSphere and contain the common code, use case code, EJBs and the Web application.
- The second EAR would be distributed to client systems and would contain only the common code, use case code, EJB client files and the code for the standalone application client.

The XML that describes the package target is shown in Figure 9-57. The EAR file is created using the standard Ant `jar` task, pulling in the five modules and the EAR meta-data files.

```
<target name="package" depends="init">
  <echo>Packaging ${ant.project.name}</echo>
  <jar jarfile="${ear.file}">
    <fileset dir="${global.module.dir}">
      <include name="piggybank-common.jar"/>
      <include name="piggybank-ejb.jar"/>
      <include name="piggybank-usecase.jar"/>
      <include name="piggybank-client.jar"/>
      <include name="piggybank-webapp.war"/>
    </fileset>
    <fileset dir="${basedir}">
      <include name="META-INF/*"/>
    </fileset>
  </jar>
  <echo>Finished packaging ${ant.project.name}</echo>
</target>
```

Figure 9-57 EAR build file package target

Editing the EAR file

The `edit-ear` target is used to help us edit the meta-data information for all of the modules included in the enterprise archive, as well as the archive itself. The XML that defines the target is shown in Figure 9-58.

This target starts AAT using the Ant *exec* task so we can edit the descriptors. Unfortunately, AAT does not recognize file names supplied as command-line parameters, so the assembly tool prompts us to open the appropriate temporary file.

Once the assembly tool has terminated, Ant takes the modified EAR file and extracts the meta-data files for all of the modules out of the EAR into a temporary directory and puts them back into the appropriate locations in the source tree. The Ant copy task only copies a file over another file if the time stamp on the destination file is older than that on the source file. If the EAR file was not modified by AAT the meta-data files are not altered and Ant does not copy them.

```
<target name="edit-ear" depends="init">
   <property name="ear.temp.dir" value="${global.temp.dir}/edit-ear.${DSTAMP}.${TSTAMP}"/>
   <mkdir dir="${ear.temp.dir}"/>
   <echo>
     Starting the WebSphere Application Assembly Tool (AAT)
     When the tool starts, open and edit the file
     ${global.module.dir}/piggybank.ear
     Wnen you have finished editing the file close AAT and the
     deployment descriptors will be copied back into the source tree.
   </echo>
   <exec executable="assembly"/>

   <echo>Assembly tool has terminated - processing changed deployment descriptors</echo>

   <!-- EAR descriptors -->
   <unjar src="${global.module.dir}/piggybank.ear"
          dest="${ear.temp.dir}"
   />
   <copy todir="META-INF">
     <fileset dir="${ear.temp.dir}/META-INF">
       <exclude name="MANIFEST.MF"/>
     </fileset>
   </copy>

   <!-- EJB descriptors -->
   <mkdir dir="${ear.temp.dir}/ejb"/>
   <unjar src="${ear.temp.dir}/piggybank-ejb.jar"
          dest="${ear.temp.dir}/ejb"
   />
   <copy todir="../ejb/META-INF">
     <fileset dir="${ear.temp.dir}/ejb/META-INF"/>
   </copy>

   <!-- Client descriptors -->
   <mkdir dir="${ear.temp.dir}/client"/>
   <unjar src="${ear.temp.dir}/piggybank-client.jar"
          dest="${ear.temp.dir}/client"
   />
   <copy todir="../client/META-INF">
     <fileset dir="${ear.temp.dir}/client/META-INF"/>
   </copy>

   <!-- webapp descriptors -->
   <mkdir dir="${ear.temp.dir}/webapp"/>
   <unjar src="${ear.temp.dir}/piggybank-webapp.war"
          dest="${ear.temp.dir}/webapp"
   />
   <copy todir="../webapp/META-INF">
     <fileset dir="${ear.temp.dir}/webapp/META-INF"/>
   </copy>
   <copy todir="../webapp/WEB-INF">
     <fileset dir="${ear.temp.dir}/webapp/WEB-INF">
       <exclude name="classes"/>
       <exclude name="lib"/>
     </fileset>
   </copy>

   <delete dir="${ear.temp.dir}"/>
   <echo>EAR meta-data has been copied</echo>
 </target>
```

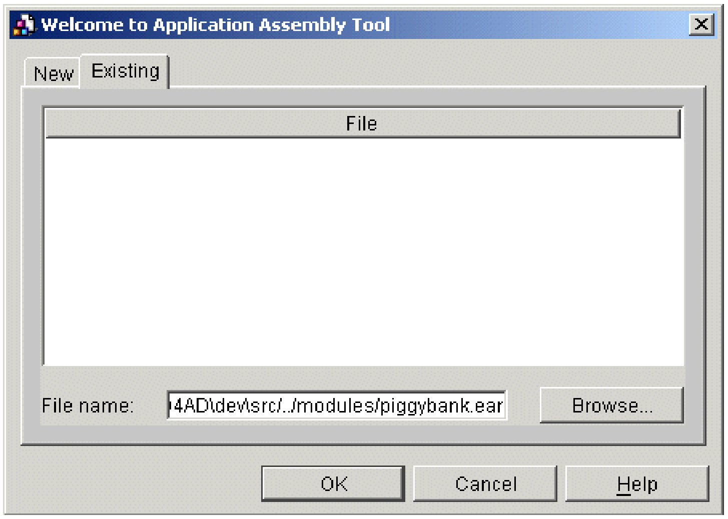
Figure 9-58 EAR build file edit-ear target

Figure 9-59 shows the output from Ant as it launches AAT.

```
D:\ITSO4AD\dev\src\ear>ant edit-ear
Buildfile: build.xml

init:
     [echo] Build of itso4ad started at 1015 on June 7 2001

edit-ear:
    [mkdir] Created dir: D:\temp\edit-ear.20010607.1015
     [echo]
      Starting the WebSphere Application Assembly Tool (AAT)
      When the tool starts, open and edit the file
      D:\ITSO4AD\dev\src/../modules/piggybank.ear
      Wnen you have finished editing the file close AAT and the
      deployment descriptors will be copied back into the source tree.
```

Figure 9-59 Output from the edit-ear target as it launches the assembly tool

When AAT starts, we copy the name of the temporary EAR file onto the Windows clipboard, click on the *Existing* tab in the Welcome dialog, and press Ctrl-V to paste the name into the dialog (Figure 9-60). We then click *OK* to load the file into the tool, and edit the information that goes into the new deployment descriptors.



File name: 4AD\dev\src/../modules/piggybank.ear

Figure 9-60 Opening the EAR file using AAT

When we have finished working with the descriptor information, we save the EAR file and exit from AAT. When the assembly tool terminates, Ant takes the edited EAR file, and extracts the deployment descriptors using the Ant `unjar` task. The output from Ant as it extracts the files is shown in Figure 9-61.

```
     [echo] Assembly tool has terminated - processing changed deployment descriptors
    [unjar] Expanding: D:\ITSO4AD\dev\src\ear\..\..\modules\piggybank.ear into
D:\temp\edit-ear.20010607.1021
     [copy] Copying 3 files to D:\ITSO4AD\dev\src\ear\META-INF
    [mkdir] Created dir: D:\temp\edit-ear.20010607.1021\ejb
    [unjar] Expanding: D:\temp\edit-ear.20010607.1021\piggybank-ejb.jar into
D:\temp\edit-ear.20010607.1021\ejb
     [copy] Copying 4 files to D:\ITSO4AD\dev\src\ejb\META-INF
    [mkdir] Created dir: D:\temp\edit-ear.20010607.1021\client
    [unjar] Expanding: D:\temp\edit-ear.20010607.1021\piggybank-client.jar into
D:\temp\edit-ear.20010607.1021\client
     [copy] Copying 3 files to D:\ITSO4AD\dev\src\client\META-INF
    [mkdir] Created dir: D:\temp\edit-ear.20010607.1021\webapp
    [unjar] Expanding: D:\temp\edit-ear.20010607.1021\piggybank-webapp.war into
D:\temp\edit-ear.20010607.1021\webapp
     [copy] Copying 1 file to D:\ITSO4AD\dev\src\webapp\META-INF
     [copy] Copying 1 file to D:\ITSO4AD\dev\src\webapp\WEB-INF
   [delete] Deleting directory D:\temp\edit-ear.20010607.1021
     [echo] EAR meta-data has been copied

BUILD SUCCESSFUL

Total time: 4 minutes 42 seconds
```

Figure 9-61 Ant unjar output from the edit-ear target

Installing the EAR file

The `install` target is used to deploy our PiggyBank EAR file into a local instance of WebSphere Application Server, Single Server Edition. It uses the WebSphere `SEAppInstall` command to install the archive, supplying command line parameters that prevent the tool for prompting for any missing information. The XML that implements the target is shown in Figure 9-62.

```
<target name="install" depends="init">
  <echo>Installing EAR file ${ear.file} into WebSphere AEs</echo>
  <exec executable="SEAppInstall">
    <arg line="-install ${ear.file}"/>
    <arg line="-precompileJsp false"/>
    <arg line="-interactive false"/>
    <arg line="-ejbdeploy false"/>
  </exec>
  <echo>EAR file ${ear.file} installed</echo>
</target>
```

Figure 9-62 EAR build file install target

We use the `-ejbdeploy false` flag to tell WebSphere not to regenerate the deployed code for the EJBs when the EAR is installed. We can do this because our EAR file already contains EJBs with deployed code generated, as described in "Packaging EJBs and generating deployed code" on page 217.

The `-precompileJsp` option is set to `false` to save time when installing the application—in a unit testing environment we are less likely to require every single JSP in the application, so the effort spent precompiling them would be largely wasted.

Sample output from Ant when the `install` target is invoked is shown in Figure 9-63. We can safely ignore the warnings because our application does not use security, and the datasource used by the CMP EJBs has been defined at the container level, so the individual beans will inherit that.

```
D:\ITSO4AD\dev\src\ear>ant install
Buildfile: build.xml
init:
     [echo] Build of ear started at 1942 on June 6 2001
install:
     [echo] Installing EAR file D:\ITSO4AD\dev\src\ear/../../modules/piggybank.ear into WebSphere
            AEs
     [exec] IBM WebSphere Application Server Release 4, AEs
     [exec] J2EE Application Installation Tool, Version 1.0
     [exec] Copyright IBM Corp., 1997-2001
     [exec]
     [exec] The -configFile option was not specified. Using
                             D:\WebSphere\AppServer\config\server-cfg.xml
     [exec] Loading Server Configuration from D:\WebSphere\AppServer\config\server-cfg.xml
     [exec] Server Configuration Loaded Successfully
     [exec] Loading D:\ITSO4AD\dev\modules\piggybank.ear
     [exec] Getting Expansion Directory for EAR File
     [exec] Expanding EAR File to D:\WebSphere\AppServer\installedApps\piggybank.ear
     [exec]     Removed EAR From Server
     [exec] Installed EAR On Server
     [exec] Validating Application Bindings...
     [exec] CHKW4518W: No datasource has been specified for the container managed entity bean ?.
                      The default datasource specified for the EJB jar will be used.
     [exec] CHKW4518W: No datasource has been specified for the container managed entity bean ?.
                      The default datasource specified for the EJB jar will be used.
     [exec] CHKW6505W: A subject (user or group) has not been assigned for security role,
                       DenyAllRole.
                  The security role assignment should be made prior to running the application.
     [exec] Finished validating Application Bindings.
     [exec] Saving EAR File to directory
     [exec] Saved EAR File to directory Successfully
     [exec] Saving Server Configuration to D:\WebSphere\AppServer\config\server-cfg.xml
     [exec] Backing Up Server Configuration to: D:\WebSphere\AppServer\config\server-cfg.xml~
     [exec] Save Server Config Successful
     [exec] JSP Pre-compile Skipped......
     [exec] Installation Completed Successfully
     [echo] EAR file D:\ITSO4AD\dev\src\ear/../../modules/piggybank.ear installed
BUILD SUCCESSFUL

Total time: 14 seconds
```

Figure 9-63 Output from the install target

10

Development using WebSphere Studio

In this chapter we describe the process of developing a Web application using WebSphere Studio Version 4.0 Advanced Edition, which includes some new features, such as the support for WebSphere Application Server Version 4.0.

Detailed information about development of Web applications with WebSphere Studio Version 3 and Version 3.5 can be found in the redbooks:

- ***Servlet and JSP Programming with IBM WebSphere Studio and VisualAge for Java*, SG24-5755**
- ***Version 3.5 Self Study Guide: VisualAge for Java and WebSphere Studio*, SG24-6136**
- ***How about Version 3.5? VisualAge for Java and WebSphere Studio Provide Great New Function*, SG24-6131**

Developing Web applications with WebSphere Studio

WebSphere Studio Version 4.0 is a tool that allows us to develop a J2EE compliant Web application from end to end:

- Developing the Web presentation content: HTML, CSS, JSP
- Developing the server-side code: servlets, JavaBeans, database access
- Assembling the Web application: creating and publishing the WAR file to the server

WebSphere Studio components

WebSphere Studio includes the following tools to aid in the development of a Web application:

- *Page Designer, WebArt Designer, GifAnimator Designer*—tools to create and edit Web content (HTML and JSP pages, CSS stylesheets, images)
- *Applet Designer*—a visual tool for creating Java applets combining JavaBeans and adding multimedia content
- *Page Detailer*—tool to analyze the performance of the site (download times)

In addition a set of wizards are provided to generate skeleton code:

- *SQL, database, and JavaBean wizards*—allow to retrieve data from databases or access beans and include it in the Web pages as dynamic content
- *Web service creation/consumption wizards*—create and consume Web services. The creation wizard generated WSDL and XML files describing a Web service based on a JavaBean or servlet. The consumption wizard generates a JSP and a client proxy to interact with an existing Web service.
- *Jar creation wizard*—acts as a companion to Applet Designer, converting any class file into a JavaBean that can be added to an applet using this tool.
- *Content and user wizards*—used to define collections of personalization resources, by connecting to a database and building the resources from its tables.
  - User resources: attributes of the users visiting the Web site
  - Content resources: content attributes to be inserted in the Web pages

  To execute these two wizards, it is necessary that the properties for the project are setup to Version 3.5 of the application server and compile-time classes.

For developing the pure Java content (servlets, JavaBeans and their utility classes), the integration with VisualAge for Java Version 4.0 allows to import content directly from the repository into the Web archive (this is a feature already existing in previous versions of Studio). For details about this topic, see section "Integration with VisualAge for Java" on page 246 in this chapter.

There is also support for debugging using the IBM Distributed Debugger (this topic is covered in Chapter 17, "Debugging the application" on page 467).

Figure 10-1 shows the Studio workbench with the Page Designer.

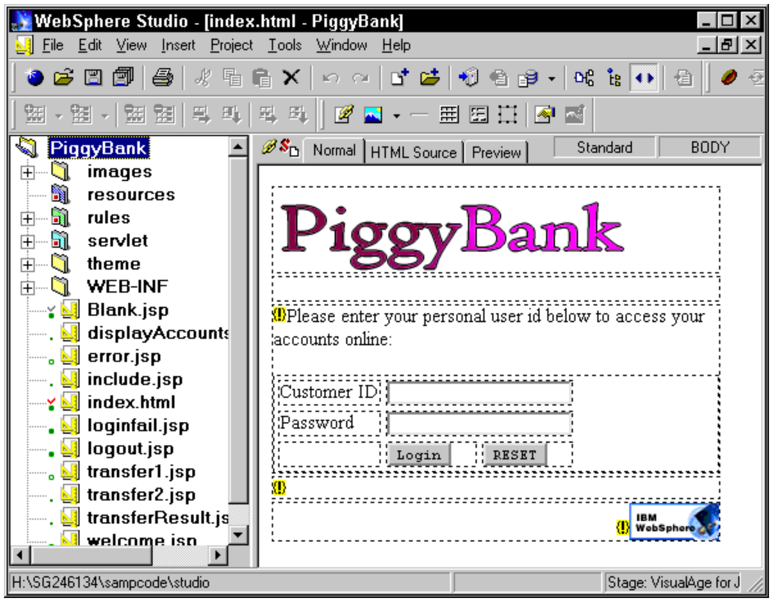


Figure 10-1 Page Designer window

New features in WebSphere Studio Version 4.0

The new version of WebSphere Studio includes several new features:

- Support for WebSphere Application Server Version 4.0: this implies support for the J2EE specifications: servlet 2.2 and JSP 1.1
- Creation of Web archives (WAR files): in this new version of Studio, it is possible to create the Web modules and publish them to the server locally or via FTP.
- Support for custom tag libraries (as part of the JSP 1.1 specification)
- New creation and consumption wizards for Web services

Structuring the project in Studio

The default structure of a Studio project is shown in Figure 10-2:

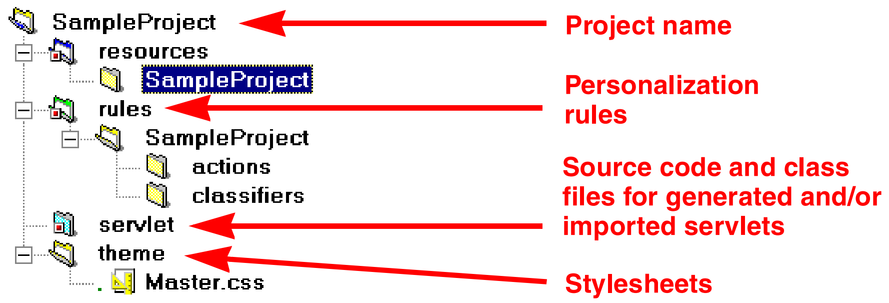
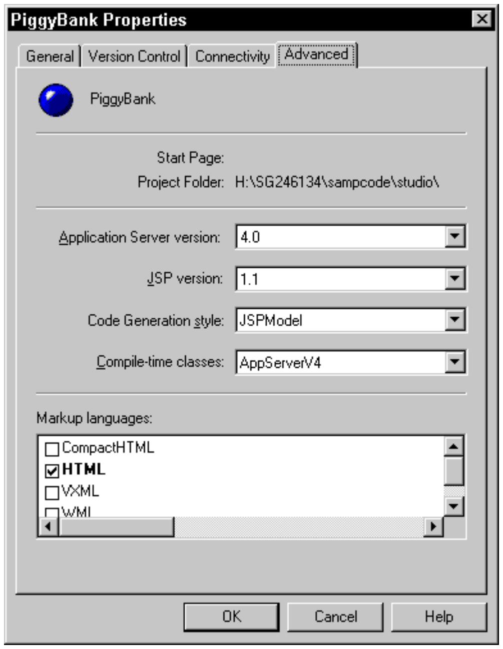


Figure 10-2 Default structure for a project in WebSphere Studio

When structuring a custom project, we might want to create other folders containing JSP and/or HTML pages, images, and so forth.

For each project, we must setup the properties related to the application server where we will publish the files (Figure 10-3).



PiggyBank

Start Page:
Project Folder: H:\SG246134\sampcode\studio\

Application Server version: 4.0
JSP version: 1.1
Code Generation style: JSPModel
Compile-time classes: AppServerV4

Markup languages:
- [ ] CompactHTML
- [x] HTML
- [ ] VXML
- [ ] WML

Figure 10-3 Setting up properties for the project

In the project's properties window, *Advanced* tab, we specify:

- *Application Server version*—Studio 4.0 supports WAS 3.0, 3.5 and 4.0, but in this book we focus on WAS 4.0
- *JSP version*—if our application is using custom tag libraries, it is necessary that we select JSP 1.1 (we do that in the case of the PiggyBank application).

  In the case of a general application (not deliberately developed for WAS 4.0 but that is going to be installed in it), it is necessary to migrate to JSP 1.1 if we have been using JSP 0.91 (which is no longer supported by the new version).
- *Code generation style*—this is used when we have generated code using the database or JavaBeans wizard. Two styles are possible:
  - *Servlet model*—it acts similarly to the *Servlet SmartGuide* in VisualAge for Java, creating a servlet that processes the request from an input HTML page, and a JSP page that receives the response (creating an error page or a "no data returned" page is optional).
  - *JSP model*—no servlet is created; instead, a JSP is in charge of processing the request from an HTML page and send the response to another JSP page. Again, the creation of an error/no data page is optional.

    In the database wizard, with this code generation style, the connection to the database is done in the JSP page, using the WebSphere custom tags (see the product documentation for details).

    In general, we do not recommend this usage, as it means tight coupling between the presentation and the model (database) tier; if the database schema changes, we would have to update all the JSPs that use the WebSphere database tags. See Chapter 13, "Guidelines for coding WebSphere applications" on page 325 for more information about coding guidelines.
- *Compile-time classes*—because we are deploying to WAS 4.0, we select the classes that match this version of the application server.
- *Markup languages*—WebSphere supports HTML, WML, VHML and CompactHTML (so that we can develop applications for pervasive devices).

Publishing stages and publishing targets

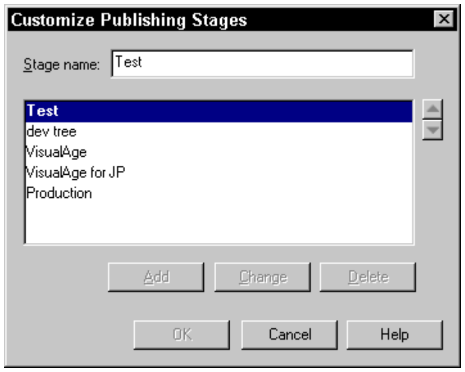
A useful feature of Studio is the possibility to create different publishing stages. A publishing stage is a model structure of the Web application used for publishing purposes. By defining different stages, we can publish our application to different servers, or with different functionality. For example, we can define publishing stages for the development, testing and production phases of the project. By default, `Test` and `Production` are stages already configured in Studio.

Publishing stages can be associated with different servers and different publishing targets (that is, destination locations for the Web application files).

Consider for example that we are developing the Java code in VisualAge and the Web pages in Studio. Then we can define a publishing stage called `VisualAge` with publishing targets aiming at the WebSphere Test Environment folder in the VisualAge for Java project resources. This way we can test our application in VisualAge for Java before deploying our Java code out of the repository.

Another scenario related to testing would be the use of WebSphere Application Server Version 4.0 Single Server Edition (its features are described in “New single server version” on page 66). This edition of the application server is intended for unit testing and development, so that in this case, we would define a Web application *war* publishing target (Creation of Web archive WAR files is described in “Creating and publishing WAR files” on page 249) and publish our Web application to the server using FTP.

Figure 10-4 shows an example of custom publishing stages.



Stage name: Test

Test
dev tree
VisualAge
VisualAge for JP
Production

Figure 10-4 Customizing publishing stages in WebSphere Studio

Custom file status

Another feature of Studio is the ability to create custom status for the project files. By default, Studio includes three status definitions:

- Work-in-Progress
- Submitted-for-Approval
- Ready-for-Publishing

It is possible to define five more status, each of them associated to a color label. Figure 10-5 shows examples of possible custom status definitions.

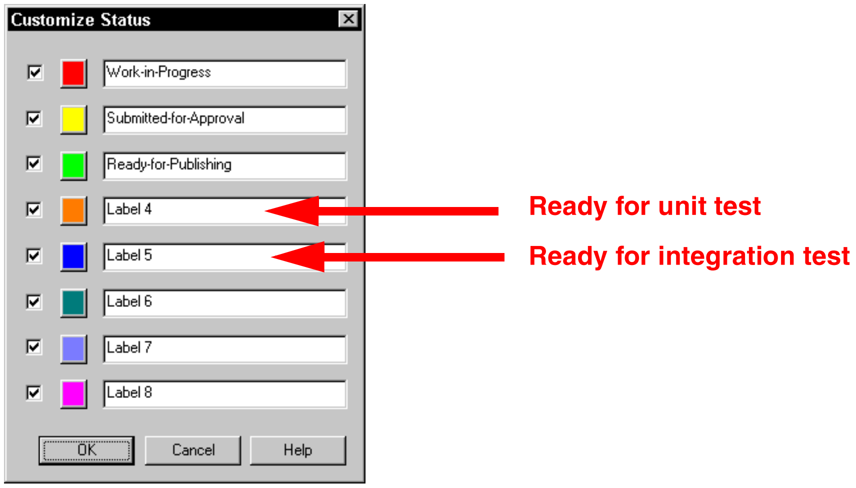


Figure 10-5 Customizing file status in WebSphere Studio

When working in a team environment, this feature can allow developers to quickly identify the status of the files. We can define additional status codes:

- To specify file ownership (this is not suitable if the Web development group is too big, as there are not enough available status)
- To specify details about the content of the files in an immediate recognizable way (JSPs with just HTML content, needing Java code, and so forth)

Working in a team environment with Studio

A possible configuration for this scenario is to place the Studio project and its files in a network drive in some centralized server so that all the Web developers can access it and perform their work.

When a file is being edited (for example, with Page Designer), Studio marks it as *checked out*, and prevents any other developer to access it (except in read-only mode). *Check out* is also the label assigned to files under a Version Control System not accessible for the current user (when that user is not in the group that can edit the file).

WebSphere Studio stores the checked out files in a default location under the directory where the product is installed (though it is modifiable):

```
D:\WebSphere\Studio40\check_out\ProjectName
```

The files stored at this location are the files currently being edited.

When the developer finishes working with a file, there are two possible options to add back the file to the general access:

- *Check In*—the changes are saved and the new file is copied from the check out location to the project default location.
- *Undo Check Out*—the changes are not saved and Studio reverts to the version of the file prior to the check out process.

Custom tag libraries

WebSphere Studio Version 4.0 includes support for using custom tag libraries as a part of the JSP 1.1 specification support.

Custom tags provide an abstraction of Java code as opposed to scriptlets, which require Java programming experience. For a Web developer in charge of writing JSPs, using custom tags is easier, as he/she may not have in depth Java knowledge. Also, with this approach, it is easier to maintain “separation“ between the presentation tier (the Web pages) and the logic beneath them: in case we want to change something in that logic, our custom tags can stay unchanged while we modify the functionality of the classes that handle them. With this view, we can consider custom tags as wrappers for the Java code.

Using tag libraries also means making the debugging process easier: although VisualAge for Java supports JSP debugging (through the generated servlet code), and we can use the Distributed Debugger in WebSphere Studio, the task of debugging a JSP is not as easy as debugging normal Java classes (this topic is covered in Chapter 17, “Debugging the application” on page 467 in *A special case: how to debug a JSP*).

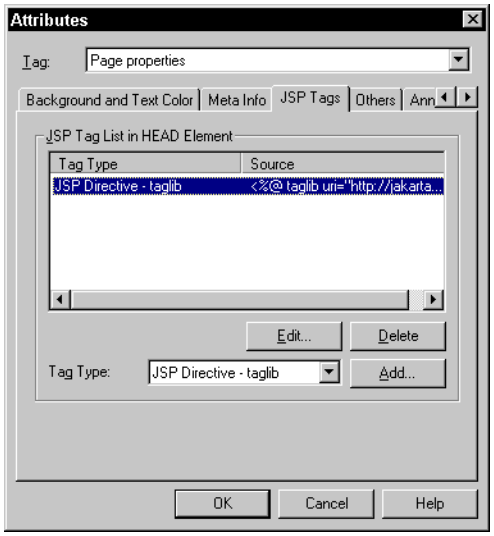
Custom tag libraries also mean reusability: having a tag library defined, we may be able to reuse it within our application or in other applications.

Let’s go through the process of adding and using a custom tag library for the PiggyBank application.

A tag library file is an XML file containing information about the tags, basically their name, associated Java class, descriptive information and attributes. Their processing will be delegated to these associated classes. To learn more about tag libraries, see the redbook *Programming J2EE APIs with WebSphere Advanced*, SG24-6124.

The tag library and its supporting Java classes are typically developed in VisualAge for Java and imported into Studio so that the Web developers can add the tags to the JSP pages and assemble the components together to build the Web module and publish the files.

In the case of the PiggyBank application, the tag library file is named `utility.tld`. When using Studio, we add this file to the `WEB-INF` directory of our project (if the folder does not exist, we create it). Then, to use this taglib in a JSP page, we have to insert a taglib directive in the *Page Properties* panel on the JSP tags tab (Figure 10-6).



Tag Type	Source
JSP Directive - taglib	<%@ taglib uri="http://jakarta...

Tag Type: JSP Directive - taglib

Figure 10-6 Including a tag library in a JSP page

The code inserted in the `<HEAD>` tag of the JSP page is shown in Figure 10-7.

```
<HTML>
<HEAD>
<META name="GENERATOR" content="IBM WebSphere Page Designer V4.0 for
Windows">
<META http-equiv="Content-Style-Type" content="text/css">
<TITLE>
This is a sample JSP page using a custom tag library
</TITLE>
<LINK href="/theme/Master.css" rel="stylesheet" type="text/css">
<%@ taglib uri="http://jakarta.apache.org/taglibs/utility" prefix="utils" %>
</HEAD>
<BODY>
   <!-- custom tags used here -->
</BODY>
</HTML>
```

Figure 10-7 Including a tag library in a JSP page: the taglib directive

Then we can proceed to use the custom tags in the JSP page, by selecting the menu *Insert -> JSP tags -> Insert a Custom tag*. The window displayed displays a list of the tag libraries loaded in the project and the corresponding tags in each of the tag libraries (Figure 10-8).

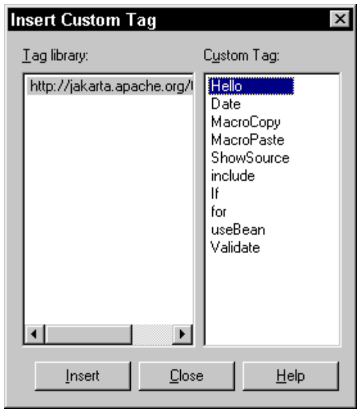


Figure 10-8 Inserting custom tags from tag libraries

When assembling the Web module in a WAR file for publication in the application server, we have to include the classes that process our custom tags in the module, in a JAR file under `\WEB-INF\lib`. This is in fact the folder where the utility classes JAR files must be placed (classes used by other components, such as servlets, or directly by the JSPs in scriptlets).

Note: The `WEB-INF` directory name must be in upper case in J2EE archive files

Integration with VisualAge for Java

One important feature of the integration between VisualAge for Java and Studio is the possibility of exporting/importing code between the two tools. For example, servlets or Java beans developed with the Studio wizards can be imported into VisualAge for Java to be debugged or improved with new functionality. When the process is complete, the servlet class file can be exported to WebSphere Studio to be assembled in the Web module (WAR file) and published to the application server.

Setup

To access VisualAge for Java code from Studio, it is necessary to start the *Remote Access to Tool API* in VisualAge, by selecting *Options -> Remote Access to Tool API* and start the service.

To access Studio files from VisualAge for Java, it is not necessary that Studio is running. To import Java code generated in Studio to VisualAge for Java, we can use the import SmartGuide in the usual way, or install the *WebSphere Studio Tools* so that we can send directly the Java source code to the VisualAge for Java Workbench from Studio. Select *Project -> VisualAge for Java -> Install Studio Tools in VisualAge* and the installation is automatic (it is required to restart VisualAge for Java for the changes to take effect).

Interfacing with VisualAge for Java from Studio

While specialized Java developers may be working in the servlets and JavaBeans in VisualAge for Java, the Web developers write the JSPs and other presentation content in Studio, and whenever it is necessary to create a build for the Web application (assemble the WAR file), Studio users can request the update of the code from VisualAge for Java and get the latest version of the class files.

If the main Java development tool is VisualAge for Java, it is a good practice to have just the class files in the Studio project, while the Java source code only exists in the VisualAge for Java repository. Also, the Java source code is never published in Studio.

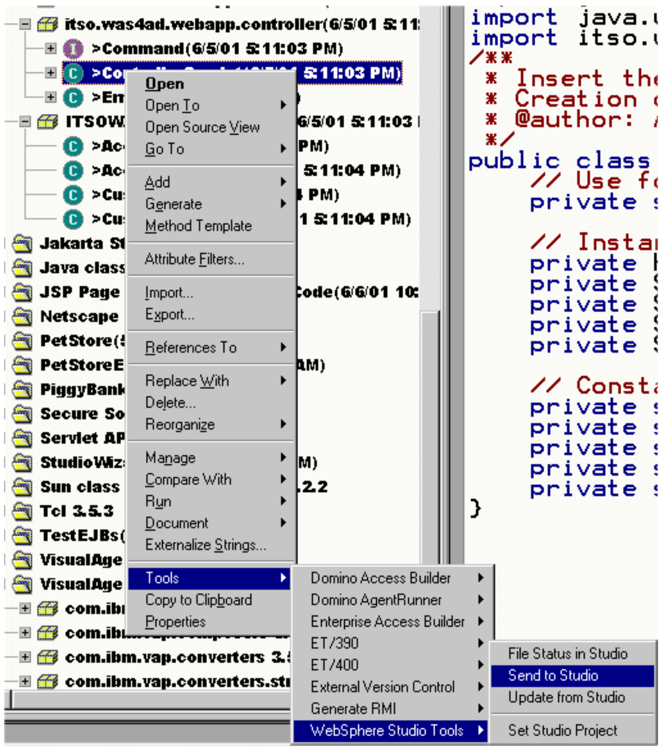
We can send the Java source files from Studio to VisualAge for Java by selecting the appropriate file in the workbench, and then selecting *Project -> VisualAge for Java -> Send to VisualAge*. On the first operation we are prompted to select the VisualAge for Java project that is used to import the Java code.

In the same way, we can update Java and/or class files by selecting *Project -> VisualAge for Java -> Update from VisualAge*. The files are retrieved from VisualAge for Java and checked out.

Interfacing with Studio from VisualAge for Java

After installing the Studio tools in VisualAge for Java we can send files to and retrieve files from a Studio project. Studio does not have to run, the interface is purely with the underlying file system where the Studio project is stored.

The Studio options in VisualAge for Java are shown in Figure 10-9.



```
import java.
import itso.
/**
 * Insert the
 * Creation
 * @author:
 */
public class
    // Use f
    private

    // Insta
    private
    private
    private
    private
    private

    // Const
    private
    private
    private
    private
    private
}
```

Figure 10-9 Using the Studio tools in VisualAge for Java

On the first request you are prompted to select the location of the Studio project by navigating to the appropriate `projectname.wao` file (the Studio project file). You can also set the Studio project at any time using the action shown in Figure 10-9.

Note that you retrieve Java source files (and they are imported), but you can only send class files. If you want to use the interface from VisualAge for Java, then that is where the master source files are kept. Class files can be sent to Studio for publishing.

Integration with other development tools

When using other Java development tools different from VisualAge for Java (for example, the Java 2 SDK as described in Chapter 9, “Development using the Java 2 Software Development Kit” on page 183), integration with WebSphere Studio is also guaranteed.

For example, in the case of the Java SDK, we may develop and compile our Java components with the SDK and then import them in Studio to assemble the Web module. Similarly, the Java components created in Studio via the wizards can be refined and improved with the SDK (as with VisualAge for Java).

To test the components, if we are not using VisualAge for Java and the WebSphere Test Environment but other tools such as the J2SDK and Apache Tomcat, we can anyway proceed as in the WAS case: defining a publishing target to the server so that we can test the Web application.

To manage version control it would be advisable to use a separate SCM system, such as Rational ClearCase, that can be integrated with Studio, to provide version control for both parts of the Web application (presentation and Java code). Version control and SCM are discussed in more extent in Chapter 14, “Software Configuration Management” on page 385.

Creating and publishing WAR files

Creating the Web application WAR file in WebSphere Studio 4.0 allows the developer to assemble the Web module directly in his environment, instead of doing it on the application server environment (this approach is covered in Chapter 15, “Assembling the application” on page 389).

This new feature of Version 4.0 can only be used when developing an application based on the servlet 2.2 and JSP 1.1 specification. These are the only levels supported by WAS Version 4.0, though, because JSP 1.1 is a superset of JSP 1.0, we can consider that 1.0 JSPs will work in the Version 4.0 environment.

We now show an example of creation and deployment of a WAR file with Studio 4.0 to WebSphere Application Server 4.0, based on the PiggyBank application.

First of all, the properties of the project have to be properly configured to target the application server version and supported specifications, as we described in section “Structuring the project in Studio” on page 240.

After configuring the project, the server has to be setup to specify the following information:

- Server address
- Context root (Web application Web path in case of WAS 3.x)

That can be done selecting the server in the *Publishing View*, and then *Edit -> Properties -> Publishing*.

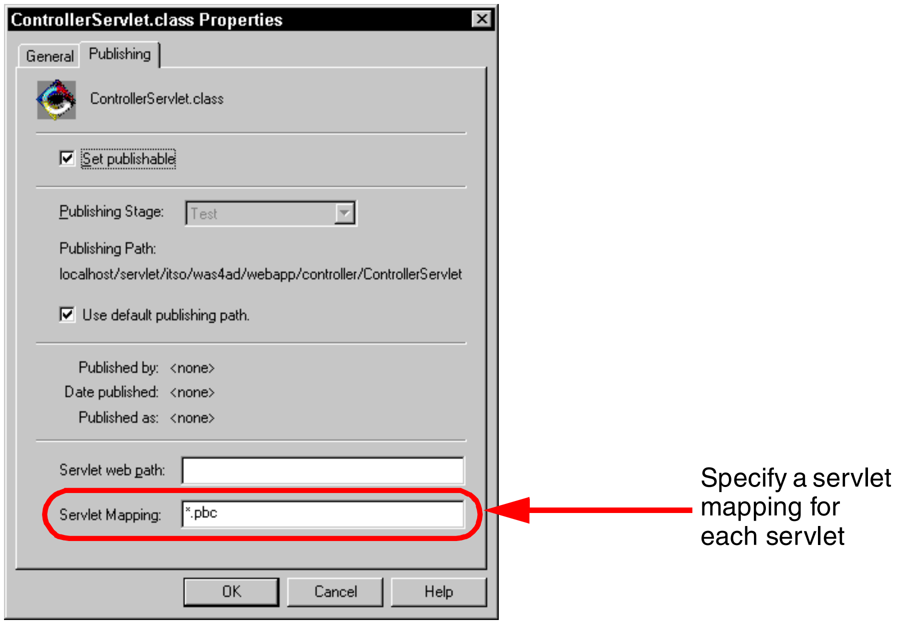
Creating the WAR file

We are ready to create the Web application file when we have added all the files to the project in Studio. The first step is to create the Web configuration file (the deployment descriptor of the Web module). This file contains information about the servlets and tag libraries included in the module.

Before creating the deployment descriptor file, we configure the Web application:

- For every servlet class file added to the module, we specify a servlet mapping.

  We do this by selecting the servlet properties box (right button menu), select the *Publishing* tab and type the required information (Figure 10-10).


ControllerServlet.class Properties

ControllerServlet.class

☑ Set publishable

Publishing Stage: Test

Publishing Path:
localhost/servlet/itso/was4ad/webapp/controller/ControllerServlet

☑ Use default publishing path.

Published by: <none>
Date published: <none>
Published as: <none>

Servlet web path:

Servlet Mapping: *.pbc

Figure 10-10 Specifying servlet mappings in Studio

  If we are using the AutoInvoker servlet in WAS, we have to specify Web paths for every servlet in our Web application (WAS 3.x).

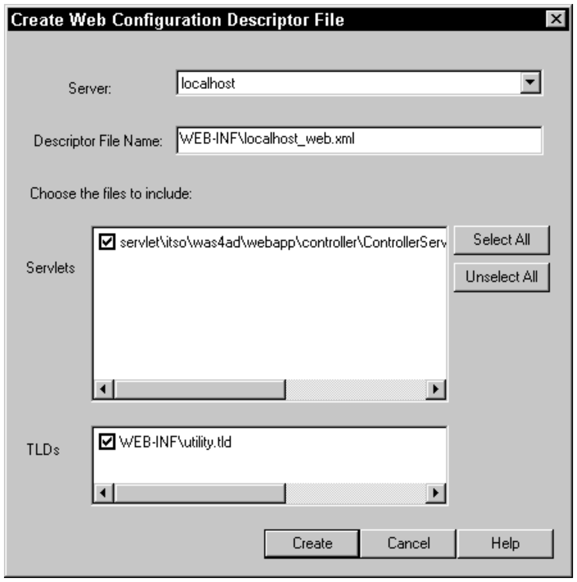
- We include the taglib file under the `\WEB-INF` folder
- Last, we define the Web path for the Web application by selecting the *Properties* right-button menu for the server.

Then we are ready to generate the deployment descriptor file. To do so, we select *Project -> Create Web Configurator Descriptor File.*

We can select which components of the project we want to include in the deployment descriptor, but when we generate the WAR file, all the elements of the project will be included in it. If we want to make several different Web modules out of our Studio project, it is a good idea to create several publishing stages, so that only the desired files are included in each stage. Then we generate the descriptor and the Web module per stage (or for the configurations we want to publish and test).

This can be a suitable approach when we want to unit test parts of the Web application. For example, in an online banking application such as PiggyBank, we might want to test only the account managing options, so we would create a publishing stage including only the files related to account management. Then we would create the descriptor and the WAR file and install the Web module in the application server for testing.

The deployment descriptor creation window is shown in Figure 10-11.


Create Web Configuration Descriptor File

Server: localhost

Descriptor File Name: WEB-INF\localhost_web.xml

Choose the files to include:

Servlets: ☑ servlet\itso\was4ad\webapp\controller\ControllerServ

TLDs: ☑ WEB-INF\utility.tld

Figure 10-11 Creating the Web application deployment descriptor in Studio

WebSphere Studio creates this file automatically with the default name of `servername_web.xml`. The default location to place it is in the `\WEB-INF` folder.

Figure 10-12 shows the resulting structure of the project in Studio's workbench.

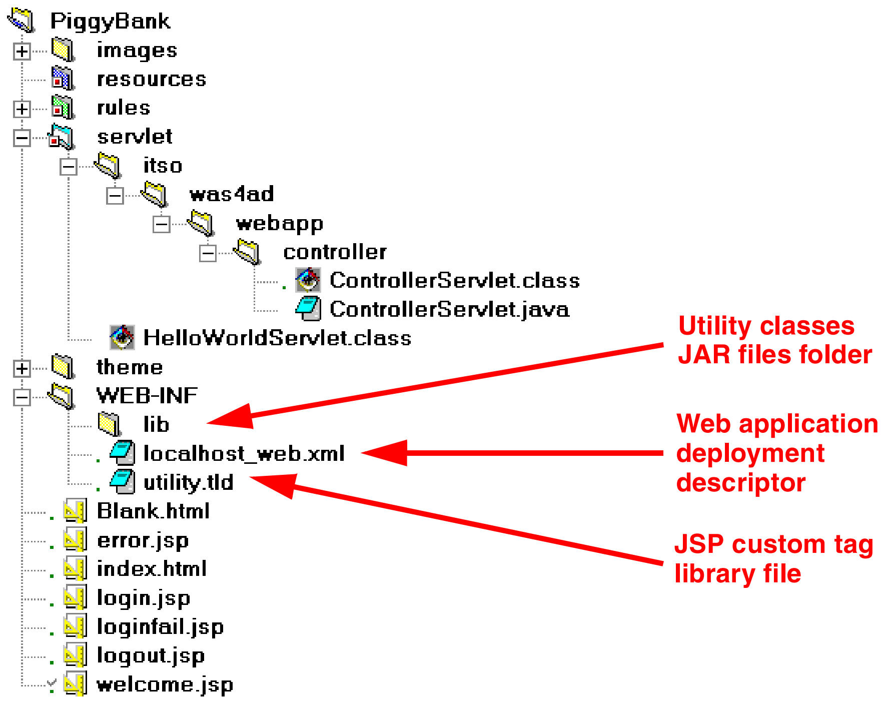


Figure 10-12 Project structure in Studio Version 4.0

Once the deployment descriptor has been generated, we can generate the WAR file. This file contains all the components in the current publishing stage. As it is possible to generate several deployment descriptor files (depending on the contents of the stage), when we select the *Project -> Create Web Archive file,* we have to select the appropriate one for our purposes. In case we have not created one yet, it is possible to do so at this point. We also select the server name to which the Web module is going to be installed, and the save location.

The last step is to publish the just created WAR file to the application server, where it will be installed as a stand alone module or as part of an enterprise application.

Tip: It is useful to create a *war* publishing stage to publish the whole Web module to the desired location in the application server (publishing through FTP).

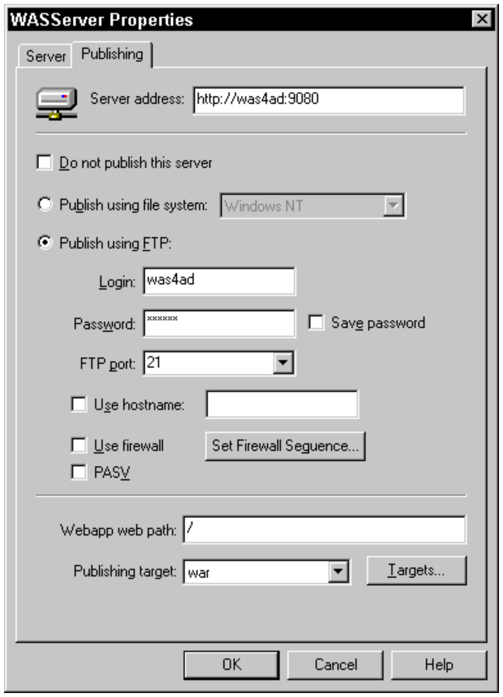
War files are published via FTP to a folder in the application server machine that has been configured for this type of access (this requires that an FTP server is installed in that machine). In the case of WAS 4.0, we can publish to:

```
D:\WebSphere\AppServer\installableApps
```

Because this is the folder that contains the applications available to be installed in the server.

In the case of our application, we used the **WAR FTP Daemon** (a shareware tool) as an FTP server, and we configured the `\installableApps` folder with permissions to read and write. We configured a user, *was4ad*, that could access only this folder (for security reasons, in case we are accessing the server via FTP for other purposes apart from publishing).

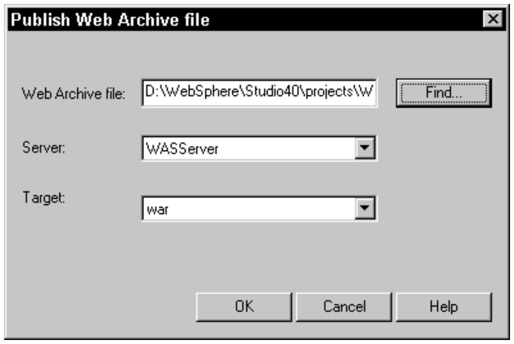
Then, after setting up the FTP server, we configured the publication in Studio for a server with the same name (Figure 10-13).



Server address: http://was4ad:9080
[ ] Do not publish this server
( ) Publish using file system: Windows NT
(•) Publish using FTP:
Login: was4ad
Password: ******
[ ] Save password
FTP port: 21
[ ] Use hostname:
[ ] Use firewall
[ ] PASV
Webapp web path: /
Publishing target: war

Figure 10-13 Publishing via FTP in Studio

To publish the WAR file, we select *Project -> Publish Web Archive*, then select the appropriate data (Figure 10-14).



Web Archive file: D:\WebSphere\Studio40\projects\W

Server: WASServer

Target: war

Figure 10-14 Publishing the Web archive file to a server

Then we can use either the Administrative Console (Advanced Edition and Single Server Edition) or the SEAppInstall command line tool (Single Server Edition) to install the application. For details about this process refer to Chapter 16, “Deploying to the test environment” on page 431.

Web services wizards

WebSphere Studio Version 4 provides two new wizards to create and consume Web services.

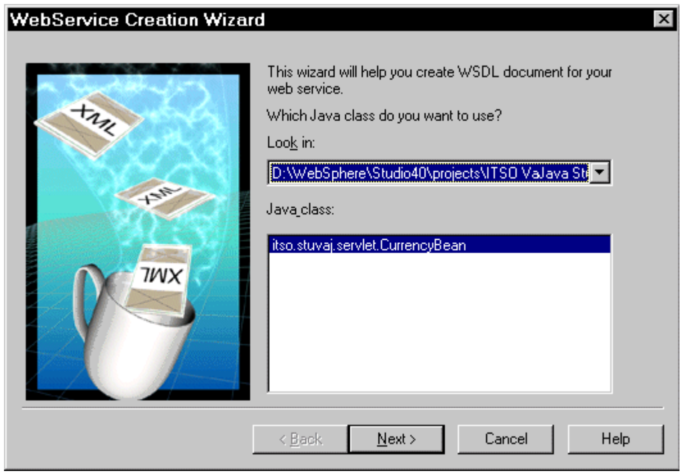
Web services creation wizard

The creation wizard is started using *Tools -> Wizards -> Web Service Creation Wizard*. The wizard looks in the servlet folder for beans and servlets, displays a list, and you have to select an entry:

- When selecting a bean, you are presented with the list of methods, and then the Web services description language (WSDL) files and an XML file are generated.
- When selecting a servlet, you choose between `doGet` and `doPost` methods, and you specify the parameters that have to be passed to the servlet. Again two WSDL files are generated.

Example

In this example we use an existing JavaBean (CurrencyBean) and generate a Web service for it (Figure 10-15).



This wizard will help you create WSDL document for your web service.

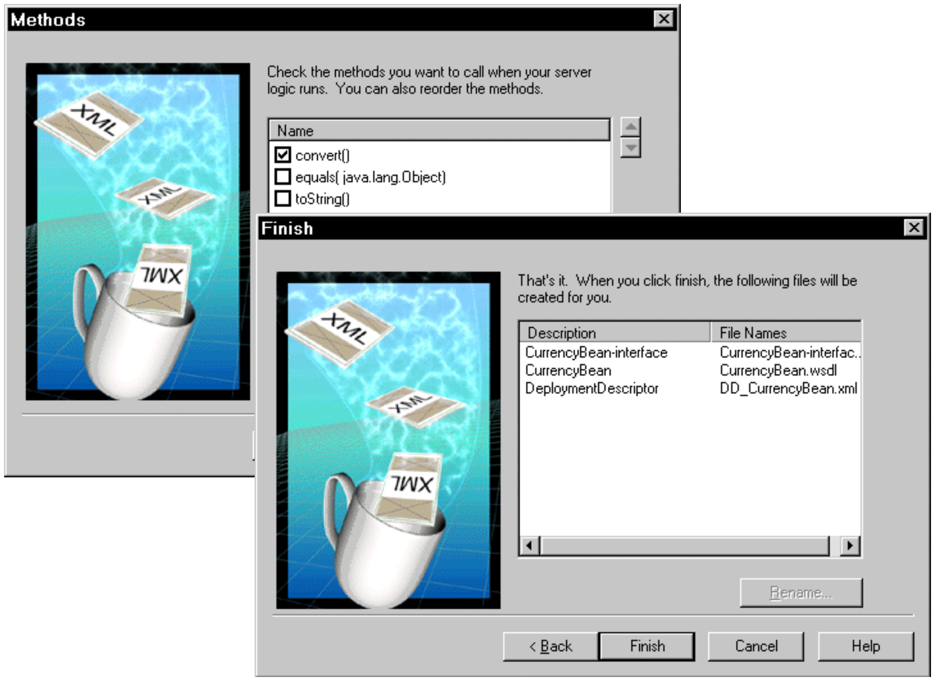
Which Java class do you want to use?

Look in: D:\WebSphere\Studio40\projects\ITSO VaJava St

Java class: itso.stuvaj.servlet.CurrencyBean

Figure 10-15 Web services creation wizard bean selection

The wizard displays the methods of the bean for your selection, and finally displays the files that will be generated (Figure 10-16).



Check the methods you want to call when your server logic runs. You can also reorder the methods.

Name
☑ convert()
☐ equals(java.lang.Object)
☐ toString()

That's it. When you click finish, the following files will be created for you.

Description	File Names
CurrencyBean-interface	CurrencyBean-interfac..
CurrencyBean	CurrencyBean.wsdl
DeploymentDescriptor	DD_CurrencyBean.xml

Figure 10-16 Web services creation wizard code generation

The output files are placed into the servlet folder. The files are:

- `DD_CurrencyBean.xml`—deployment descriptor, provides instructions for SOAP for loading the class files (Figure 10-17)
- `CurrencyBean.wsdl`—implementation, defines where the interface is published and where the Web service is deployed(Figure 10-18)
- `CurrencyBean-interface.wsdl`—interface, to be published to a UDDI registry to discover and consume the service (Figure 10-19)

```
<isd:service xmlns:isd="http://xml.apache.org/xml-soap/deployment"
id="urn:currencybean-service" checkMustUnderstands="false">
  <isd:provider type="java" scope="Application" methods="convert">
    <isd:java class="CurrencyBean_ServiceService" static="false"/>
  </isd:provider>
</isd:service>
```

Figure 10-17 Generated deployment descriptor XML file

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="CurrencyBean_Service"
  targetNamespace="http://www.currencybeanservice.com/CurrencyBean"
  xmlns="http://schemas.xmlsoap.org/wsdl/"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:tns="http://www.currencybeanservice.com/CurrencyBean"
    xmlns:xsd="http://www.w3.org/1999/XMLSchema">

<import
    location="http://localhost:8080/wsdl/CurrencyBean-interface.wsdl"
    namespace="http://www.currencybeanservice.com/CurrencyBean-interface">
</import>

<service
      name="CurrencyBean_Service">
  <documentation>IBM WSTK 2.0 generated service definition file
  </documentation>
  <port
      binding="CurrencyBean_ServiceBinding"
        name="CurrencyBean_ServicePort">
    <soap:address location="http://localhost:8080/soap/servlet/rpcrouter"/>
  </port>
</service>

</definitions>
```

Figure 10-18 Generated implementation WSDL file

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="CurrencyBean_Service"
  targetNamespace="http://www.currencybeanservice.com/CurrencyBean-interface"
  xmlns="http://schemas.xmlsoap.org/wsdl/"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
  xmlns:tns="http://www.currencybeanservice.com/CurrencyBean"
    xmlns:xsd="http://www.w3.org/1999/XMLSchema">
<message
    name="InconvertRequest"/>
<portType
      name="CurrencyBean_Service">
  <operation
        name="convert">
    <input
    message="InconvertRequest"/>
  </operation>
</portType>
<binding
    name="CurrencyBean_ServiceBinding"
      type="CurrencyBean_Service">
  <soap:binding style="rpc"
      transport="http://schemas.xmlsoap.org/soap/http"/>
  <operation
        name="convert">
    <soap:operation
          soapAction="urn:currencybean-service"/>
    <input>
      <soap:body
          encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
          namespace="urn:currencybean-service" use="encoded"/>
    </input>
  </operation>
</binding>
</definitions>
```

Figure 10-19 Generated interface WSDL file

The implementation WSDL file must be edited to define the correct URL for the interface file (<import> tag) and the deployed Web service (<soap:address> tag).

Web services consumption wizard

The consumption wizard is started using *Tools -> Wizards -> Web Service Creation Wizard*. The consumption wizard uses an existing WSDL interface file as the base for generating a JSP and a client proxy.

Because the consumption wizard must create a connection to the Web service in order to create the Java client proxy, you must publish the interface WSDL file before attempting to use the implementation WSDL file.

The wizard examines the WSDL file and provides you with a list of the available methods in the Web service. After choosing the method(s) the Java proxy client and the JSP are generated. Based on the selected methods, complex type handlers and serializers for parameters may be generated in addition.

The wizard notifies you if you have to modify the JSP to edit the parameters. You may also have to edit the JSP to handle the output of the Web service.

11

Development using VisualAge for Java

In this chapter we describe application development with VisualAge for Java Version 4.0. We cover:

- General concepts of development and organization of the code in VisualAge for Java
- Web application development
- WebSphere Test Environment configuration and usage
- EJB developing and deployment

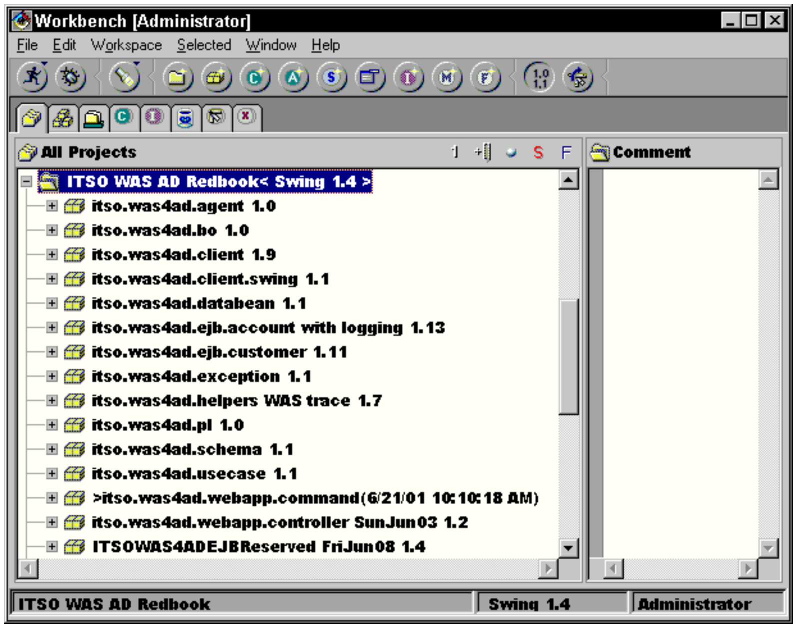
General information about programming with VisualAge for Java can be found in the redbook *Programming with VisualAge for Java Version 3.5*, SG24-5264.

The integrated development environment (IDE)

In this section we provide general information about code development in VisualAge for Java.

Configuring the projects and packages

Let's take a look at the structure of the PiggyBank application in VisualAge for Java (Figure 11-1).


Workbench [Administrator]
File Edit Workspace Selected Window Help
All Projects
Comment

ITSO WAS AD Redbook< Swing 1.4 >
itso.was4ad.agent 1.0
itso.was4ad.bo 1.0
itso.was4ad.client 1.9
itso.was4ad.client.swing 1.1
itso.was4ad.databean 1.1
itso.was4ad.ejb.account with logging 1.13
itso.was4ad.ejb.customer 1.11
itso.was4ad.exception 1.1
itso.was4ad.helpers WAS trace 1.7
itso.was4ad.pl 1.0
itso.was4ad.schema 1.1
itso.was4ad.usecase 1.1
>itso.was4ad.webapp.command(6/21/01 10:10:18 AM)
itso.was4ad.webapp.controller SunJun03 1.2
ITSOWAS4ADEJBReserved FriJun08 1.4

ITSO WAS AD Redbook | Swing 1.4 | Administrator

Figure 11-1 PiggyBank application in VisualAge for Java

When structuring the project in VisualAge for Java (as well as when using other development tools), it is convenient to set up some conventions for packages and code.

A general convention accepted for package naming is to set it as the URL of the company (reversed) plus some more detailed domain description. In our case, all the packages could have been named `com.ibm.itso.was4ad.pkgname[.subpkg]`.

When it comes to project structuring, we can consider different approaches:

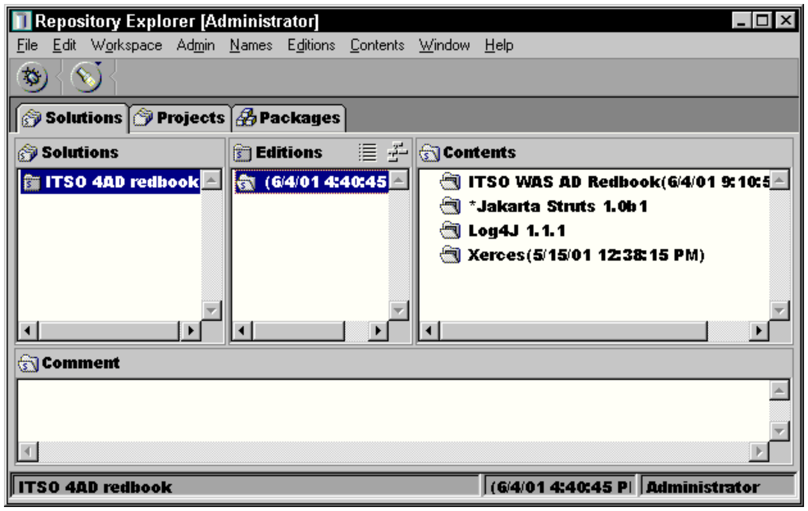
- Just one project including all the code

- Separate projects for different features or components of the application. An example of this could be as follows:
  - Core of the application
  - EJB classes
  - Utility packages (logging and tracing)

We can use this technique when we have several development teams, each of them in charge of a piece of the application (one team developing the EJBs, another team developing the logging utilities.). Setting up different user groups for each project also helps to maintain clearer boundaries between the teams.

Solution

If we have defined different projects according to the features or components of the application, it is useful to setup a global *Solution* for the whole application, as a means to ease the global version control process. A *Solution* is a container for related projects at a certain version. Solutions are defined in the *Repository Explorer* window (Figure 11-2).


Repository Explorer [Administrator]
File Edit Workspace Admin Names Editions Contents Window Help
Solutions
Projects
Packages
Solutions
Editions
Contents
Comment

ITSO 4AD redbook

(6/4/01 4:40:45

ITSO WAS AD Redbook(6/4/01 9:10:5
*Jakarta Struts 1.0b1
Log4J 1.1.1
Xerces(5/15/01 12:38:15 PM)

ITSO 4AD redbook | (6/4/01 4:40:45 P | Administrator

Figure 11-2 Defining a solution in the Repository Explorer

This mechanism is useful as an internal control of VisualAge for Java, because the *Solution* has no real existence out of the repository. Also, it allows the correct versions of related projects to be loaded together into the workbench.

Generating documentation in VisualAge for Java

VisualAge for Java provides options to setup the method and types head information both for SmartGuides or direct creation. Selecting *Window -> Options -> Coding*, we can edit both of this specifications. The javadoc information for the classes and methods created using SmartGuides can be edited in the *Macros* section. *Method Javadoc* and *Type Javadoc* detail the information to be included for methods and types (classes/interfaces) created directly by the user.

We can use the standard Javadoc tags, such as `@author` and `@version`, and the two macros predefined by VisualAge: `<user>` and `<timestamp>`, which insert the name of the Workspace owner and the timestamp of the method/type's creation.

To generate the documentation once the code is completed, select the project or packages for which we want to create documentation and select *Document -> Generate Javadoc* from the context menu. The options available are the same as with the standard `javadoc` command of the J2SDK, plus some more options regarding the aspect of the generated Web pages (header, footer, bottom line text).

Working in a team environment

Team development means how to organize a development project where a number of developers can simultaneously work on VisualAge for Java code. In a project where two or more developers access the same code, it is necessary to implement some kind of software control mechanism to ensure consistency.

VisualAge for Java provides two solutions to this problem:

- Team repository—a shared repository on a central server accessible by all developers. Change control works at the object level and is based on object ownership.
- External version control—an interface to third-party tools for Software Configuration Management (SCM). Read more about this topic in Chapter 14, “Software Configuration Management” on page 385.

It is possible to use both solutions simultaneously, though we do not recommend such a setup. When using an external SCM system, the shared repository should not be configured.

Figure 11-3 shows the configuration of the shared repository solution.

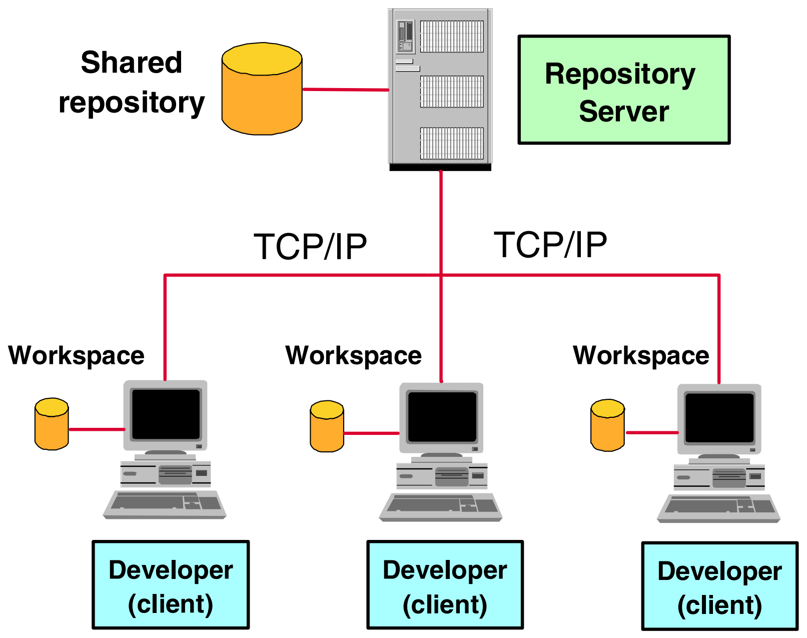


Figure 11-3 VisualAge for Java team development environment

In each of the developer's (client) machines there is a *workspace* file containing the developer's working code set. The shared repository is stored in a centralized server and the clients access it through TCP/IP connections. Both the workspace and the repository files are binary files. The *repository* file contains all the source and compiled code of all the developer's work spaces.

Clients can connect to the shared repository and explore it to add new projects or features to their work spaces. A system of permissions can be established so that only user with administrator's rights can perform delete operations. Code changes made in the local work spaces are automatically saved to the repository.

The concept of team development in VisualAge is based on *ownership*. Each element has an owner, and other users can be set as additional editors. The owner of the element is responsible for managing versions and releases of it. This ownership applies to classes, packages and projects. This implementation of the team development environment means that the developers do not have to perform *check in/check out* tasks, and the code is always accessible for everyone that has it loaded in his/her workspace.

Developing Web applications with VisualAge for Java

Web applications usually consist of HTML code, servlets, JSPs, and (optionally) EJBs. VisualAge for Java does not provide editing facilities for HTML and JSP code, although simple Web pages can be generated using the Servlet SmartGuide. WebSphere Studio provides the Page Designer for the editing of Web pages (Figure 10-1 on page 239).

Detailed information about development of Web applications with VisualAge for Java Version 3 and Version 3.5 can be found in the redbooks:

- ***Servlet and JSP Programming with IBM WebSphere Studio and VisualAge for Java*, SG24-5755**
- ***Version 3.5 Self Study Guide: VisualAge for Java and WebSphere Studio*, SG24-6136**
- ***How about Version 3.5? VisualAge for Java and WebSphere Studio Provide Great New Function*, SG24-6131**

Developing servlets

The supported servlet specification in VisualAge for Java Version 4.0 is 2.2. There are basically three ways of developing (and testing) servlets with VisualAge for Java:

- Hand-coding of the servlet code
- Using the Servlet SmartGuide to generate skeleton code
- Importing servlets generated by WebSphere Studio wizards or by other tools

Hand-coding servlets

Experienced Web programmers generally write servlets by hand as subclasses of the `HttpServlet` class. In many cases they copy existing servlets as models and then modify the code.

Servlet SmartGuide

The Servlet SmartGuide can generated skeleton servlet Java code and, if provided with a JavaBean, also a skeleton HTML input page and a result JSP:

- WIthout using the Import JavaBean option, the SmartGuide only generates the skeleton code for the basic servlet methods, and it is up to the developer to complete the code as he/she wishes.
- With the Import JavaBean option, the SmartGuide creates an HTML input page with a form that includes the specified fields of the bean, as well as a result JSP that displays the result of the operation performed when submitting

the form. This operation relates to a method of the JavaBean that we have selected. A JSP error page is also created for handling the exceptions caught in the servlet.

The generated servlet can be an `HttpServlet`, or a `PageListServlet` (an IBM extension that requires a .servlet configuration file).

With the Import JavaBean option enabled, the Servlet SmartGuide output is quite similar to the WebSphere Studio JavaBean wizard, though that wizard provides more options, for example, the code generation style (servlet or JSP model):

- Servlet model—with this choice, the wizard creates an HTML input page, a servlet that uses the JavaBean, a JSP that formats the result data, and a `.servlet` configuration file.
- JSP model—with this choice the wizard generates an HTML input page and a JSP that does all the processing.

Importing servlets from WebSphere Studio

Servlets generated by WebSphere Studio can be brought into VisualAge for Java for testing and debugging in several ways:

- Use VisualAge for Java to import the Java source files from any directory
- Use WebSphere Studio to "send" the code into the VisualAge for Java Workbench (see "Integration with VisualAge for Java" on page 246)
- Use VisualAge for Java to retrieve the Java source code from a Studio project directory (see "Integration with VisualAge for Java" on page 246)

When working with multiple products you have to have a well-defined process that specifies how code is shipped between products and where the "master" code resides.

Developing JSPs

VisualAge for Java 4.0 supports three JSP specifications: 0.91, 1.0 and 1.1, however, VisualAge for Java is a Java IDE and is not built for developing Web content (apart from the JSPs generated by the Servlet SmartGuide), but the functionality provided by the WebSphere Test Environment makes VisualAge for Java appropriate for testing and debugging purposes.

We develop our JSPs using tools such as WebSphere Studio (or writing the code from scratch in text editors such as Notepad) and copy them to the WebSphere Test Environment project resources folder for testing and debugging.

We can also copy JSPs to the Web application's project resources folder so that we can perform version control operations of all related code (Java and other).

Developing EJBs in VisualAge for Java

VisualAge for Java provides the EJB development environment to create and test EJBs. VisualAge for Java Version 4.0 supports only the EJB 1.0 specification, but an export SmartGuide is provided to deploy EJBs to WebSphere Application Server Version 4.0 that supports EJB 1.1.

Detailed information about EJB development can be found in the redbook *EJB Development with VisualAge for Java for WebSphere Application Server*, SG24-6144. This redbook also contains information about deployment of EJBs to WebSphere Application Server Version 3.5 and Version 4.0.

EJB development environment

The VisualAge for Java EJB development environment consists of:

- EJB page—where EJB groups and enterprise beans are defined
- Persistent name server—where EJBs are registered for testing (part of the WebSphere Test Environment)
- EJB server—provides an EJB container to instantiate enterprise beans
- EJB test client—enables testing of any EJB running in the EJB container

EJB 1.1 specification

EJB 1.1 addresses many of the limitations and loop holes found in EJB 1.0. The most significant changes are listed here:

- Entity bean support, both container- and bean-managed persistence, is required.
- Java RMI-IIOP argument and reference types must be supported. That is the client API must support the Java RMI-IIOP programming model for portability, but the underlying communication protocol can be anything.
- The `javax.ejb.deployment` package has been dropped in favor of a XML based deployment descriptor.

  In the 1.0 specification, each EJB had its own deployment descriptor. In 1.1 specification, only one deployment descriptor has to be written per EJB JAR file (and it contains information about all the beans in that JAR file).
- Declarative security authorization (access control) has changed to be more role driven.
- Isolation levels are now managed explicitly through JDBC (BMP), the database or other vendor specific mechanisms.
- The bean-container contract has been enhanced to include a default JNDI context for accessing properties and resources, for example, JDBC and JMS.

- The basic EJB roles have been expanded and redefined to better separate responsibilities involved in the development, deployment, and hosting of enterprise beans.
- Allows a session bean instance to be removed upon a time-out while the instance is in the passivated state.
- Allows enterprise beans to read system properties.

Migrating 1.0 EJBs to 1.1 EJBs for WebSphere 4.0

If you have existing EJB code developed for an earlier version of WebSphere there are a number of changes in the EJB specification that you should consider.

WebSphere Version 4.0 provides some backwards-compatibility support for EJB 1.0, which means you do not necessarily have to modify your 1.0 EJBs in order to deploy and run them in the new version of WebSphere. You must, however, modify your EJB code to deploy your beans into WebSphere Version 4.0 under the following circumstances:

- An enterprise bean uses the `javax.jts.UserTransaction` interface—it has to be modified to use the new `javax.transaction.UserTransaction` interface
- An entity bean that uses the `UserTransaction` interface (not allowed in 1.1)

We also recommend you make the following changes to your 1.0 EJBs, in order to comply fully with Version 1.1 of the EJB specification:

- EJBs should be modified to use the `getCallerPrincipal()` and `isCallerinRole(String roleName)` methods instead of the deprecated `getCallerIdentity()` and `isCallerInRole(Identity)` methods.
- CMP EJBs should be updated to return the bean’s primary key class from `ejbCreate(...)` methods, instead of `void` as required by the 1.0 specification. Returning the key class enables the creation of bean-managed beans that are subclasses of container-managed beans.
- Entity bean finder methods should be updated to define `FinderException` in their `throws` clauses. EJB 1.1 requires that all finders define the `FinderException`.
- Enterprise beans should no longer throw `java.rmi.RemoteException` from the bean implementation class—this use of the exception is deprecated in EJB 1.1. `RemoteException` must still be defined in EJB home and remote interfaces, as required by RMI.

  The bean implementation class should throw application exceptions where required by the business logic. Unrecoverable system-level errors and other non-business problems should throw a `javax.ejb.EJBException`; this class extends `java.lang.RuntimeException` and does not need to be declared in the throws clause of a 1.1 EJB.

- Enterprise beans in EJB 1.1 must not use the `UserTransaction` interface and implements the `SessionSynchronization` interface at the same time.
- EJB code should be updated to look up other EJBs, data sources, and other resources in JNDI using local JNDI references. See “Using JNDI” on page 326 for more information about using JNDI in WAS 4.0.

Developing 1.1 EJBs in VisualAge for Java

If you want to use VisualAge for Java to develop EJBs that comply with Version 1.1 of the EJB specification, you should be aware of the following factors:

- The VisualAge for Java EJB creation SmartGuide automatically inserts an import of `java.security.Identity` in new EJBs—this is deprecated in JDK 1.3 and causes deprecation warnings if you recompile the code with the JDK used by WebSphere Version 4.0.
- VisualAge for Java does not provide support for the EJB 1.1 feature that allows an entity bean’s key class to be a Java primitive type, such as an `int`.
- The EJB page reports a warning for CMP entity beans that return the key class from `ejbCreate` methods, as required by EJB 1.1.
- The EJB page does not require that every `ejbCreate` method has a matching `ejbPostCreate`, as defined in the EJB 1.1 specification.
- The EJB page reports an error for any entity bean finder method that returns a `java.util.Collection`—the VisualAge for Java runtime only supports `java.util.Enumeration` for multi-object finders.
- The WebSphere Test Environment (WTE) only supports JNDI lookups in the global JNDI namespace (see “Using JNDI” on page 326). This means you cannot develop code using EJB references to locate other EJBs.
- The WTE does not support the WebSphere Version 4.0 EJB security model—it never supported Version 3.x security either, however.
- The WTE includes an earlier implementation of `javax.ejb.EJBException` that does not extend `java.lang.RuntimeException`. We wanted to convert our EJB implementation classes to remove `RemoteException` from the `throws` clause and throw `EJBException` instead, without adding it to the `throws` clause as permitted by EJB 1.1. When we do this VisualAge reports a compile error because the exception is not handled. Declaring the `EJBException` in the `throws` clause only magnifies the problem, because VisualAge for Java then requires all the EJB’s clients to handle the exception.

  We were able to work around this problem by importing the 1.1 `EJBException` class from the WebSphere Version 4.0 `j2ee.jar` file into our workspace. This allowed us to compile and export our code from VisualAge—we must emphasize however that running the WTE with this modified runtime is not a supported configuration.

- VisualAge does not support the new way of defining custom finders for CMP EJBs, where the SQL WHERE clause is specified in a meta-data file.

EJB development and deployment options

Table 11-1 show the different options that are available with VisualAge for Java to develop EJBs for WebSphere Application Server.

Table 11-1 EJB development and deployment

	Visual Age for Java Version 3.5	VisualAge for Java Version 4
WebSphere Version 3.5	Develop according to EJB 1.0. Deploy in either of two ways: ▸ Create an EJB 1.0 JAR file and use WebSphere to deploy the EJBs (generate the mapping and the deployed classes) ▸ **Create an EJB 1.0 deployed JAR file and install the deployed JAR in WebSphere without generating any code. This is recommended, and is required for associations and inheritance.**	Identical to VisualAge for Java Version 3.5. Do not create an EJB 1.1 JAR file (which cannot be deployed in WebSphere Version 3.5)
WebSphere Version 4	Develop according to EJB 1.0. Create an EJB 1.0 JAR file and use WebSphere to deploy the EJBs (generate the mapping and the deployed classes). You can manually edit the mapping information and redeploy the JAR file using the EJB deployment tool (see “EJB deployment tool” on page 418). This is not recommended; we suggest to install VisualAge for Java Version 4.	Develop according to EJB 1.0. Deploy in either of two ways: ▸ Create an EJB 1.0 JAR file and use WebSphere to deploy the EJBs (generate the mapping and the deployed classes). This is not recommended. ▸ **Create an EJB 1.1JAR file (with mapping information) and use WebSphere to deploy the JAR file using the EJBDeploy tool.** Develop according to EJB 1.1 (which means that you cannot test the EJBs) and create an EJB 1.1 JAR file and deploy it in WebSphere.

See “Assembling the application” on page 389 for information about conversion and deployment of EJBs using the Application Assembly Tool and the EJB deployment tool, as well as information about CMP persistence mapping in WebSphere Version 4.0.

WebSphere Test Environment

The WebSphere Test Environment (WTE) in VisualAge for Java 4.0 is a lightweight version of the WebSphere Application Server Version 3.5.3 Advanced Edition (so the specification level is different from WebSphere Version 4.0).

While WAS 4.0 supports EJB 1.1, VisualAge for Java 4.0 only supports EJB 1.0. The specification levels for servlets and JSPs are the same: servlet 2.2 and JSP 1.1. The JSP specification levels 0.91, 1.0, and 1.1 are supported in VisualAge for Java 4.0 and the WTE can be configured to use any of them—we recommend however, that you keep the default 1.1 JSP level.

WTE contains the runtime environment for the application server, and it is intended for unit testing purposes. With this tool, the developer can test his/her code without exporting it from the VisualAge for Java repository.

Configuration

The WTE allows a single server configuration with multiple Web applications. The configuration files for the WTE are located in the project resources directory:

`D:\IBMVJava\ide\project_resources\`**`IBM WebSphere Test Environment`**`\`

To setup the configuration for a Web application, we edit the following file:

```
...\IBM WebSphere Test Environment\properties\default.servlet_engine
```

We also have to create a `webapp_name.webapp` for each Web application we are going to test with the WTE. This files resides in:

```
...\IBM WebSphere Test Environment\hosts\default_host\webapp_name\servlets
```

The WTE has one Web application preconfigured, the default application (`default_app`).

Configuring a Web application

The file `default.servlet_engine` is an XML file containing information about the Web applications (document root, class path, virtual hosts and MIME types).

For example, to add a new Web application to the WTE configuration, we have to add the following tags to the file (Figure 11-4, see bold lines).

```
<?xml version="1.0"?>
<websphere-servlet-engine name="servletEngine">
   <active-transport>http</active-transport>
   <transport>
      <name>http</name>
      <code>com.ibm.servlet.engine.http_transport.HttpTransport</code>
      <arg name="port" value="8080"/>
      <arg name="maxConcurrency" value="50"/>
      <arg name="server_root" value="$server_root$"/>
   </transport>
   <websphere-servlet-host name="default_host">
    <websphere-webgroup name="default_app">
       <description>Default WebGroup</description>
       <document-root>$approot$/web</document-root>
       <classpath>$approot$/servlets$psep$$server_root$/servlets</classpath>
       <root-uri>/</root-uri>
       <auto-reload enabled="true" polling-interval="3000"/>
       <shared-context>false</shared-context>
    </websphere-webgroup>
    <websphere-webgroup name="PiggyBank">
       <description>PiggyBank</description>
       <document-root>$approot$</document-root>
       <classpath>$approot$</classpath>
       <root-uri>/</root-uri>
       <auto-reload enabled="true" polling-interval="3000"/>
       <shared-context>false</shared-context>
    </websphere-webgroup>
      <mime type="application/SLA">
         <ext>STL</ext>
         <ext>stl</ext>
      </mime>
<!-- ... (other MIME types) -->
</websphere-servlet-host>
   <hostname-binding hostname="localhost:8080" servlethost="default_host"/>
   <hostname-binding hostname="127.0.0.1:8080" servlethost="default_host"/>
</websphere-servlet-engine>
```

Figure 11-4 Configuring Web applications in the WTE

For each Web application, we create a new directory under

```
...\IBM WebSphere Test Environment\hosts\default_host\
```

Each of these directories contains a folder named `servlets` (to place the `webapp_name.webapp` file and the servlets' class files), and a `web` folder (to place the JSPs, HTML pages, images).

To create the Web application configuration file (`webapp_name.webapp`), we can use the `default_app.webapp` file as a template.

Here is a list of the main tags we might need to include for our Web applications:

- *Error page*—to specify the general error page for the application.
- *Servlet properties*—name, Web path and initial parameters for servlets in the Web application.
- *Invoker servlet*—we might want to include data for this servlet, that lets us load classes by class name (`/servlet/pkgname.class`).
- *JSP specification level*—VisualAge for Java supports the three JSP specifications (0.91, 1.0 and 1.1). The WTE lets us switch the specification level by selecting the appropriate class name for the servlet that processes the JSPs (Figure 11-5).

```
...
<servlet>
   <name>jsp</name>
   <description>JSP support servlet</description>

   <!--
      ***
      *** Replace the JSP compiler with the required specification level.
      ***

      *** JSP 0.91 Compiler ***
<code>com.ibm.ivj.jsp.debugger.pagecompile.IBMPageCompileServlet</code>

      *** JSP 1.0 Compiler ***
      <code>com.ibm.ivj.jsp.runtime.JspDebugServlet</code>

      *** JSP 1.1 Compiler ***
      <code>com.ibm.ivj.jsp.jasper.runtime.JspDebugServlet</code>
   -->
...
   <code>com.ibm.ivj.jsp.jasper.runtime.JspDebugServlet</code>
   <init-parameter>
      <name>scratchdir</name>
      <value>$server_root$/temp/JSP1_1/default_app</value>
   </init-parameter>
...
```

Figure 11-5 Switching the JSP configuration level (default_webapp.webapp)

Figure 11-6 shows an example of a `webapp` configuration file for the PiggyBank application.

```
<?xml version="1.0"?>
<webapp>
   <name>PiggyBank</name>
   <description>PiggyBank Application</description>
   <error-page>/error.jsp</error-page>
   <servlet>
      <name>ControllerServlet</name>
      <description>Controller servlet for PiggyBank</description>
      <code>itso.was4ad.webapp.controller.ControllerServlet</code>
      <servlet-path>*.pbc</servlet-path>
      <init-parameter>
         <name></name>
         <value></value>
      </init-parameter>
      <autostart>false</autostart>
   </servlet>

   <servlet>
      <name>jsp</name>
      <description>JSP support servlet</description>
      <!--*** JSP 1.1 Compiler ***-->
      <code>com.ibm.ivj.jsp.jasper.runtime.JspDebugServlet</code>
      <init-parameter>
         <name>workingDir</name>
         <value>$server_root$/temp/default_app</value>
      </init-parameter>
      <init-parameter>
         <name>jspemEnabled</name>
         <value>true</value>
      </init-parameter>
      <init-parameter>
         <name>scratchdir</name>
         <value>$server_root$/temp/JSP1_1/default_app</value>
      </init-parameter>
      <init-parameter>
         <name>keepgenerated</name>
         <value>true</value>
      </init-parameter>
      <autostart>true</autostart>
      <servlet-path>*.jsp</servlet-path>
   </servlet>
</webapp>
```

Figure 11-6 Writing the Web application's configuration file for WTE

The WTE provides session management functions as well, and the configuration parameters are setup in:

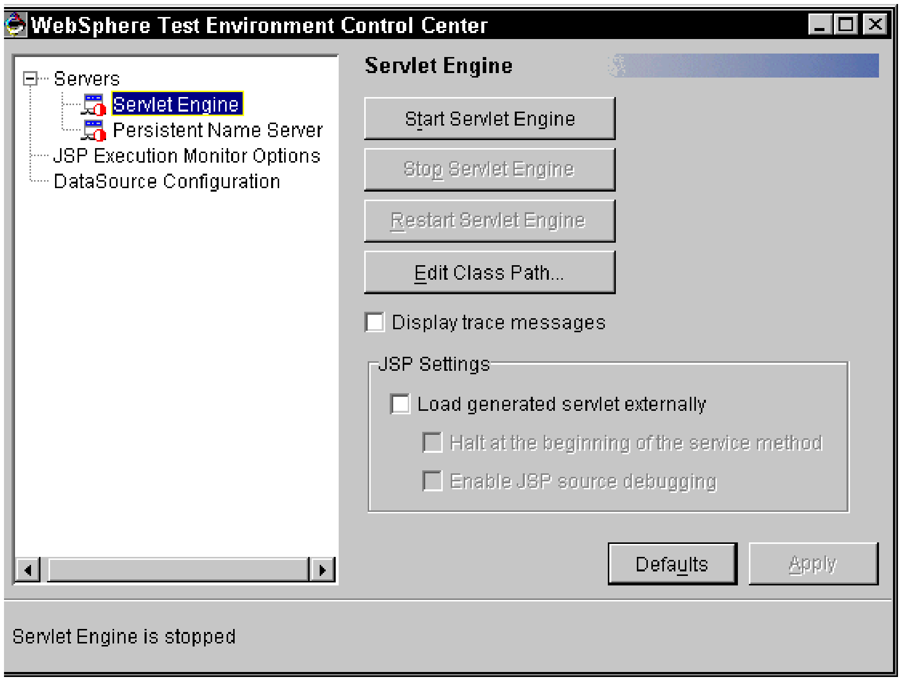
```
...\IBM WebSphere Test Environment\properties\session.xml
```

The tags within `<session-data>` are the tags used by the WTE (the rest should be ignored).

WebSphere Test Environment Control Center

The WTE is controlled through the WebSphere Test Environment Control Center.

The WebSphere Test Environment Control Center is launched by selecting *Workspace -> Tools -> WebSphere Test Environment* (Figure 11-7).



Display trace messages

JSP Settings

Load generated servlet externally

Halt at the beginning of the service method

Enable JSP source debugging

Servlet Engine is stopped

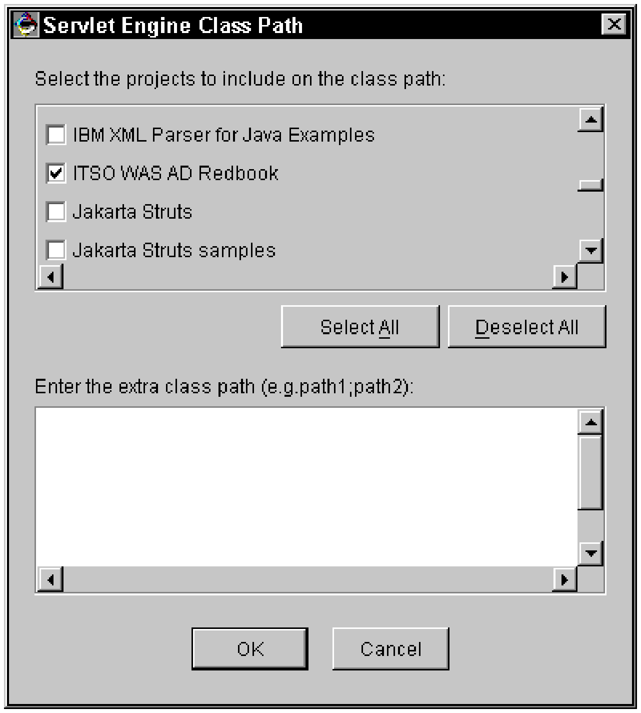
Figure 11-7 WebSphere Test Environment Control Center

Let's take now a closer look at each of the components of the WTE.

Servlet Engine

The Servlet Engine is used to run servlets and JSPs in the WebSphere Test Environment. We can run code from the Workspace (either servlets or compiled JSPs) or from external directories.

To run a Web application, we have to set up the class path for the Servlet Engine (Figure 11-8). We select the projects containing classes used by the servlets we are going to test. System projects required to run the Servlet Engine (for example, the IBM WebSphere Test Environment project) are added automatically.


Servlet Engine Class Path

Select the projects to include on the class path:

- [ ] IBM XML Parser for Java Examples
- [✓] ITSO WAS AD Redbook
- [ ] Jakarta Struts
- [ ] Jakarta Struts samples

Enter the extra class path (e.g.path1;path2):

Figure 11-8 Setting up the Servlet Engine class path

The Servlet Engine must be started/restarted after any changes to the class path or other settings are made.

The JSP and HTML pages, as well as the images and other static Web content are placed in the directory (or subdirectory):

```
...\IBM WebSphere Test Environment\default_host\webappname\web
```

If we are using servlets that are not in the repository, we place them under

```
...\IBM WebSphere Test Environment\default_host\webappname\servlets
```

We also place in this directory the servlet configuration files (`*.servlet`).

To invoke the components in the browser, we use the virtual hosts defined in the `default.servlet_engine` file (see an example in Figure 11-4 on page 271):

```
http://localhost:8080/index.html
```

If we select the option *Display trace messages*, the console shows a detailed output of the configuration parameters and servlets that are loaded from the `*.webapp` files. When we select this option, we have to click on *Apply* and restart the Servlet Engine for the changes to take effect.

JSP processing

JSPs are translated into a servlet by the JSP processor (for the specification level that we have selected, see for example Figure 11-5 on page 272). The default settings for the Servlet Engine make the generated code to be imported in the VisualAge repository under the project `JSP Page Compile Generated Code`.

However, the generated code is not imported if there are compilation errors in the JSP. In this case we get error messages, but it might be difficult to figure out where the problem is. We can try to run the code in the Scrapbook to get better understandable error messages, or we can select to *Load the generated servlet externally* in the Servlet Engine window, so that the compiled Java classes are stored in (when using the 1.1 specification level):

```
...\IBM WebSphere Test Environment\temp\JSP1_1\web_app_name\etc
```

Then we can import the code manually to inspect and debug it. See more about debugging JSPs in “A special case: how to debug a JSP” on page 511.

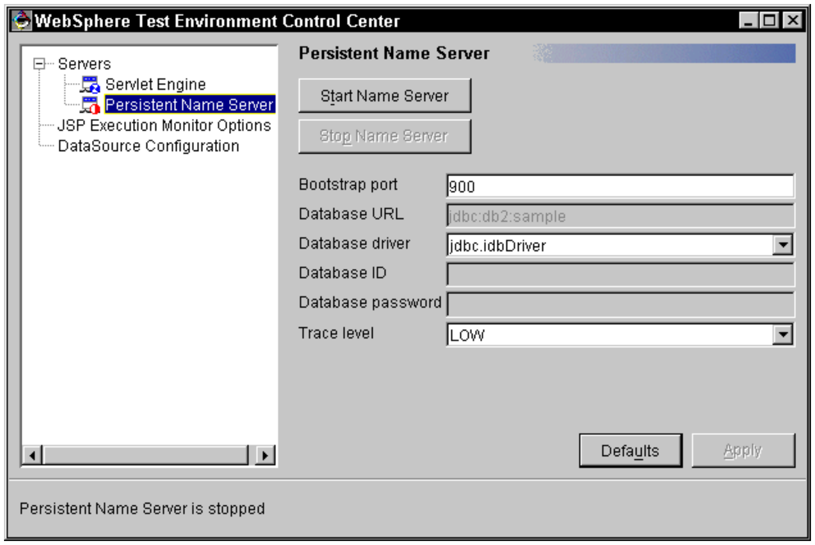
When using the option of loading the generated servlet externally, we can also select:

- *Halt at the beginning of the service method*—which acts like a breakpoint set at the service method of the generated servlet
- *Enable JSP source debugging*—which brings up the VisualAge for Java debugger with the JSP source code. We then can step through the JSP source, but there are some restrictions, for example, we cannot step into Java code embedded in the JSP.

Persistent Name Server

The Persistent Name Server (PNS) is used to configure and access DataSource objects, as well as to test EJBs.

DataSources and EJBs are bound to a context. The Persistent Name Server gives access to this context to perform JNDI operations. When an object is bound to its JNDI name, a description of the object is stored in the database specified in the PNS properties (Figure 11-9).



Bootstrap port: 900
Database URL: jdbc:db2:sample
Database driver: jdbc.idbDriver
Database ID:
Database password:
Trace level: LOW

Persistent Name Server is stopped

Figure 11-9 Persistent Name Server in the WTE

The parameters that we configure before starting the Persistent Name Server are the following:

- Bootstrap port—is the port used to lookup EJB homes and DataSources. The default value of 900 is also used by the WebSphere Application Server, so you have to use another port or stop WAS if WAS has been started on the same machine.
- Database URL—the JDBC URL if a relational database is used for storage.
- Database driver—the JDBC driver used to access the database. The default driver corresponds to InstantDB (a relational database simulated in files), which is recommended for simple configurations.
- Database ID and password—used to connect to a real relational database.
- Trace level—specifies the level (high/medium/low) of the trace information that is displayed through the console.

Using DataSource objects with the WTE

To configure and use DataSources, we have to start first the Persistent Name Server. Once this is done, the WTE interface provides a *DataSource Configuration* screen that allows us to add new DataSource objects (Figure 11-10).

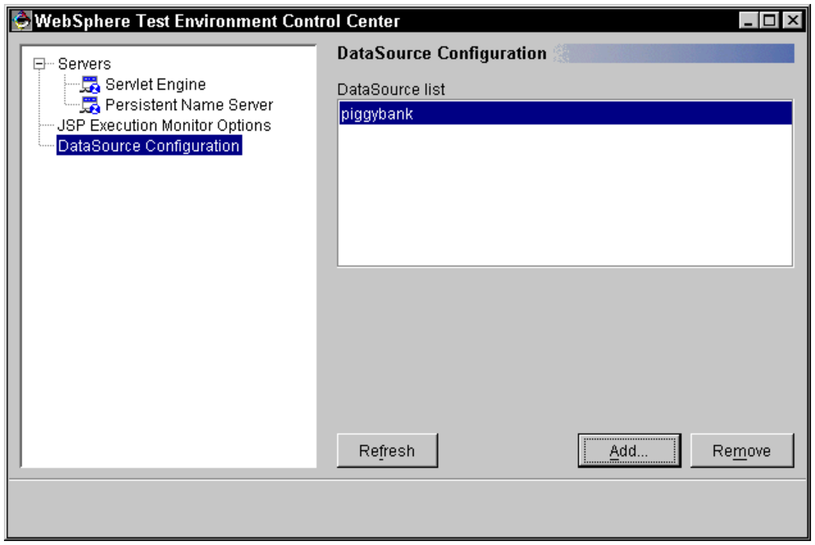


Figure 11-10 Using DataSources in the WTE

When the Persistent Name Server is started, it retrieves the list of DataSources configured previously (if any). To add a new DataSource object, we have to configure the following parameters (Figure 11-11):

- Name—the name used to perform the lookup (with the prefix `jdbc/`). In our example, the JNDI lookup would be through the name `jdbc/piggybank`.

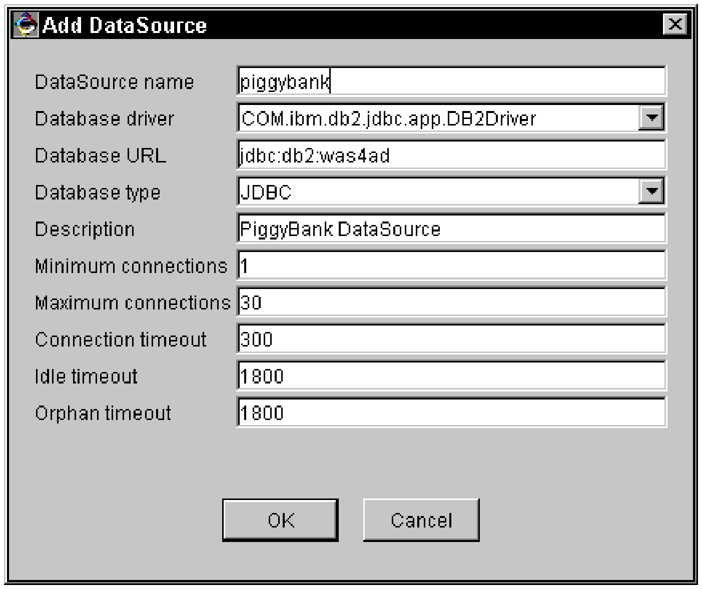
> **Note:** Because VisualAge for Java includes a WAS Version 3.5.3 runtime, the WTE only supports the .use of global JNDI names, such as `jdbc/piggybank`. The WTE does not support local JNDI references such as, for example, `java:comp/env/jdbc/piggybank`.
>
> See “Using JNDI” on page 326 for more information on JNDI in WAS 4.0.

- Driver—the JDBC driver used to connect with the relational database

> **Note:** The list of driver classes for datasources differs between the WTE (based on a WAS 3.5.3 runtime) and WAS Version 4.0.

- URL—the URL that matches the JDBC driver
- Type—JDBC or JTA (two-phase commit)
- Description—optional
- Connection and timeout parameters—we recommend to use the default values (they are suitable for unit testing in most of the cases)

Figure 11-11 shows the DataSource configuration window.



DataSource name	piggybank
Database driver	COM.ibm.db2.jdbc.app.DB2Driver
Database URL	jdbc:db2:was4ad
Database type	JDBC
Description	PiggyBank DataSource
Minimum connections	1
Maximum connections	30
Connection timeout	300
Idle timeout	1800
Orphan timeout	1800

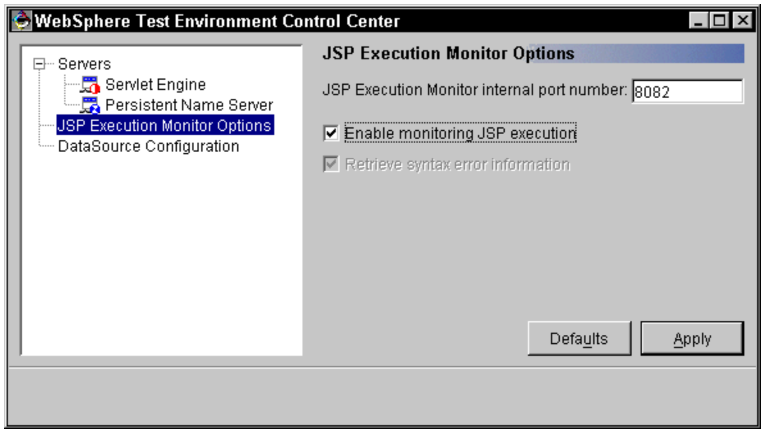
Figure 11-11 Adding a DataSource

JSP execution monitor

The JSP execution monitor is used to monitor the execution of JSP files. We can view both the JSP source code, the generated servlet code and the HTML output, and execute the code step by step or in combination with the VisualAge for Java debugger.

The JSP execution monitor is activated through the WebSphere Test Environment Control Center (Figure 11-12).

More details about the usage of this feature are described in Chapter 17, “Debugging the application” on page 467, in the section “A special case: how to debug a JSP” on page 511.


WebSphere Test Environment Control Center
Servers
Servlet Engine
Persistent Name Server
JSP Execution Monitor Options
DataSource Configuration
JSP Execution Monitor Options
Defaults
Apply

JSP Execution Monitor internal port number: 8082

[✓] Enable monitoring JSP execution

[✓] Retrieve syntax error information

Figure 11-12 Enabling the JSP execution monitor

Exporting the code

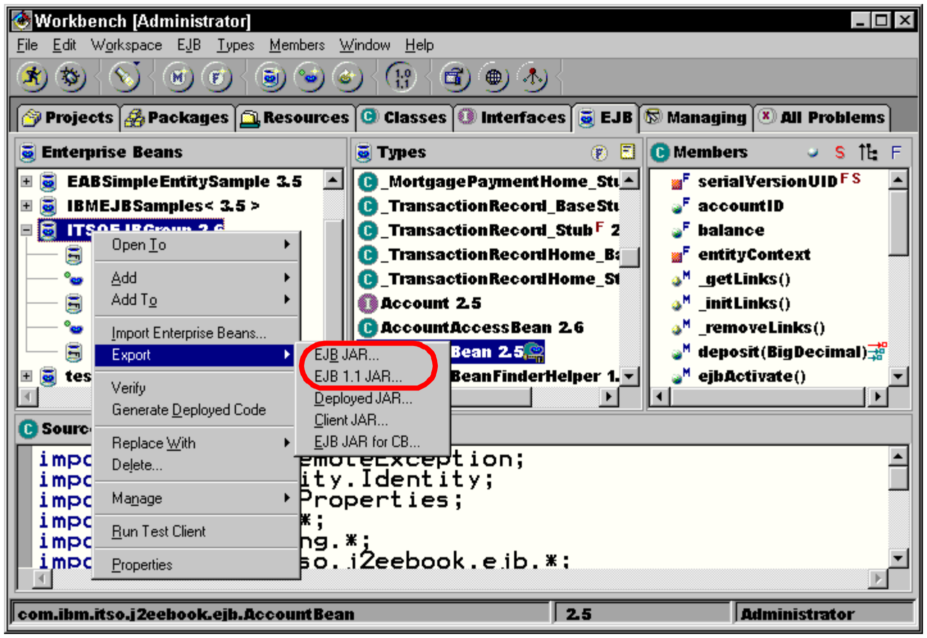
After testing Web applications in VisualAge for Java, the code can be exported in a number of ways tor deployment to WebSphere Application Server:

- Export to a directory—Java source code and compiled class files can be exported into a directory structure. Subdirectories are generated according to the package names.
- Export to JAR file—Java source code and compiled class files can be exported into a JAR file that can be added to the class path for execution.
- Export in repository format—This is not for deployment but for archiving or to move code from one VisualAge for Java system to another. Exporting a project also exports the project resources into a directory structure in the target location (where the repository .dat file is created).
- Exporting the resources—Resource files used by the application can be copied for the project's resources directory to another location.
- Exporting EJBs—VisualAge for Java can create an EJB JAR file that can be deployed in WebSphere. The generated JAR file can be an EJB 1.0 JAR file or an EJB 1.1 JAR file (this is really the only new feature of VisualAge for Java Version 4.0).

Exporting the EJB code

For the EJB code, we have multiple possibilities in VisualAge 4.0 (Figure 11-13):

- Export to an EJB 1.0 JAR file—such a file must be deployed in WAS.
- Export to an EJB 1.1 JAR file—such a file must be deployed in WAS.
- Export to a deployed JAR file that can be installed in WAS Version 3.5.
- Export to an EJB JAR file for CB (Component Broker).



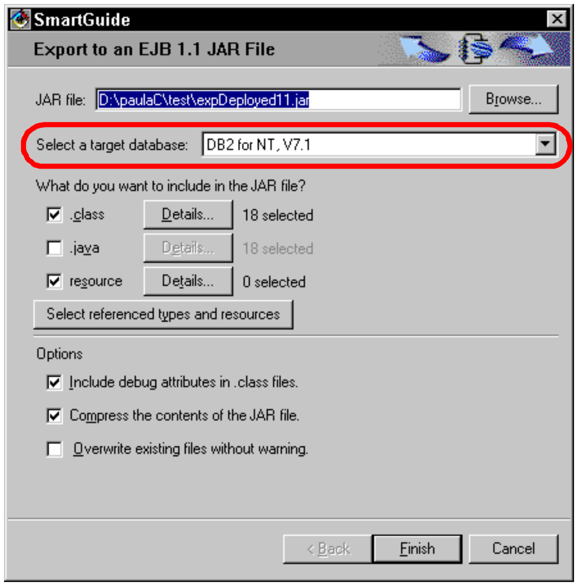
```
impo          emoteException;
impo          ity.Identity;
impo          roperties;
impo          *;
impo          ng.*;
impo          so.j2eebook.ejb.*;
```

Figure 11-13 Export options for EJBs for WebSphere Version 4.0

If we select the first option (*EJB JAR*), when we add our EJBs to the enterprise application, it is necessary to convert the 1.0 file to a 1.1 file (WebSphere 4.0 no longer supports the 1.0 specification). This process is explained in Chapter 15, “Assembling the application” on page 389 in *Creating an EJB module*.

Exporting EJBs to a 1.1 JAR file

When exporting to a 1.1 JAR file (Figure 11-14) you have to specify the target database so that the schema and map files that define the mapping of CMP fields to your database schema can be generated. See “Customizing CMP persistence mapping” on page 420 for more information about these files.


SmartGuide

Export to an EJB 1.1 JAR File

JAR file: D:\paulaC\test\expDeployed11.jar

Select a target database: DB2 for NT, V7.1

What do you want to include in the JAR file?

[✓] .class — 18 selected

[ ] .java — 18 selected

[✓] resource — 0 selected

Options

[✓] Include debug attributes in .class files.

[✓] Compress the contents of the JAR file.

[ ] Overwrite existing files without warning.

Figure 11-14 Exporting the EJBs to a 1.1 JAR file

EJB deployment tool

An enhanced version of the `EJBDeploy` tool shipped with the first release of WebSphere AEs is available for download from VisualAge Developer Domain. This tool can handle all the extra information generated into the EJB 1.1 JAR file by VisualAge. The enhanced tool will be included in the first AE release, and upgraded in a refresh of the AEs product.

See "EJB deployment tool" on page 418 for more information.

Debugging in VisualAge for Java

See "Debugging with VisualAge for Java Version 4.0" on page 468 for detailed information about debugging.

12 Development with frameworks

This chapter discusses how to develop code for a WebSphere application using some available frameworks.

The frameworks we discuss are:

- Jakarta Struts
- IBM WebSphere Business Components Composer (WSBCC)

The intended reader of this chapter is more interested by the technical aspects of the frameworks, especially hands-on implementation. It is therefore useful to Java and Web developers as well as to application designers who are curious to know how the code to be implemented actually works.

See Chapter 7, “Designing with frameworks” on page 153 for introductions to the frameworks and information about designing applications to use them.

Jakarta Struts

This section explains how to develop a version of the the PiggyBank Web application using Struts. The example code we develop in this section is available as part of the additional material for this Redbook—see Appendix A, “Additional material” on page 557 for information on how to obtain and use the additional material.

The Struts version of our Web application completely replaces the Web application module included in the base version of the example PiggyBank application—the two versions of the Web module are completely interchangeable.

We used release Version 1.0b1 of Struts to develop these examples. The binary and source distributions can be downloaded from the Jakarta Web site:

`http://jakarta.apache.org/builds/jakarta-struts/`

Note: The final Struts Version 1.0 was released shortly before work on this book was completed. Due to time constraints, however, we were unable to test our example code with this final release.

Using Struts in your development environment

Before we discuss how to implement the Web application in Struts, we describe how to configure your environment to develop, build and test the Struts version of our example PiggyBank application. We describe how you can integrate Struts with the following tools described in the this redbook:

- VisualAge for Java and the WebSphere Test Environment, described in Chapter 11, “Development using VisualAge for Java” on page 259
- WebSphere Studio, described in Chapter 10, “Development using WebSphere Studio” on page 237
- Ant and the Java 2 SDK, described in Chapter 9, “Development using the Java 2 Software Development Kit” on page 183

As you read through we advise you to select information from the sections that follow depending on your environment and the tools available to you.

Importing Struts into VisualAge for Java

If you want to develop applications using Struts in VisualAge for Java, we recommend you import the Struts source files into your workspace, rather than importing the compiled .class files. This allows you to view and step through the framework source when coding and debugging with VisualAge for Java.

The Struts source files are distributed under the form of plain Java files in the Struts source distribution's `src\share` directory. To import these files into VisualAge for Java expand the source distribution into a local directory—our example uses `D:\jakarta-struts`—then follow these steps:

- Create a new project in VisualAge for Java, for example `Jakarta Struts`.
- Right-click on the `Jakarta Struts` project and select *Import*. In the import SmartGuide, select the *Directory* radio button. Click *Next*.
- Browse to the `D:\jakarta-struts\src\share\` directory. Choose to import the .java and resources files. Leave the options unchecked. Click *Finish*.
- Right-click on the project. Select *Manage -> Version*. Select *One Name* and enter `1.0b1 with source`. This will version the imported packages and classes recursively.

Struts uses the JAXP APIs in the `javax.xml.parsers` package, which are not provided by the standard XML parser in the IBM XML Parser for Java project included with VisualAge for Java. Later on, when we configure the WebSphere Test Environment (see "Setting up the WebSphere Test Environment for Struts" on page 287) we also require a stylesheet processor. We solve both problems by downloading and importing Xalan, a stylesheet processor from Apache that also includes the Xerces XML parser, and is available from:

`http://xml.apache.org/xalan-j/`

Importing the projects was a little tricky, because some of the classes in the Xerces JAR conflict with classes in the `IBM XML Parser for Java` project used by the WebSphere Test Environment.

We managed to get everything to work successfully by creating a new edition of the IBM XML Parser for Java project, importing the following packages from the Xerces JAR file:

- `org.w3c.dom`
- `org.xml.sax`
- `org.xml.sax.helpers`

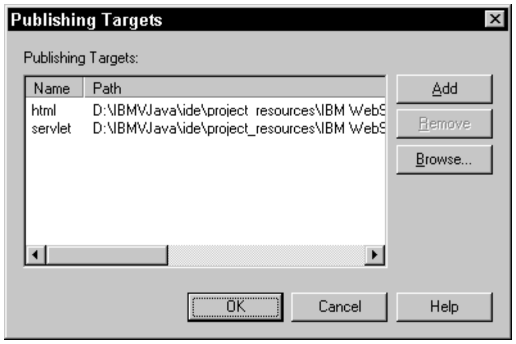
We then created a separate project for the rest of the Xerces classes and the Xalan code.

Setting up WebSphere Studio for Struts

We used WebSphere Studio to modify our Web content to work with Struts. We created a new Studio project to manage our files. To make it easier to test our changes in VisualAge for Java, we set up a publishing target that published the files from Studio to the directory used by the VisualAge WebSphere Test Environment.

In the *Publishing Targets* configuration (Figure 12-1), we set the html and servlet publishing targets to:

```
D:\IBMVJava\ide\project_resources\IBM WebSphere Test Environment\
               hosts\default_host\piggybank-struts\web        <==== html
               hosts\default_host\piggybank-struts\servlets   <=== servlet
```



Publishing Targets:

Name	Path
html	D:\IBMVJava\ide\project_resources\IBM WebS
servlet	D:\IBMVJava\ide\project_resources\IBM WebS

Figure 12-1 Studio publishing targets configuration

Attention: Any change on any file should be checked in and published from Studio before being tested in the WebSphere Test Environment. If we do not publish the files, VisualAge will not pick up the changes. For more information on Studio checkin and publishing mechanism, see Chapter 10, “Development using WebSphere Studio” on page 237.

Populating the Studio project

Next we inserted the following files from the original PiggyBank application into our Studio project:

- `index.html`
- `welcome.jsp`
- `include.jsp`

The `loginfail.jsp` is not required as Struts provides an appropriate error reporting that can be combined with the login page.

The `index.html` file displays a HTML login form. To benefit from the Struts input form facility, this file is changed to a JSP file. Because an HTML file is a particular case of JSP with no JSP tag, changing its extension to `.jsp` is initially the only change we make to the file. We must also remember to update the Web application’s welcome file list to include `index.jsp` so that the file name does not have to be entered explicitly into the browser.

Next we create a `WEB-INF` directory in our Studio project and insert these files from the Struts distribution:

- `struts-html.tld`
- `struts-bean.tld`
- `struts-logic.tld`
- `struts-config.xml`

Finally we also import the `images` and `themes` directories from the original PiggyBank application.

Setting up the WebSphere Test Environment for Struts

If we want to test the Struts version of the PiggyBank alongside existing Web applications, the WebSphere Test Environment must be configured to support a new Web application. To do this we edit the file:

```
D:\IBMVJava\ide\project_resources\IBM Websphere Test Environment\
                    properties\default.servlet_engine file
```

We add the following declaration of the `piggybank-struts` Web application between the `<websphere-servlet-host name="default_host">` tags:

```
<websphere-webgroup name="piggybank-struts">
   <description>PiggyBank application using Struts</description>
   <document-root>$approot$</document-root>
   <classpath>$approot$</classpath>
   <root-uri>/piggybank-struts</root-uri>
   <auto-reload enabled="true" polling-interval="3000"/>
   <shared-context>false</shared-context>
</websphere-webgroup>
```

This block redirects all the URI starting with `/piggybank-struts` to the Piggybank-Struts Web application, which is configured in the configuration file:

```
D:\IBMVJava\ide\project_resources\IBM Websphere Test Environment\
      hosts\default_hosts\piggybank-struts\servlets\piggybank-struts.webapp
```

The `webapp` file is very generic and can be generated from the default `web.xml` files included in the Struts distribution.

Generating the WTE webapp file from a web.xml file

The standard J2EE `web.xml` files are similar to the `.webapp` file VisualAge for Java uses. Because both are written in XML, it is possible to convert a `web.xml` file into a `.webapp` file using an XSL stylesheet, as represented in Figure 12-2.

Such a stylesheet is provided with VisualAge for Java in the file:

```
D:\IBMVJava\ide\project_resources\IBM Websphere Test Environment\
         properties\webapp.xsl
```

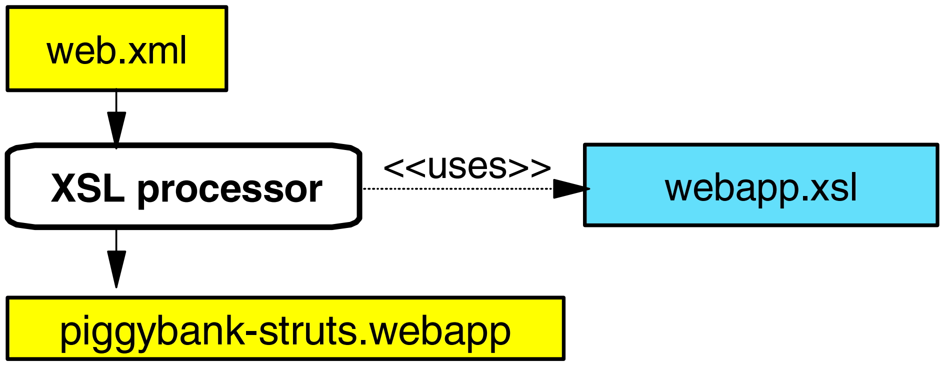


Figure 12-2 Converting a web.xml file into a .webapp file using XSL

To apply a stylesheet to an XML, we need a stylesheet processor program such as Xalan, which we downloaded earlier.

To run the processor go to the `org.apache.xalan.xslt.Process` class, right-click on it and select *Run -> Run main with*. In the *Class Path* tab, check the Project path check box and click on *Compute Now*. In the *Program* tab, enter this in *Command line arguments*:

```
-IN C:\temp\web.xml -xsl D:\IBMVJava\ide\project_resources\IBM Websphere
Test Environment\properties\webapp.xsl -OUT C:\temp\piggybank-struts.webapp
```

Make sure the `web.xml` file to convert is in the `C:\temp` directory and click *OK*. In this case we want to use the example `web.xml` file shipped with Struts. Check the console to watch any error messages. An empty console for that program indicates a successful conversion. The converted `piggybank-struts.webapp` can be found in the same `C:\temp` directory.

To run Xalan outside of VisualAge for Java, we recommend you use Xerces as an XML parser. It is also possible to (export and) use the IBM XML Parser or any other JAXP-compliant XML parser. In most versions of the Xalan distribution, a compatible version of Xerces is included. To run the same command from the command line copy both `xalan.jar` and `xerces.jar` files into the `C:\temp\` directory, open a Command Prompt window and type the following command line:

```
D:\Websphere\AppServer\java\bin\java
-cp C:\temp\xalan.jar;C:\temp\xerces.jar org.apache.xalan.xslt.Process
-IN C:\temp\web.xml
-xsl "D:\IBMVJava\ide\project_resources\IBM Websphere Test Environment\
                                        properties\webapp.xsl"
-OUT C:\temp\struts-example.webapp
```

Finally, the generated `.webapp` file has to be tweaked a little bit to properly support JSP compilation in the WebSphere Test Environment. After:

```
<servlet>
<name>jsp11</name>
```

Add these lines:

```
   <init-parameter>
      <name>jspemEnabled</name>
      <value>true</value>
   </init-parameter>
```

The resulting `.webapp` file should be put in the corresponding class path, which for PiggyBank-Struts is:

```
D:\IBMVJava\ide\project_resources\IBM Websphere Test Environment\
                hosts\default_hosts\piggybank-struts\servlets
```

The `.webapp` file can be managed and published from Studio.

Building the Struts version of the application using Ant

The Struts example code in the additional material includes modified versions of the Ant build scripts described in Chapter 9, "Development using the Java 2 Software Development Kit" on page 183. These scripts have been modified as follows:

- Struts software locations added to `global.properties` file
- Web application compile class path updated to include the Struts JAR
- Web application WAR file now includes the Struts JAR file and the Struts tag libraries used by the example code

We must also update the Web application Web content and remove the existing Web application code from the source tree—the Struts code completely replaces the basic Web application code.

Updating the global properties file

The following entries were added to the `global.properties` file:

```
global.struts.dir=D:/jakarta-struts
global.struts.jar=${global.struts.dir}/lib/struts.jar
```

These entries assume the Struts binary distribution has been extracted into `D:/jakarta-struts`.

Updating the Web application class path

The class path in the Web application build file `build.xml` was updated with:

```
<path id="webapp.classpath">
  <pathelement location="${global.was.dir}/lib/j2ee.jar"/>
  <pathelement location="${global.struts.jar}"/>
  <pathelement path="${global.build.dir}/common"/>
  <pathelement path="${global.build.dir}/usecase"/>
</path>
```

This change is required to compile the Web application code that uses Struts.

Adding Struts components to the WAR file

We updated the Web application package target in the Web application build file `build.xml` as shown in Figure 12-3.

```
<target name="package" depends="init,compile">
   <echo>Packaging ${ant.project.name}</echo>
   <mkdir dir="${global.module.dir}"/>
   <war warfile="${webapp.war.file}"
        webxml="WEB-INF/web.xml"
        basedir="web"
        manifest="META-INF/MANIFEST.MF"
   >
     <classes dir="${webapp.build.dir}"/>
     <!-- Pick up the reource files -->
     <classes dir="${basedir}">
       <include name="**/*.properties"/>
     </classes>
     <webinf dir="WEB-INF">
       <exclude name="web.xml"/>
     </webinf>
     <!-- Include the struts TLDs -->
     <webinf dir="${global.struts.dir}/lib">
       <include name="struts-bean.tld"/>
       <include name="struts-html.tld"/>
       <include name="struts-logic.tld"/>
     </webinf>
     <!-- Include the struts jar -->
     <lib dir="${global.struts.dir}/lib">
       <include name="struts.jar"/>
     </lib>
   </war>
   <echo>Finished packaging ${ant.project.name}</echo>
 </target>
```

Figure 12-3 Struts updates to the Ant Web application package target

The additions to the target are highlighted in bold. The Struts tag libraries are added to the WAR file `WEB-INF` directory, and the Struts runtime JAR to the `WEB-INF/lib` directory. We also include properties files from the source tree—we add these later to enable message externalization and internationalization.

Updating the PiggyBank Web content

We use the original PiggyBank Web content as a basis for our Struts version of the PiggyBank. Before we start, however, there are a few changes we have to make.

First, we must rename the `index.html` page to `index.jsp`—this is required because we are introducing JSP tags into our index page.

We also have to update our Web application deployment descriptor, `web.xml`. We update the welcome file list to use the index JSP as the home page for our application instead of the old HTML. The new welcome file list looks like this:

```
<welcome-file-list>
  <welcome-file>index.jsp</welcome-file>
</welcome-file-list>
```

We remove the PiggyBank controller servlet and add the Struts action servlet in its place:

```
<servlet>
    <servlet-name>action</servlet-name>
    <servlet-class>org.apache.struts.action.ActionServlet</servlet-class>
    <init-param>
      <param-name>application</param-name>
      <param-value>PiggyBankResources</param-value>
    </init-param>
    <init-param>
      <param-name>config</param-name>
      <param-value>/WEB-INF/struts-config.xml</param-value>
    </init-param>
    <init-param>
      <param-name>debug</param-name>
      <param-value>2</param-value>
    </init-param>
    <init-param>
      <param-name>detail</param-name>
      <param-value>2</param-value>
    </init-param>
    <init-param>
      <param-name>validate</param-name>
      <param-value>true</param-value>
    </init-param>
    <load-on-startup>2</load-on-startup>
</servlet>
```

The last update adds the Struts tag library descriptors:

```
<taglib>
  <taglib-uri>/WEB-INF/struts-bean.tld</taglib-uri>
  <taglib-location>/WEB-INF/struts-bean.tld</taglib-location>
</taglib>
<taglib>
  <taglib-uri>/WEB-INF/struts-html.tld</taglib-uri>
  <taglib-location>/WEB-INF/struts-html.tld</taglib-location>
</taglib>
<taglib>
  <taglib-uri>/WEB-INF/struts-logic.tld</taglib-uri>
  <taglib-location>/WEB-INF/struts-logic.tld</taglib-location>
</taglib>
```

Finally we must add the Struts configuration file for our application. We start by using the basic `struts-config.xml` described in “Struts configuration file” below. We place the configuration file in the `WEB-INF` directory, the location referenced in the Struts action servlet initialization parameter.

Struts configuration file

Struts configuration is described in an XML file. A minimal `struts-config.xml` configuration file looks like this:

```
<?xml version="1.0" encoding="ISO-8859-1" ?>
<!DOCTYPE struts-config PUBLIC
          "-//Apache Software Foundation//DTD Struts Configuration 1.0//EN"
          "http://jakarta.apache.org/struts/dtds/struts-config_1_0.dtd">
<struts-config>
<!-- Here come the application elements declarations -->
</struts-config>
```

We will enhance this file throughout this chapter to have PiggyBank-Struts working.

Building a Struts form

We start from the existing PiggyBank JSPs.

Every HTML form element can be rewritten using the Struts custom tags library. Table 12-1 shows some mappings between the normal HTML tags and the Struts custom tags. The Struts documentation provides a more complete and detailed list.

Table 12-1 Struts custom tags mapping with HTML form tags

HTML form tag	Struts custom tag
`<FORM ACTION="/xxx" METHOD="POST">`	`<html:form action="xxx.do">`
`<INPUT TYPE=”text” NAME=”name”>`	`<html:text property=”name”/>`
`<INPUT TYPE=”submit”>`	`<html:submit/>`
`</FORM>`	`</html:form>`

Each time a form is defined in a JSP, a corresponding form bean should be defined in VisualAge for Java. For example, the Struts `index.jsp` form:

```
<html:form action="/login.do" focus="user">
  <html:text property="user" size="20" maxlength="20"/>
  <html:password property="password" size="20" maxlength="20"
               redisplay="false"/>
  <html:submit property="submit" value="Login"/>
  <html:reset/>
</html:form>
```

This should have a counterpart form bean implemented in Java:

- Create a new package in the existing PiggyBank project for our form beans and name it `itso.was4ad.webapp.form`
- Create a new class in this package called `LoginForm` that extends `org.apache.struts.action.ActionForm`
- Add a private `user` field to the `LoginForm` class—make the field a `String`, and create public getter and setter methods for it.
- Add another `password` string field in the same manner

The Struts framework is able to recognize the `LoginForm` class as a JavaBean and access its properties using the Java reflection APIs—each field in the HTML form must have a corresponding property with the same name in the form bean. The form bean is populated from the fields when the form is submitted.

To let Struts know about this class as a form bean, edit the `struts-config.xml` file and add the following declaration inside the `<struts-config>` tags:

```
<form-beans>
   <form-bean name="logonForm" type="itso.was4ad.action.form.LoginForm"/>
</form-beans>
```

The association between the form and its form bean is done through the action specified in the form and the associated action mapping, as explained in the section that follows. This means that a form bean can be reused in several similar forms.

Building a Struts action

As we described in “Action objects” on page 160, Struts actions are the interface between the incoming requests and the business logic. Every HTTP request causes a call from the `ActionServlet` to the `perform` method of the appropriate action class, an instance of which is created.

For the Struts version of the login there is one possible request from the `index.jsp` page. We must create a corresponding action class in Java by performing these tasks:

1. Create a new package for our action classes—we name the new package `itso.was4ad.webapp.action`
2. Create a new class named `LoginAction` in the new package
3. Create the `perform` method in the class, as described below

LoginAction perform method

The code for the `perform` method is shown in Figure 12-4.

```
public ActionForward perform(
    ActionMapping mapping,
    ActionForm form,
    HttpServletRequest request,
    HttpServletResponse response)
    throws IOException, ServletException {

    // Get a reference to the HTTP session
    HttpSession session = request.getSession();

    try {
        // Use the DisplayCustomer use case to locate the customer info
        DisplayCustomer useCase = new DisplayCustomer();
        useCase.setCustomerId(((LoginForm) form).getCustomerId());
        CustomerData data = (CustomerData) useCase.execute();

    // Save the user information in the session
        session.setAttribute("customer", data);
    } catch (Exception e) {
        ActionForward result = mapping.findForward("loginNotSuccessful");
        return result;
    }

    ActionForward result = mapping.findForward("loginSuccessful");
    return result;
}
```

Figure 12-4 Struts LoginAction perform method

The full method signature is:

```
public org.apache.struts.action.ActionForward perform(
   org.apache.struts.action.ActionMapping mapping,
   org.apache.struts.action.ActionForm form,
   javax.servlet.http.HttpServletRequest request,
   javax.servlet.http.HttpServletResponse response)
   throws java.io.IOException, javax.servlet.ServletException
```

We can avoid the long fully-qualified class names by adding these imports to the class source code:

```
import org.apache.struts.action.*;
import javax.servlet.http.*;
import java.io.IOException;
import javax.servlet.ServletException;
```

The method is very similar to the `execute` method in the `LoginCommand` class in the basic PiggyBank application—it simply uses the `DisplayCustomer` use case class to locate the customer information and store it in the HTTP session. No authentication is performed.

The method looks up and returns a Struts `ActionForward` object depending on whether the login was successful or not. The forward object is referenced by a symbolic name—the actual URI to forward to is determined by the Struts configuration file. This arrangement allows the structure of the Web site to be modified without altering any code.

Defining the action mapping

The action class and the pages it may forward to are defined in the Struts configuration file. We edit the `struts-config.xml` file and add the declaration for our new action inside the `<struts-config>` tags (Figure 12-5).

```
<action-mappings>
   <action path="/login"
      type="itso.was4ad.action.LoginAction"
      name="loginForm"
      scope="request"
      validate="false"
      input="/index.jsp">
      <forward name="loginSuccessful" path="welcome.jsp"/>
      <forward name="loginNotSuccessful" path="index.jsp"/>
   </action>
<action-mappings>
```

Figure 12-5   Struts login action configuration

Form validation

In “Defining the action mapping” on page 295”, the form validation attribute in the XML is set to `false`. We can enable validation of the form by modifying the `validate` attribute of the mapping to `true`. We must also include the `input` attribute—this tells the framework which page to return to if input validation fails.

When we enable validation this the `ActionServlet` calls the `validate` method of the specified form bean before performing the action. The method can examine the submitted values and report any validation errors back to the client.

To validate our login input information, we add a `validate` method to the `LoginForm` class (Figure 12-6).

```
public ActionErrors validate(ActionMapping mapping, HttpServletRequest req)
{
   ActionErrors errors = new ActionErrors();
   try {
      this.customerId = Integer.parseInt(getUser());
   } catch (Exception e) {
      errors.add("user", new ActionError("error.login.user"));
   }
   return errors;
}
```

Figure 12-6 ActionErrors validate method

This method basically checks if the customer ID supplied is a valid integer.

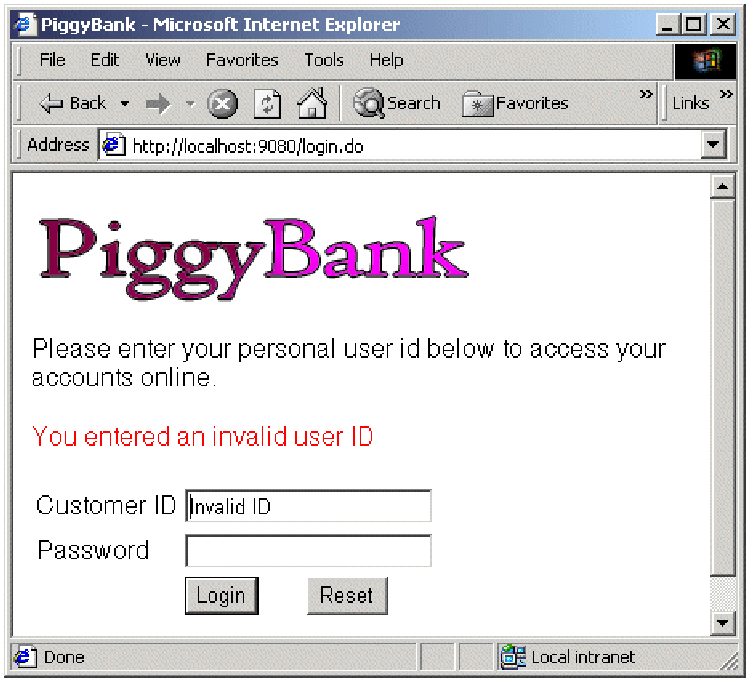
The return object is a collection of all the errors that have been encountered during the validation process. If it contains more than one `ActionError`, the `ActionServlet` returns a collection of error objects to the input JSP, which can retrieve and display the errors—for instance above the form—using a very simple Struts custom tag:

```
<html:errors/>
```

The PiggyBank login form includes this tag, enclosed in `font` tags to display the errors in red:

```
<p>
<font color=”red”>
  <html:errors/>
</font>
</p>
```

Figure 12-7 shows the output from the login page displayed when a user enters an invalid user ID.


PiggyBank - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites Links
Address
Login
Reset
Done
Local intranet

http://localhost:9080/login.do

PiggyBank

Please enter your personal user id below to access your accounts online.

You entered an invalid user ID

Customer ID: Invalid ID

Password:

Figure 12-7 Output from the Struts errors tag

The error message displayed in the JSP is resolved using the key `error.login.user` given at `ActionError` object creation time. Let's explains this error message resolving process in depth.

Message facility

It is usually a good idea to not hardcode messages from the application in Java code and JSPs, but to externalize them in plain text files instead. This can bring several benefits:

- There is no need to recompile every time a message changes.
- There are fewer opportunities to introduce errors in the code accidentally.
- Presentation formatting can be separated from the presentation content.
- Internationalization is more straightforward (see “Internationalization” on page 299).
- Messages can be reused, leading to better consistency and reduced translation costs.

Struts provides complete support for externalizing messages, built on top of the standard Java implementation. The messages are all placed in a standard `.properties` resource file, where they are identified by keys and associated by the `=` sign, following the `<key>=<value>` pattern, such as:

```
welcome.hello=Hello
welcome.welcome=Welcome to PiggyBank!
error.login.user=You entered an invalid user ID
```

We place the lines above in a file and call it `PiggyBankResources.properties`.

To access the messages from the application, the resource file must be fully-qualified in the class path and referenced from the Web application configuration file, as a parameter of the `ActionServlet` configuration:

```
<servlet>
   <servlet-name>action</servlet-name>
   <servlet-class>org.apache.struts.action.ActionServlet</servlet-class>
   [...]
   <init-param>
      <param-name>application</param-name>
      <param-value>PiggyBankResources</param-value>
   </init-param>
   [...]
</servlet>
```

To put this file in the VisualAge for Java class path, click on the *Resources* tab, right-click on the project and select *Add -> Resource*. Browse to the desired directory and select the file or files you want to add.

When we package the code into a J2EE Web module we must remember to include the file in the WAR archive in the `WEB-INF/classes` directory.

If you are using Ant to build your application place the properties file at the base of the source tree and add the following to the XML build file:

```
<!-- Pick up the resource files -->
<classes dir="${basedir}">
   <include name="**/*.properties"/>
</classes>
```

To display the "Hello" message in `include.jsp`, we can remove the hardcoded String and replace it by this Struts custom tag:

```
<bean:message key="welcome.hello"/>
```

To have the custom tag working, the following tag library must be declared in the JSP:

```
<%@ taglib uri="/WEB-INF/struts-bean.tld" prefix="bean" %>
```

To externalize text in HTML reset and submit buttons, we can use the following Struts tags:

```
<html:reset>
   <bean:message key="button.reset"/>
</html:reset>
```

Internationalization

In “Message facility” on page 297“, we saw how hardcoded message strings can be removed from our code and JSPs. This leads us to a new advantage: it becomes much easier to internationalize the application. Internationalization (also known as I18N, because there are 18 letters between the I and the N) is the means by which we can enable our single application for user communities that understand different human languages.

There are two stages to internationalizing our application:

- Translate messages in our resource files into our chosen languages
- Provide a means in the application to change the current locale (the language settings for the application to use)

The steps we must perform for Struts more or less follow the standard Java internationalization technique, which is clearly explained in the Javasoft tutorial and API documentation:

```
http://java.sun.com/j2se/1.3/docs/api/java/util/ResourceBundle.html
http://java.sun.com/docs/books/tutorial/i18n/index.html
```

Translating messages

Translated messages are put in files named `PiggyBankResources_xx.properties` where `xx` is the ISO-639 language code. A list of the ISO-639 language codes can be found at:

```
http://www.ics.uci.edu/pub/ietf/http/related/iso639.txt
```

For completeness, a list of the ISO-3166 country codes can be found at:

```
http://userpage.chemie.fu-berlin.de/diverse/doc/ISO_3166.html
```

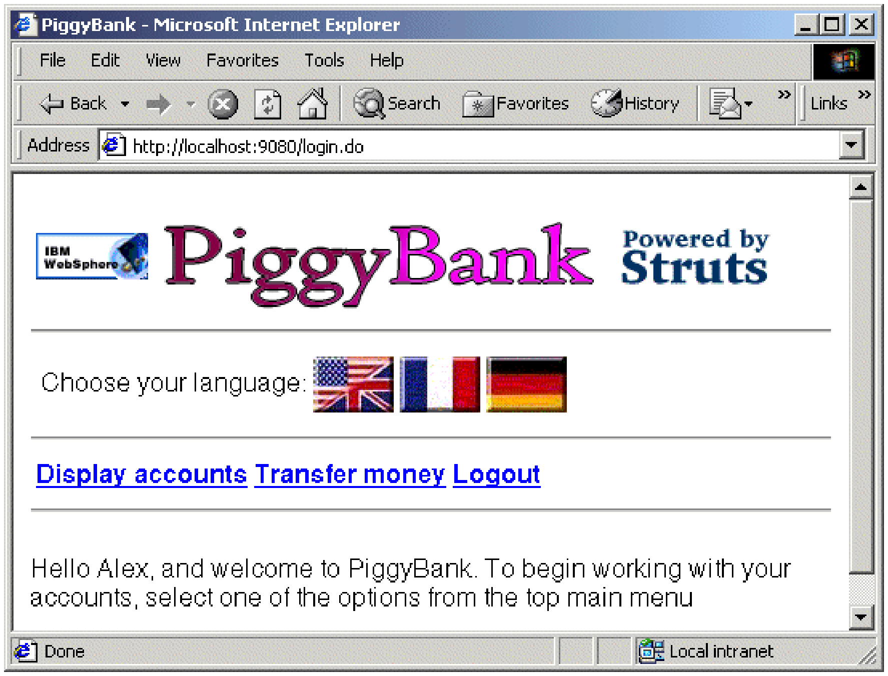
Selecting a language

Web applications often place a *Choose language* option in the main menu, which can appear during the entire HTML navigation. This can be easily done for our PiggyBank Struts example in the `include.jsp` file:

```
<TD><A HREF="javascript:submitChangeLanguageForm('en', 'US')"><IMG
src="images/en.gif" width="50" height="35" border="0"></A></TD>
```

```
<TD><A HREF="javascript:submitChangeLanguageForm('fr', 'FR')"><IMG
src="images/fr.gif" width="50" height="35" border="0"></A></TD>
<TD><A HREF="javascript:submitChangeLanguageForm('de', 'DE')"><IMG
src="images/de.gif" width="50" height="35" border="0"></A></TD>
```

This displays three flags—American English, French and German—linked to a JavaScript function which passes two locale parameters to a new Struts form that allows us to change the language. The language selection menu can be seen in Figure 12-8.


PiggyBank - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites History Links
Address http://localhost:9080/login.do
IBM WebSphere
PiggyBank
Powered by Struts
Done
Local intranet

Choose your language:

Display accounts Transfer money Logout

Hello Alex, and welcome to PiggyBank. To begin working with your accounts, select one of the options from the top main menu

Figure 12-8 The PiggyBank Struts language selection menu

The JavaScript function is defined as follows:

```
<script language="JavaScript" type="text/javascript">
<!--
function submitChangeLanguageForm(languageValue, countryValue) {
   changeLanguageForm.language.value=languageValue;
   changeLanguageForm.country.value=countryValue;
   changeLanguageForm.submit();
}
-->
</script>
```

This JavaScript function puts the locale parameters into the following hidden form and submits it:

```
<html:form action="/changeLanguage.do">
   <html:hidden property="language"/>
   <html:hidden property="country"/>
</html:form>
```

On the server-side, a corresponding `ChangeLanguageForm` bean must be created with the two locale properties:

- `language`
- `country`

And declared in the Struts configuration file:

```
<form-beans>
   [...]
   <form-bean name="changeLanguageForm"
      type="itso.was4ad.action.form.ChangeLanguageForm"/>
</form-beans>
```

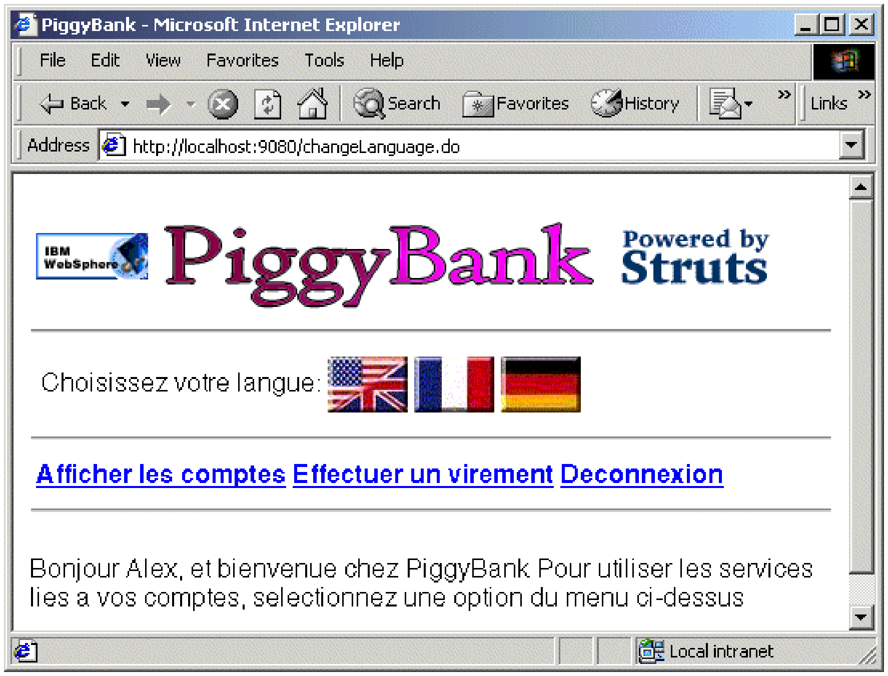
The locale is set by the `ChangeLanguageAction` class, which has a `perform` method that looks like this:

```
try {
   itso.was4ad.action.form.ChangeLanguageForm changeLanguageForm =
      (itso.was4ad.action.form.ChangeLanguageForm) form;
   java.util.Locale locale =
      new java.util.Locale(
         changeLanguageForm.getLanguage(),
         changeLanguageForm.getCountry());
   session.setAttribute(Action.LOCALE_KEY, locale);
} catch (Exception e) {
   // do not mind, default locale will be used by Struts
}
ActionForward result = mapping.findForward("welcome");
return result;
```

This action must also be declared in the Struts configuration file:

```
<action-mappings>
[...]
   <action path="/changeLanguage"
         type="itso.was4ad.action.ChangeLanguageAction"
         name="changeLanguageForm" scope="request" validate="false">
      <forward name="welcome" path="welcome.jsp"/>
   </action>
</action-mappings>
```

Now, clicking on one of the flags in the main menu will automatically change all the Struts messages displayed in the JSPs to the selected language (provided these JSPs use the Struts message facility, at least everywhere the `<bean:message>` tag is used). This locale setting will last to the end of the session or until the user selects another language. This is illustrated in Figure 12-9.


PiggyBank - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites History Links
Address
Local intranet

http://localhost:9080/changeLanguage.do

PiggyBank Powered by Struts

Choisissez votre langue:

Afficher les comptes Effectuer un virement Deconnexion

Bonjour Alex, et bienvenue chez PiggyBank Pour utiliser les services lies a vos comptes, selectionnez une option du menu ci-dessus

Figure 12-9 French language welcome page using Struts

Struts conclusions

This chapter has briefly covered only some of the capabilities of the Struts framework. In addition to the actions described here, the example code described in Appendix A, “Additional material” on page 557 also implements logout and account display actions. Beyond this we recommend you examine the example code and documentation that comes with the Struts distribution to gain a fuller grasp of the framework’s capabilities.

Despite the limited scope we hope we have given you an insight into the capabilities of Struts and the ease with which applications can be developed using the framework.

WebSphere Business Components Composer

In this section we explain how to develop the PiggyBank application using the WebSphere Business Components Composer, which we name here “WSBCC” while the PiggyBank application developed with WSBCC will be named PiggyBank-WSBCC.

We first consider that the application has to be written from scratch, so that we use the least from the framework capabilities. Then, we introduce the use of additional WSBCC features and so rely on less custom code. At the end, the application looks the most like a typical front-end application integrated in an existing enterprise environment.

Importing WSBCC into VisualAge for Java

The WSBCC binary files are distributed under the form of compiled class files in a jar-packaged archive file. To import those files from it into VisualAge for Java:

- Create a new project in VisualAge for Java, for example `WSBCC`.
- Right-click on the `WSBCC` project and select *Import*. In the import SmartGuide, select the *Jar file* radio button. Click *Next*.
- Browse to the `$wsbcc-dir$\WSBCC4.jar` file. Choose to import the .class and resource files. Leave the options unchecked. Click *Finish*.
- Right-click on the `WSBCC` project. Select *Manage -> Version*. Select *One Name* and enter `4.0`. This should version the imported packages and classes recursively.

For further development, we also create a new `itso.was4ad.wsbcc` package in the ITSO WAS AD project.

WebSphere Studio setup

Create a new project in Studio or use an existing one. Create a working directory called `piggybank-wsbcc`. In the *Publishing Targets* configuration (Figure 12-1), set the HTML publishing target to:

```
D:\IBMVJava\ide\project_resources\IBM WebSphere Test Environment\
                hosts\default_host\piggybank-wsbcc\web
```

Insert these files from PiggyBank:

- `welcome.jsp`
- `accountDisplay.jsp`
- `displayAccountsResult.jsp`
- `transfer.jsp`

- `transferResult.jsp`
- `error.jsp`
- `loginfail.jsp`
- `logout.jsp`

Create an `XML` directory in the piggybank-wsbcc directory and insert six files:

- `dse.ini`
- `dseoper.xml`
- `dsectxt.xml`
- `dsedata.xml`
- `dsefmts.xml`
- `dsesrvc.xml`

These files are the WSBCC configuration files. We alternatively suggest not to start actually from blank files but to use the final sample files provided with the redbook. This jumpstarts the first tests. All along this chapter, we are going to show and explain the major concepts behind the code.

Attention: Remember these directories are supposed to be published. Any change on any file should be checked in and published from Studio before being tested in the Websphere Test Environment. For more information on the Studio checkin and publishing mechanism, see Chapter 10, “Development using WebSphere Studio” on page 237.

WTE setup

To test PiggyBank-WSBCC along with the existing Web applications, the WebSphere Test Environment has to be configured to support a new Web application. Edit the file:

```
D:\IBMVJava\ide\project_resources\IBM Websphere Test Environment\
         properties\default.servlet_engine file
```

and add the following PiggyBank-Struts Web application declaration between the `<websphere-servlet-host name="default_host">` tags:

```
<websphere-webgroup name="piggybank-wsbcc">
   <description>PiggyBank application using WSBCC</description>
   <document-root>$approot$</document-root>
   <classpath>$approot$</classpath>
   <root-uri>/piggybank-wsbcc</root-uri>
   <auto-reload enabled="true" polling-interval="3000"/>
   <shared-context>false</shared-context>
</websphere-webgroup>
```

This block redirects all the URI starting with `/piggybank-wsbcc` to the Piggybank-WSBCC Web application, which is configured in the configuration file:

```
D:\IBMVJava\ide\project_resources\IBM Websphere Test Environment\
      hosts\default_hosts\piggybank-wsbcc\servlets\piggybank-wsbcc.webapp
```

The `piggybank-wsbcc.webapp` configuration file is based on the common `webapp` file plus one declaration for the WSBCC servlet that will pass the HTTP requests to the Web service:

```
<servlet>
   <name>htmlController</name>
   <code>com.ibm.dse.cs.servlet.CSReqServlet</code>
   <autostart>true</autostart>
   <servlet-path>/html</servlet-path>
</servlet>
```

Please note this configuration file is temporary and will be enhanced in the next sections.

Automatic server startup

We suggest to write first a simple and convenient facility that helps developing with WSBCC. We want to have:

- WSBCC started at servlet engine startup
- The ability to restart WSBCC from the browser, for instance to reload the WSBCC XML configuration file
- A versatile configuration support for both development and production

To this end, we create a new servlet `itso.was4ad.wsbcc.StartServerServlet`. It will be architected in three parts, that is, three methods (Figure 12-10):

- A servlet-engine-triggered server start entry point
- A browser-triggered server restart entry point
- A common server initialization routine

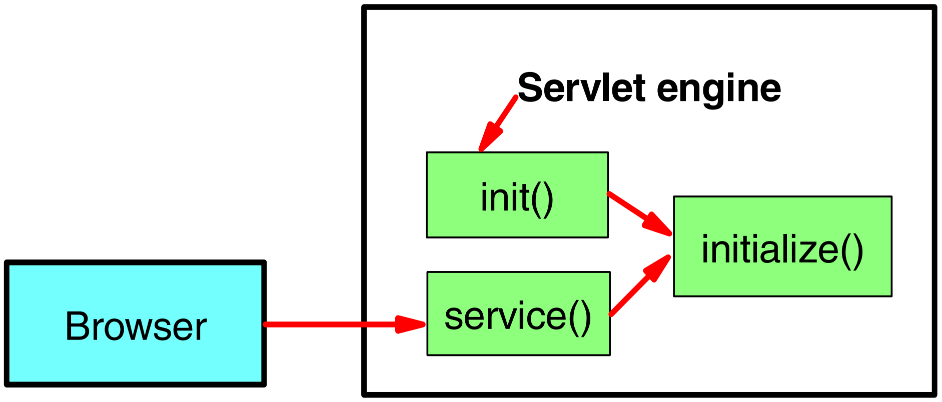


Figure 12-10 WSBCC startup architecture

initialize method

The `initialize` method basically calls the necessary framework code to have it initialized properly from the specified `.ini` file, the path of which is given as a parameter:

```
private void initialize(String iniFileName) throws Exception {
   Context.reset();
   HandlerRegistry.resetInstance();
   // Read data from .ini file
   Settings.reset(iniFileName);
   Settings.initializeExternalizers(Settings.MEMORY);
   // Create the initial context in the server
   Context context = new Context("globalContext");
   // Initialize the client-server service, required for session management
   ((CSServerService)context.getService("CSServer")).initiateServer();
}
```

init method

The `init` method overrides the standard J2EE API method to initialize a servlet. It takes the `.ini` file name from the servlet engine Web application configuration and calls the `initialize` method.

Furthermore, it gets another parameter stating whether WSBCC can be restarted through a HTTP request or not. Finally, a third parameter tells whether the servlet class loading into the servlet engine should trigger the WSBCC startup sequence. This is mainly used to start WSBCC along with the servlet engine.

In total, this method gets three parameters that usually take these values, according to the runtime environment (Table 12-2).

Table 12-2 WSBCC startup parameters

Parameter name	Development value	Production value
`initStart`	true/false	true
`acceptHttpRestart`	true	false

The `iniFile` can have any name and can be put in any place. Just make sure it is not accessible through the Web. The init method code looks like this:

```
public void init(ServletConfig sc) {
    try {
        super.init(sc);
        String initStart = getInitParameter("initStart");
        if (initStart != null && initStart.equals("false")) {
            // Do nothing: the user doesn't want to initialize the
            // environment in the Application Server's startup
        } else {
            // set the HTTP restart preference
            this.acceptHttpRestart =
                (new Boolean(getInitParameter
                          ("acceptHttpRestart"))).booleanValue();
            // Get the path of the server's dse.ini file
            String path = getInitParameter("iniFile");
            if (path == null) {
                path = this.defaultIniFileName;
            } else {
                this.defaultIniFileName = path;
            }
            //only try to initialize when the .ini file exists
            //otherwise trust initialize() to be called from
            //the doGet() method
            if (new java.io.File(path).exists()) {
                initialize(path);
            }
            log("StartServerServlet initialized properly");
        }
    } catch (Exception e) {
        log("Exception in StartServerServlet.init(): " + e);
    }
}
```

service method

The `service` method uses the same `iniFile` parameter to restart WSBCC from a HTTP request, typically coming from a browser. This feature is especially useful in development as any change in an XML configuration file can be reloaded into WSBCC without having to restart the entire servlet engine. As enhancements, the basic HTML output generated can include a hyperlink to the Web application home page and the framework restart access can be protected by a simple password. The appropriate method looks like this:

```
public void service(HttpServletRequest req, HttpServletResponse res)
                          throws java.io.IOException {
   // check the password
   String password = req.getParameter("password");
   if ((!this.acceptHttpRestart) || ((this.requiredPassword != null) &&
         (!this.requiredPassword.equals(password)))) {
      // consider it as violent DoS so immediately close the output stream
      // NB : it could be considered to log the hit source as well
      log("StartServerServlet hit with a wrong password");
      ServletOutputStream strmOut = res.getOutputStream();
      strmOut.println("No access.");
      strmOut.close();
      return;
   }
   // determine the entry point URL
   String entryPointUrl = req.getParameter("entryPointUrl");
   if (entryPointUrl == null) {
      log("Warning: no 'entryPointUrl' parameter in URL, using default");
      entryPointUrl = this.defaultEntryPointUrl;
   }
      // determine the location of .ini file
   String iniFileName = req.getParameter("iniFile");
   if (iniFileName == null) {
      log("Warning: no 'iniFile' parameter in URL, using default");
      iniFileName = this.defaultIniFileName;
   }
   String strMessage = "Initialization OK";
      // start the server
   try {
      log("Using the ini file: " + iniFileName);
      strMessage += " from file " + iniFileName + ".<BR>";
      strMessage += "You can now <a href=\"" + entryPointUrl +
                     "\">enter</a> the application.";
      initialize(iniFileName);
   }
   catch (Throwable t) {
      strMessage = "ERROR in server: " + t.toString();
   }
   //send response to client
```

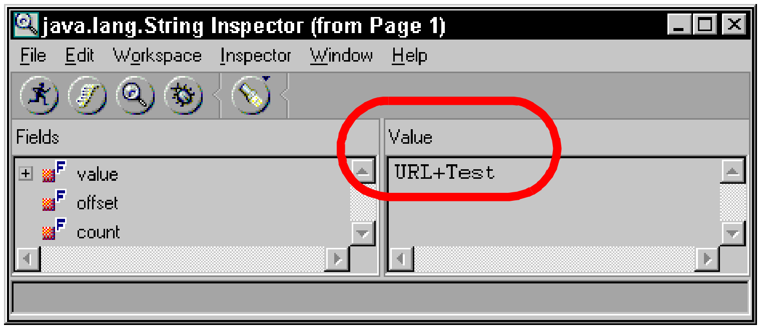
```
        ServletOutputStream strmOut = res.getOutputStream();
        strmOut.println("OK");
        strmOut.println("<TITLE>WSBCC Start</TITLE>");
        strmOut.println("</HEAD><BODY>");
        strmOut.println("<H1><B>" + this.getClass().getName() +
                        "</B></H1><HR>");
        strmOut.println(strMessage + "<BR><HR>");
        strmOut.println("</BODY></HTML>");
        strmOut.close();
        return;
    }
```

A benefit of this method is that we can create a project development page including several links to this servlet with various WSBCC configuration files and Web application home pages as HTTP parameters.

Default values are provided to lighten the URL while keeping a flexible entry point. This is especially useful when accessing the servlet through a HTTP GET method, where the URL parameters have to be encoded.

To have URL parameters encoded for GET method, it very convenient to use the VisualAge for Java scrapbook:

1. In the VisualAge for Java Workbench menu, select *Window -> Scrapbook*.
2. Type: `java.net.URLEncoder.encode("URL Test");`
3. Select all this code by pressing *CTRL+A* or by selecting *Edit -> Select All* in the scrapbook menu.
4. Inspect the result by pressing *CTRL+Q* or by selecting *Edit -> Inspect* in the scrapbook menu.
5. You can now copy and paste the encoded text in the inspector value window (Figure 12-11).



URL+Test

Figure 12-11 Inspector value window

Note that WSBCC expects a `.ini` file path name containing an OS-dependant path separator. While on a UNIX system it works with `/root/dse.ini`, WSBCC on a Win32 system requires a form like `C:\\dse\\dse.ini`. Also note that in this latter case, the \ sign is doubled, because Java Strings creation from constants uses the \ sign as an escape character. Furthermore, if you use the `java.net.URLEncoder`, enter a quadruple \\\\ because it also has to build a Java String from a constant before converting.

Configuration

The servlet we just built has to be declared in the Web application configuration:

```
<servlet>
   <name>startServerServlet</name>
   <code>itso.was4ad.wsbcc.StartServerServlet</code>
   <autostart>true</autostart>
   <servlet-path>/startServer</servlet-path>
   <init-parameter>
      <name>acceptHttpRestart</name> <value>true</value>
   </init-parameter>
   <init-parameter>
      <name>initStart</name> <value>true</value>
   </init-parameter>
   <init-parameter>
      <name>iniFile</name>
      <value>D:\\IBMVJava\\ide\\project_resources\\
            IBM WebSphere Test Environment\\hosts\\default_host\\
            piggybank-wsbcc\\XML\\dse.ini</value>
   </init-parameter>
</servlet>
```

The URL to restart the WSBCC server with default values is therefore:

```
http://localhost:8080/piggybank-wsbcc/startServer
```

Building WSBCC operations

WSBCC operations are similar to Struts action (see “Action objects” on page 160) and commands (see “Command pattern” on page 105). They are also use-case driven and follow the same design considerations as commands. There are as many operations as screens. Therefore, in PiggyBank-WSBCC, there are four main operations to be defined:

- `LoginOperation (welcome.jsp)`
- `DisplayAccountsOperation (displayAccountsResult.jsp)`
- `Transfer1Operation (transfer)`
- `TransferOperation (transferResult)`

Each operation executes its operation flow and gives the hand to the appropriate JSP. This latter functionality is not completely provided by the framework, and we have to extend the operation class to add it (Figure 12-12).

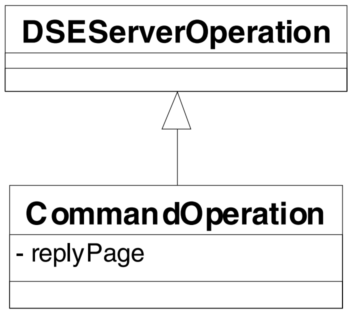


Figure 12-12 Operation extension

To do this in VisualAge for Java, right-click on the `itso.was4ad.wsbcc` package and select *Add -> Class*. Create a new class `CommandOperation` extending the superclass `com.ibm.dse.base.DSEServerOperation`. Add it a `replyPage` String attribute with a default value of `null`. Now a generic operation class has been defined with a `replyPage` attribute and all the child operation classes can use it.

The actual forward to the reply page will be performed by WSBCC as long as the `dse_replyPage` attribute is set to the corresponding value in the standard list of the HTTP request attribute. This can be done in the `execute` method, which has to be overridden by any user-defined operation:

```
public void execute() throws Exception {
   super.execute();
   setValueAt(com.ibm.dse.cs.html.HtmlConstants.REPLYPAGE,
      this.replyPage);
}
```

Note that the `dse_replyPage` parameter name is not actually hardcoded but is referenced instead by the `REPLYPAGE` constant in the `HtmlConstants` framework class.

The `execute` method implementation looks basically like the corresponding commands. An example can be found in the sample code distribution: `itso.was4ad.wsbcc.DisplayAccountsOperation`.

Extending XML externalization

To externalize for instance the `DisplayAccountsOperation` and have it recognized by WSBCC, we write some XML code in the `dseoper.xml` configuration file:

```
<itso.was4ad.wsbcc.DisplayAccounts
   id="displayAccountsOperation"
   operationContext="displayAccountsContext"
   replyPage="displayAccountsResult.jsp">
<itso.was4ad.wsbcc.DisplayAccounts />
```

This declares the WSBCC ID of the operation, the class that implements it, the operation context that is available to it and the reply page that should be sent back to the browser when the operation flow is finished.

Do the same for all the operations and include everything between `<dseoper.xml>` tags in the `dseoper.xml` file. Check as usual the standard `<?xml version="1.0"?>` XML starting tag is present.

To support the `replyPage` XML attribute, we extend the `initializeFrom` method of the `CommandOperation` class:

```
public Object initializeFrom(Tag aTag)
    throws java.io.IOException, DSEException {
    super.initializeFrom(aTag);
    com.ibm.dse.base.Vector attributes = aTag.getAttrList();
    for (int i = 0; i < attributes.size(); i++) {
        TagAttribute attribute = (TagAttribute) attributes.elementAt(i);
        if (attribute.getName().equals("replyPage")) {
            this.replyPage = (String) attribute.getValue();
        }
    }
    return this;
}
```

This copies the XML attribute to the Java attribute every time such an operation object is created by WSBCC. This functionality is automatically inherited by all the child operation classes, which can then externalize their inherited `replyPage` field. An obvious advantage here is that navigation and page naming is controlled in XML files instead of to-be-compiled Java code.

Login

In WSBCC, the login sequence is handled by a separate controller. A predefined login sequence is provided by a framework servlet, which we declare in the `piggybank-wsbcc.webapp` file:

```
<servlet>
   <name>loginController</name>
   <code>com.ibm.dse.cs.servlet.CSEstablishSessionServlet</code>
   <autostart>true</autostart>
   <servlet-path>/login</servlet-path>
</servlet>
```

This default controller executes the `startupOp` operation with an initial `sessionCtx` session context. These both WSBCC elements are defined in the `HTMLClient` section in the `dse.ini` file:

```
<kColl id="HTMLClient">
   <field id="minRequestResubmitTime" value="0"/>
   <field id="filePath" value="/"/>
   <field id="errorPage" value="error.jsp"/>
   <field id="homePage" value="welcome.jsp"/>
   <field id="startUpOp" value="loginOperation"/>
   <field id="sessionCtx" value="sessionContext"/>
   <field id="logOffOp" value="logoutOperation"/>
</kColl>
```

The `HTMLClient` section also contains three important attributes:

- `filePath`—the path to add to the web application path in order to find the JSPs when mentioning them in a communication with WSBCC, allowing to set the reply page to `XXX.jsp`, instead of a longer form like `/webappname/jsp/XXX.jsp`. Actually, in our example, we do not use this facility and we put a / value.
- `errorPage`—the error page where WSBCC can forward the request to when it encounters an uncaught exception during the web service execution.
- `homePage`—the page to forward to when the login sequence is complete and successful.

In our example, the login operation will put the customer information into the session context like the initial load did for PiggyBank.

Legacy example

In this section, we are already going to rewrite the current code to make it use more framework capabilities. We now assume the application has to run in an enterprise environment where a host serves the business requests on a legacy system. In this case, the PiggyBank-WSBCC application does not have to provide any business code, that is does not have to implement the use cases, but instead reuses and interfaces with the existing services the enterprise environment already provides.

Table 12-3 and Table 12-4 show some typical (invented) host service documentation specifying input and output message formats.

Note: The ~ character indicates a blank space.

Table 12-3 customerInfo request (similar to accountsInfo request)

Field	Length	Padding	Justify	Value
serviceId	5	0	right	00001
customerId	8	~	right	~~~~~101

Table 12-4 transfer request

Field	Length	Padding	Justify	Value
serviceId	5	0	right	00003
customerId	8	~	right	~~~~~101
debitAccountNumber	8	~	right	12345678
creditAccountNumber	8	~	right	23456789
amount	13	~	right	~~~~~~~~30.50

The request strings are typically formed by juxtaposing padded values like this:

```
00001~~~~~101
```

Response strings coming back from a host system are usually marked by delimiter characters separating the values (Table 12-5 and Table 12-6).

Table 12-5 customerInfo answer (single occurrence)

Field	Delimiter	Value
customerId	\	101\
customerName	\	Joe Bloggs\

Table 12-6 accountsInfo (multiple occurrence)

Field	Delimiter	Value
`number`	`\`	`1\`
`checkingFlag`	`\`	`YES\` or `NO\`
`balance`	`\`	`3200.00\`

A typical `accountsInfo` response would look like this:

```
31\YES\3200.00\37\NO\1415.50\
```

In "Defining formats" on page 320 we explain how WSBCC can easily handle these formats and many others.

Writing a WSBCC service

As we have no existing legacy system for PiggyBank, we are going to write a simulator that acts exactly the same (see Figure 12-13 and compare it to Figure 7-5 on page 168).

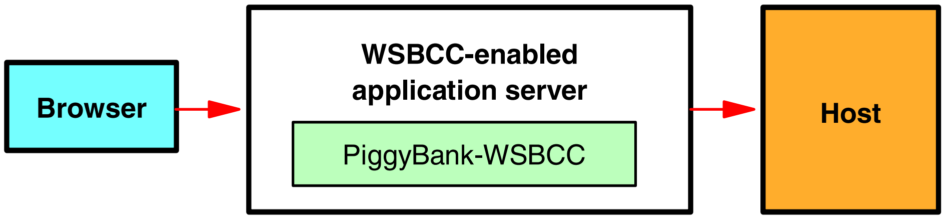


Figure 12-13 Host simulation from WSBCC point of view

Behind the scenes (Figure 12-14), the simulated host system will actually provide the required service using the working PiggyBank application we wrote in the previous chapters.

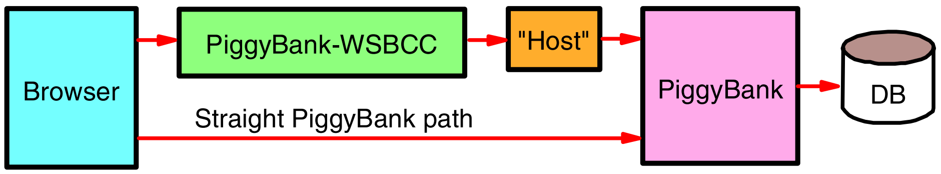


Figure 12-14 Host simulation behind the scenes

The simulator can be written as a WSBCC service, extending the com.ibm.dse.base.Service abstract class. The HostSystem class, being a Service in the OO meaning, can be externalized like any WSBCC service in the dsesrvce.xml file:

```
<HostSystem id="hostSystem" />
```

The tag name is referenced from the dse.ini file:

```
<kColl id="services">
   ...
   <field id="HostSystem" value="itso.was4ad.wsbcc.HostSystem"/>
   ...
</kColl>
```

We consider that this service is shared by the entire application. So we put it the global context in the dsectxt.xml file:

```
<context id="globalContext" type="Global" parent="nil">
   ...
   <refService refId="hostSystem" alias="hostSystem" type="host"/>
   ...
</context>
```

In "Dealing with contexts" on page 318 we explain that this creates only one instance of the itso.was4ad.wsbcc.HostSystem class in the unique global context. Therefore its execute method must be thread-safe, which can easily be achieved by using the standard Java single-semaphore facility: declare the method as synchronized.

The actual simulator code can be found in the itso.was4ad.HostSystem class in the sample code distribution provided with the redbook.

Generic WSBCC operations

As we are now connecting to the legacy system to perform the business logic, the WSBCC operations task is reduced to a single stereotyped succession of actions:

- Format a host request with context data
- Send request to the host
- Unformat the host response into the context
- Display the returned information to the user by forwarding the flow to a JSP

The last feature and the corresponding XML externalization code is already provided by our previously written `CommandOperation`. So we can extend this class to a `GenericHostOperation` that will implement the first three requirements in its `execute` method:

```
public void execute() throws Exception {
   super.execute();
   if ((this.serviceId != null) && (!this.serviceId.trim().equals(""))) {
      // get context service
      HostSystem hostSystem = (HostSystem) this.getService("HostSystem");
      // put service id in the operation context
      this.getContext().setValueAt("serviceId", this.serviceId);
      // format request from context
      FormatElement hostRequestFormat =
         (FormatElement) this.getFormat("hostRequestFormat");
      String request = hostRequestFormat.format(this.getContext());
      // host request
      String response = hostSystem.execute(request);
      // unformat response into context
      FormatElement hostResponseFormat =
         (FormatElement) this.getFormat("hostResponseFormat");
      hostResponseFormat.unformat(response, this.getContext());
   }
}
```

Extra externalization code has to be provided for the `serviceId` attribute in the `initializeFrom` method, similar to the code written for the `replyPage` attribute:

```
public Object initializeFrom(Tag aTag)
   throws java.io.IOException, DSEException {
   super.initializeFrom(aTag);
   com.ibm.dse.base.Vector attributes = aTag.getAttrList();
   for (int i = 0; i < attributes.size(); i++) {
      TagAttribute attribute = (TagAttribute) attributes.elementAt(i);
      if (attribute.getName().equals("serviceId")) {
         this.serviceId = (String) attribute.getValue();
      }
   }
   return this;
}
```

For completeness, here is a list of the services supported by the host simulator:

serviceId	**operationId**	**Use case**
`00001`	`customerInfo`	*Initial Load*
`00002`	`accountsInfo`	Display Accounts
`00003`	`transfer`	Transfer Money

As we can see in the execute method of the `GenericHostOperation`, the request and response formats are externalized too and referenced by two identifiers:

- `hostRequestFormat`
- `hostResponseFormat`

This externalization and reference are already implemented by WSBCC. Thus, a generic host operation that transfers money can be fully externalized in the `dseoper.xml` file:

```
<itso.was4ad.wsbcc.GenericHostOperation id="transferOperation"
operationContext="genericDynamicContext" serviceId="00003"
replyPage="transferResult.jsp">
   <refFormat name="hostRequestFormat" refId="transferRequestFormat" />
   <refFormat name="hostResponseFormat" refId="transferResponseFormat" />
</itso.was4ad.wsbcc.GenericHostOperation>
```

In “Defining formats” on page 320 we explain how to actually externalize the formats.

Dealing with contexts

We now define and explain the WSBCC context paradigms.

Actually, the term *context* speaks for itself. In WSBCC, a context is modeled as a group of data elements, which are illustrated in Figure 7-8 on page 172.

Figure 7-6 on page 169 shows that a context is shared all along the framework flow and is available to all of its components.

To put and retrieve data in a context, get and set methods are provided by WSBCC (`getValueAt` and `setValueAt` in the `com.ibm.dse.base.Context` class). Figure 7-7 on page 171 and Figure 12-15 illustrate an important property of the WSBCC contexts: they can be *chained*.

Figure 12-15 shows that any unsatisfied get/set method call on a context is passed to the parent context, an so on, up to the root context (having a `nil` parent).

To avoid exceptions when a data name to have its value set is not found in the hierarchy, WSBCC provides a dynamic facility on the contexts, passing the set call back down, down to the leaf context, until a dynamic `KeyedCollection` is found, creating a new appropriate data element in it to hold the value. This feature should be used wisely to avoid growing global (unique to the application) and session (unique to a user session) contexts. Practically, we recommend to provide a generic dynamic `KeyedCollection` to the operation contexts and to leave the upper contexts non dynamic.

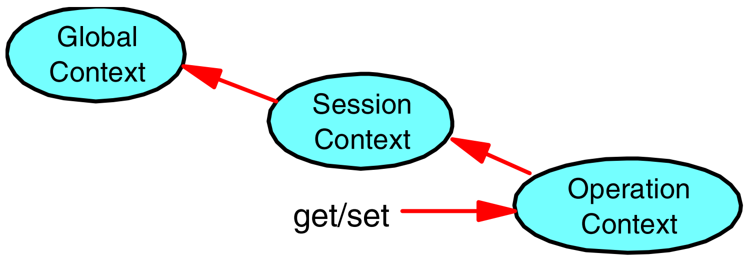


Figure 12-15 Context chaining

Contexts can also contain references to services. In PiggyBank-WSBCC, two services are needed at a global application level:

- The host simulator we developed in “Writing a WSBCC service” on page 315
- The built-in WSBCC client/server Web service

As a result to these considerations, the `dsectxt.xml` and `dsedata.xml` file look like this:

```
<?xml version="1.0"?>
<dsectxt.xml>
<context id="globalContext" type="Global" parent="nil">
   <refKColl refId="globalData"/>
        <refService refId="realCSServer" alias="CSServer" type="cs"/>
        <refService refId="hostSystem" alias="HostSystem" type="host"/>
</context>
<context id="sessionContext" type="Session" parent="globalContext">
   <refKColl refId="customerData"/>
</context>
<context id="genericDynamicContext" type="Op" parent="sessionContext">
   <refKColl refId="genericDynamicData"/>
</context>
</dsectxt.xml>
```

```
<?xml version="1.0"?>
<dsedata.xml>
<kColl id="globalData" dynamic="false">
</kColl>
<kColl id="customerData" dynamic="false">
   <field id="customerId" />
   <field id="customerName" />
</kColl>
<kColl id="genericDynamicData" dynamic="true">
</kColl>
</dsedata.xml>
```

We can see the syntax is very straightforward in this case. For further development, a complete syntax guide can be found in the WSBCC product documentation.

Defining formats

WSBCC provides a very complete and customizable set of components to externalize the context format and unformat processes that are shown in Figure 12-16.

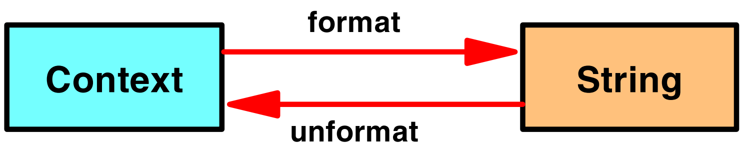


Figure 12-16 Format and unformat processes

Imagine we have a context containing these two data elements:

Name	Value
`serviceId`	`00002`
`customerId`	`1`

Also, the expected host request format is a juxtaposition of the `serviceId` element value and of the `customerId` padded with blank characters at its left.

This very common case can be solved by a simple WSBCC format externalization:

```
<fmtDef id="accountsInfoRequestFormat">
   <record>
      <fString dataName="serviceId"/><fixedLength length="5"
         justify="right" padChar="0"/>
      <fString dataName="customerId"/><fixedLength length="8"
         justify="right" padChar=" "/>
   </record>
</fmtDef>
```

The `id` attribute in the `fmtDef` tag is used to be referenced for instance from the `dseoper.xml` file as described in “Generic WSBCC operations” on page 316.

Each line between the <record> tag is composed of two different kinds of elements:

- A data reference
- A decorator

The two instances we have used and many others are described in the WSBCC product documentation. Here their respective uses are obvious in the format process. In the unformat process, the decorator is used to delimit the part of the string to be unformatted into the decorated data reference.

The two main decorators are:

- The fixed length decorator, which sets the string cut after a fixed number of characters (Figure 12-17).

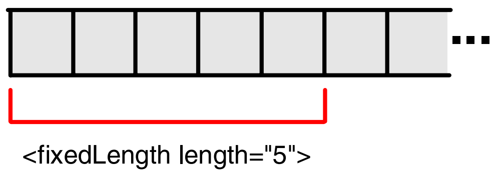


Figure 12-17 Fixed length decorator

- The delimiter decorator, which set the string cut before a specified character (Figure 12-18).

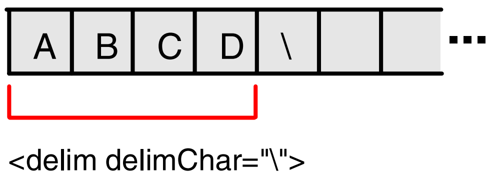


Figure 12-18 Delimiter decorator

At unformat time, the difference between a `<record>` and a `<dRecord>` is that when a data name specified in a `<dRecord>` is not present in a dynamic context, it is created instead of throwing an exception. The `<nullCheck>` decorator can be used with the similar intention at formatting time.

Finally, an `<iCollF>` tag unformats a series of record occurrences into an IndexedCollection. The `times` attribute specifies how many times the unformat process has to be applied. A value of * lets the unformat process go until the string is completely unformatted.

Presentation

As soon as the operation flow is over, the HTTP request is forwarded to the appropriate JSP, which is responsible for the presentation.

The link between the JSP and the framework is done through the use of a so-called `utb` bean. WSBCC provides a default bean:

```
com.ibm.dse.cs.html.DSEJspContextServices
```

We recommend to extend the default bean (Figure 12-19) to have three basic features to access the context without having any WSBCC classes knowledge in the JSP:

- Get a string value from a `KeyedCollection`, giving a data name
- Get a string value from an `IndexedCollection`, giving an index and a data name
- Get the size of an `IndexedCollection`

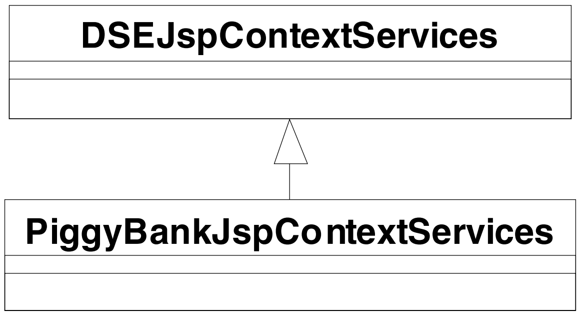


Figure 12-19 WSBCC utb bean extension

The actual implementation can be found in the sample code distribution:

```
itso.was4ad.wsbcc.PiggyBankJspContextServices
```

Before such a bean can be used in a JSP, it must be initialized with the standard JSP request variable representing the HTTP request object, where WSBCC actually stores the necessary information and can retrieve it through a call to the default `utb.initialize` method.

Therefore, the starting code of the JSP looks like this:

```
<jsp:useBean id="utb" class="itso.was4ad.wsbcc.PiggyBankJspContextServices"
scope="page">
 <% utb.initialize(request); %>
</jsp:useBean>
```

The `utb` bean can be used in the JSP to access the context in multiple ways:

```
<%
for (int i = 0; i < utb.getSize("accountsInfoIColl"); i++) {
%>
<TR>
   <TD>
      <%= utb.getIndexedToken("accountsInfoIColl", i, "number") %>
   </TD>
   <TD>
      $ <%= utb.getIndexedToken("accountsInfoIColl", i, "balance") %>
   </TD>
   <TD>
      <%= utb.getIndexedToken("accountsInfoIColl", i, "checkingFlag") %>
   </TD>
</TR>
<%
}
%>
```

```
Hi <%= utb.getStringValue("customerData.customerName") %>, and welcome to
PiggyBank.
```

Attention: As in any JSP, special care must be taken about catching possible exceptions, which break the output process and call the Servlet Engine error reporter:

- The default WebSphere error reporter displays the stack trace so this should be really avoided in a production environment.
- Although it is simply possible to write a less verbose error reporter, another possible solution is to add a standard JSP tag to specify an error page; that works as long as the JSP output is not flushed.
- Keep in mind that the `<jsp:include>` directive always flushes the output.

Additional error handling recommendations can be found in the WebSphere and WSBCC products documentations.

What we have achieved in this chapter

We have considered the PiggyBank application development starting with two different frameworks:

- Jakarta Struts
- IBM WebSphere Business Components Composer

This led us to two different development processes and mentalities. In each case, we focused our attention and efforts on the PiggyBank application components where the frameworks help the most, leading to incomplete results that are themselves very interesting, because they outline the real capabilities and main benefits of each of the frameworks.

This is therefore a valuable reference to be compared with the vision of any starting project that is considered to be developed with one these two frameworks.

13

Guidelines for coding WebSphere applications

In this chapter we present some guidelines to assist in writing code for WebSphere applications.

The topics we cover are:

- Using JNDI
- Message logging
- Coding for performance
- Managing multiple application versions in the same WebSphere environment

Using JNDI

Version 4.0 of WebSphere Application Server introduces some changes that impact the way code that accesses JNDI should be written. Version 4.0 includes JNDI backward compatibility that ensures that all existing code developed for versions 3.02 and 3.5 of WAS will continue to work. New code, however, should be written to comply with the J2EE specification—the changes required are described in this section.

Support for JNDI in WAS is described in detail in the WebSphere InfoCenter.

Obtaining an InitialContext

The J2EE specification recommends that code accessing JNDI obtain a reference to a JNDI `InitialContext` object using the default constructor, with no arguments. Furthermore, the specification requires that the container provide an environment in which a valid `InitialContext` will be obtained by using the default constructor. See Section 6.9 “Java Naming and Directory Interface (JNDI) 1.2 Requirements” in the Java 2 Platform Enterprise Edition Specification, V1.2; and Section 18.2.1.3 “JNDI 1.2 requirements” in the Enterprise JavaBeans Specification, V1.1.

WAS Version 4.0 and Version 3.5 both fulfil this requirement, meaning that an `InitialContext` can be simply obtained using the code:

```
import javax.naming.InitialContext;
...
InitialContext context = new InitialContext();
```

This code will function correctly in EJBs, servlets, JSPs and application clients running in the appropriate WebSphere container.

Earlier versions of WebSphere required you to specify the initial context factory class and the naming service provider URL in a `Hashtable` or `Properties` object to the `InitialContext` constructor. This approach will still work, however you should be aware that the factory class has changed in Version 4.0:

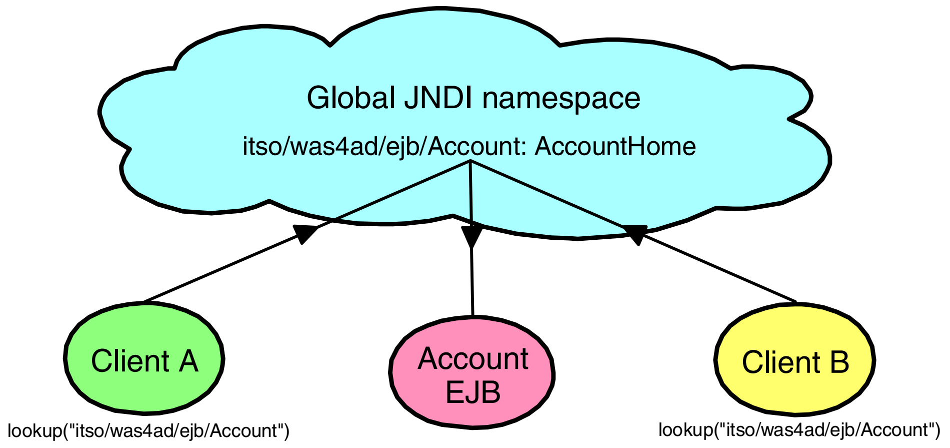
```
com.ibm.ejs.ns.jndi.CNInitialContextFactory              <=== Version 2/3
com.ibm.websphere.naming.WsnInitialContextFactory        <=== Version 4
```

The factory class supported by earlier versions of WebSphere is still provided for backwards compatibility with old code, however its use has been deprecated—the internal implementation of the earlier factory class simply uses the new factory.

When you develop new code for WebSphere Application Server Version 4.0 we recommend that you use the default constructor as described above. This approach ensures the best compatibility with the J2EE specification. If you find you must specify properties to the `InitialContext` constructor, we strongly recommend you externalize these from your application code.

Local and global JNDI namespaces

Earlier versions of WAS provided a single global JNDI namespace. Client code that needed to locate an EJB's home interface, for example, would perform a lookup on the global JNDI name for the EJB. For any one EJB home, all clients running in the same WebSphere cluster that wanted to access the EJB had to know the global JNDI name that the EJB was deployed with. This situation is illustrated in Figure 13-1, where both client A and client B locate the EJB using the global JNDI name `itso/was4ad/ejb/Account`.



Global JNDI namespace
itso/was4ad/ejb/Account: AccountHome
Client A
Account EJB
Client B
lookup("itso/was4ad/ejb/Account")
lookup("itso/was4ad/ejb/Account")

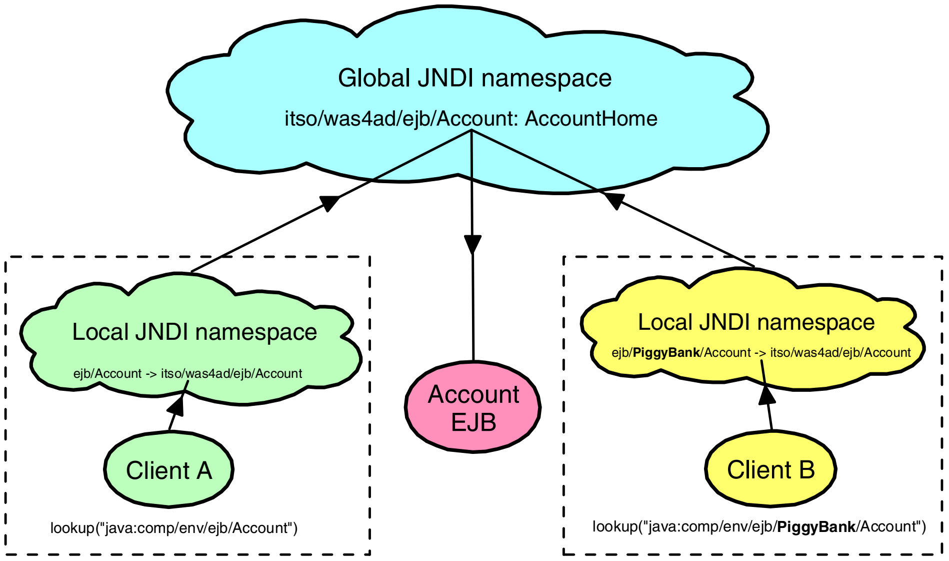
Figure 13-1 Global JNDI namespace

Version 4.0 of WAS, in line with the J2EE specification, introduces the concept of a local JNDI namespace. Each Web application, client application and individual EJB has its own local JNDI namespace that the component accesses by performing lookups with names that begin `java:comp/env`.

When a J2EE module is created each component must define in its deployment descriptor all the resources that it expects to find in the local JNDI namespace. These resources may include EJB homes, data sources, mail providers, and general configuration information the component expects to find in its environment. We can enter this information manually into the deployment descriptor XML, or use the WebSphere Application Assembly Tool (AAT) GUI.

When the module is installed into the application server, the deployment tool locates all of the local references declared in the deployment descriptor, and asks the deployer to provide global JNDI names that correspond to the actual deployed resources that the modules want to locate.

At runtime, when a client performs a JNDI lookup in its local JNDI namespace, the container uses the information supplied when the application was installed to map the local name understood by the module to the global name which identifies where the component is actually located. This is illustrated in Figure 13-2, where the two client modules locate the same EJB home using two different local JNDI names.



Global JNDI namespace
itso/was4ad/ejb/Account: AccountHome
Local JNDI namespace
ejb/Account -> itso/was4ad/ejb/Account
Client A
lookup("java:comp/env/ejb/Account")
Account EJB
Local JNDI namespace
ejb/**PiggyBank**/Account -> itso/was4ad/ejb/Account
Client B
lookup("java:comp/env/ejb/**PiggyBank**/Account")

Figure 13-2 Local and global JNDI namespaces

This extra level of indirection at the JNDI level makes it much easier to assemble applications from components created by multiple providers, because it eliminates naming conflicts between components and allows each component's resources to be configured in a consistent manner without having to know anything about the component's internal implementation.

This feature is also useful for allowing multiple versions of the same components to run independently in the same WebSphere cluster—we describe in detail how to manage this in "Managing application versions" on page 371.

Because the global JNDI namespace remains as before, existing applications coded for earlier versions of WebSphere will continue to behave in the same manner as before.

We do recommend, however, that any new code developed for Version 4.0 of the application server be written to use the local JNDI namespace as described here. Such applications will ultimately be more flexible and easier to manage, as well as complying more closely with the J2EE specification.

Caching JNDI lookup results

With earlier versions of WebSphere a well-documented and widely implemented best practice concerning coding using JNDI recommended that any object returned as a result of a JNDI lookup be cached by the application code for future reuse. Following this recommendation significantly enhanced the performance of applications running in versions of the application server that did not implement any internal JNDI caching mechanism.

As a result many organizations have written and successfully deployed application code that caches JNDI lookup results. A common implementation uses a helper class following the Singleton design pattern—the singleton manages a cache of objects in a `Hashtable`, using the JNDI name used to locate the object as a key.

With the introduction of Version 4.0 of the application server, we find ourself in a position where we must re-examine this best practice and, in certain circumstances at least, caution against its use.

Important: Existing code caching objects looked up in the **global** JNDI namespace will continue to work correctly without modification. This discussion recommends a migration away from this approach; however, it is not necessary to do so to migrate code from earlier versions of WebSphere Application Server.

There are two factors that drive this new recommendation:

- The introduction of a local JNDI namespace
- The evolution of the caching capabilities of the WebSphere JNDI implementation

First, let us consider the caching singleton helper class. With the introduction of a local JNDI namespace, just as multiple components may use different local JNDI names to refer to the same resource, two components may also use the *same* local JNDI name to refer to *different* resources. Under these circumstances use of a caching singleton to perform the JNDI lookups will result in a race condition, whereby only the first component to perform the lookup will obtain the correct resource. This situation will clearly result in application failures. We strongly caution you against making such an error.

The second consideration is the WebSphere implementation of JNDI. Internal JNDI caching by the WebSphere container code has been in the product since fixpack 2 of WAS Version 3.5. With the introduction of Version 4.0, we can reasonably consider this implementation to be both mature and robust.

This is welcome, because maintaining a cache in application code is both burdensome and redundant. It forces the application developer to consider infrastructure issues specific to the particular container implementation, and duplicates effort already implemented. We no longer believe or recommend that maintaining your own caching code is either necessary or appropriate in WAS Version 4.0.

Message logging

In this section we outline the reasons for establishing a coherent strategy for managing messages logged by an application. We use the PiggyBank application code to demonstrate how flexible logging capabilities can be incorporated into an application, leveraging two existing logging systems; the WebSphere application server trace facility, and Log4J, an open source logging mechanism from the Apache Jakarta project.

Why do we need a logging framework?

Virtually all useful applications have a requirement to output messages of one sort or another. In a simple desktop application it may be sufficient to simply display messages to the end-user in a dialog box, or write them to a local disk file. The environments in which WebSphere applications run are usually much more demanding, however.

Messages may be logged for different reasons—during the initial stages of development, for example, a developer may need to output the values of some variables to assist in debugging, whereas in a production environment an administrator needs to be notified only when important events occur. If the production system is down, however, the values that the developer needed during debugging may be crucial to the administrator solving the problem and getting the application running again.

The simplest method of logging from a Java application is simply to write messages to standard out, or to a file. Indeed, if no logging strategy is in place, this is what often happens. The problem with this scheme however, is that each developer in the team usually chooses a different message destination, a different output format, and a different mechanism for controlling what messages are displayed.

Consider also that WebSphere applications often span many machines, using load balancing and remote components. This means a single user interaction could cause code to be executed in multiple processes running on multiple machines. Performance is another critical point to bear in mind—writing to standard output or to disk is a relatively expensive operation—so we should avoid it if the message is unlikely to be read.

In our experience, the limitations of not having a strategy are usually not exposed until an application is being handed over from development to the team that will administer it in production, which far too late to effect major changes. In the early stages of the hand over, administrators investigating problems find themselves searching through different files and trying to understand whether a message they find is relevant or not to the problem they are trying to solve. Eventually they call for assistance from a developer, who then decides to insert some more messages in order to narrow down the problem. When the new build with the extra messages arrives, the cycle starts again.

In this section we describe how a small amount of effort invested in a logging framework at the beginning of a project can reap dividends later on in the development cycle. A consistent approach to logging will result in an application that is easier to debug, easier to test, and easier to manage in production.

What do we need from a logging framework?

The key requirements of a logging framework are outlined below. Both of the logging facilities we demonstrate later on in this section meet these requirements.

Ease of use	A logging framework has to be easy to use. If it is not, developers simply will not use it. If the code does not output any messages, it is not much use to anybody. Consider writing simple tools to assist with inserting basic logging statements into code.
Performance	Performance is one of the main justifications for not including detailed logging information in application code. Under normal conditions the logging code must have a minimal impact on the performance of the application. In error conditions this requirement need not be so strict, because in these circumstances providing the information needed to fix the problem is the key consideration.
Types	It must be possible to assign types to messages to distinguish between different message types and severities. It must also be possible to filter messages based on type.
Originator	Each message must be labelled so that the component that logs the message can be identified. Ideally the originating class

should be specified. In a WebSphere application many clients may be serviced concurrently by separate threads in the same process, so thread information is also crucial, and context information that identifies messages relating to a single client desirable. In a distributed environment it is also necessary to include the originating host name and process ID. It must be possible to filter messages based upon their origin.

Flexibility In addition to being able to filter messages by category and originator, it must also be possible to specify different destinations for messages based upon these criteria. For example, warning and error messages must also be fed into a systems management application such as Tivoli. For troubleshooting purposes it is also desirable that logging filters and destinations be altered dynamically at run time, since it may not always be possible to restart a WebSphere application for problem diagnosis.

Consistency All messages logged should have a consistent format to aid analysis—this is especially helpful for tools. Messages must be written to consistent destinations, based upon the originator and category of each message as described above. This removes the need to manually correlate information from multiple sources. In a distributed WebSphere application you should be able to automatically collate messages from all systems in a single location, such as a database or systems management application, or a common native operating system log.

Timestamps All messages must include a timestamp. Analyzing failures is much easier if you can tell whether two adjacent events happened together or some time apart. Ideally the absolute time should be supplied, especially where events from multiple sources are collated. In cases where performance is at a premium the cost of generating a time stamp may be reduced by using a relative timestamp.

Reliability The logging mechanism should be reliable and be able to continue logging messages in adverse circumstances. In particular, it should not have an absolute dependency on a single point of failure, or the correct functioning of a particular infrastructure component. If you are logging messages only to a database, for example, and the database suffers a failure, you will not be able to store or analyze any log entries until the database can be returned to service. In a production environment your systems management software may rely on the timely delivery of messages in order to flag error conditions for operator intervention.

PiggyBank log wrapper

In this redbook, we describe the use of two separate logging frameworks with our example PiggyBank application. In order to achieve this easily we created a simple wrapper class that hides the specific logging implementation from the application. Wrapping the logging framework in a helper class of our own provides us with a number of advantages:

- We can easily switch from one logging framework to another by rewriting the wrapper class.
- The application code is isolated from changes in a particular logging framework's API that may otherwise prevent us from moving to a newer version of the API.
- We have the potential to take advantage of new framework features without necessarily having to re-engineer our code.
- We can simplify the work for developers and improve consistency and better enforce logging policy in our application by limiting the logging API available to them.

There is some disadvantage in using the wrapper, although on balance it does suit our purpose well. You may find you have a different set of priorities in your own project. The issues we considered are listed below:

- The wrapper involves a small performance penalty—we designed the wrapper in such a way as to minimize this cost.
- The full facilities of our chosen framework are not available to us—this is countered by the ability to enhance our own API and switch frameworks if absolutely necessary.

The complete code for both implementations of our log wrapper class can be found in “Using the Web material” on page 558.

Designing the log wrapper

The PiggyBank log wrapper is designed with simplicity in mind. We decided to create a single helper class to manage logging. Each application class registers with the logging system by creating an instance of the helper class, and saving it in a static member. We decided to implement four message types:

debug	Debug messages are inserted in code by developers to assist in problem determination during the code and unit testing cycle. These messages may also be used for problem determination in the later stages of the project.
information	Information messages are used to log events that may have some importance under certain circumstances, but are not necessarily critical to the everyday functioning of the application.

	An example is to signal the successful initialization of a component such as a servlet.
warning	Warning messages are used to flag conditions that, while unexpected, do not prevent the application from functioning correctly. The condition may be a result of a flaw in the application business logic, for example the discovery of an inconsistency in the database when processing a particular record—the individual transaction may fail but the application as a whole can continue.
error	Error messages are used to signal more serious infrastructure errors, for example if resources such as EJBs, data sources or JMS queues cannot be located. The application may not be able to recover from the situation and operator intervention is required.

You may decide this set of types is too limited for your own project. Most logging frameworks provide a larger number of types, and some allow you define your own. Some frameworks also define specific types for particular event classes, such as entry and exit from methods.

Whatever conclusions you come to, take care not to overburden developers by forcing them to insert too many logging statements into the code. Make sure that for any event the appropriate choice of message type is clearly documented and understood by all developers. This is essential to having consistent logging behavior in your application.

For each of our message types our wrapper class defines two methods—one that accepts a single `Object` parameter, and another that takes two parameters—an `Object` and an `Exception`.

- The `Object` parameter will be converted to a `String` when the message is logged using its `toString` method. `Object` is used rather than `String` to allow the expense of string conversion to be delayed until absolutely necessary.
- The `Exception` parameter, if supplied, causes the stack trace from an exception to be included in the log output. This is a relatively expensive operation, so our policy encourages it be used carefully.

Some messages may require significant overhead before they are submitted to the log wrapper, for example, if a developer wants to log the contents of a data-only object. The wrapper provides methods that allow developers to check whether logging is enabled for the debug and information message types, the two types that will generate the most input and are also likely to be disabled in a production system.

The empty log wrapper class is shown in Figure 13-3. This class implements the API we expose to our application developers.

```
package itso.was4ad.helpers;
public class LogHelper {
    /**
     * Creates a new LogHelper instance for a component
     */
    public LogHelper(Class component) {}
    /**
     * Logs a debug message
     */
    public void debug(Object o) {}
    /**
     * Logs a debug message including stack trace from an exception
     */
    public void debug(Object o, Exception e) {}
    /**
     * Logs an error message
     */
    public void error(Object o) {}
    /**
     * Logs an error message including stack trace from an exception
     */
    public void error(Object o, Exception e) {}
    /**
     * Logs an informational message
     */
    public void info(Object o) {}
    /**
     * Logs an informational message including stack trace from an exception
     */
    public void info(Object o, Exception e) {}
    /**
     * Logs a warning message
     */
    public void warn(Object o) {}
    /**
     * Logs a warning message including stack trace from an exception
     */
    public void warn(Object o, Exception e) {}
    /**
     * Returns true if debug level logging is enabled for this component
     */
    public boolean isDebugEnabled() {}
    /**
     * Returns true if info level logging is enabled for this component
     */
    public boolean isInfoEnabled() {}
}
```

Figure 13-3   Empty wrapper class

Writing code to use the log wrapper

For application logging to be truly effective every class in your application must include logging code. This is true even for the most trivial components—the code may not seem complex at first, but it may evolve over time, and you never know when you might need to discover what other component is calling code that you originally thought was unimportant.

It is much easier to add logging to code as it is originally developed—retrofitting log messages to existing code is tedious at best. You will also find that the messages you log are more useful if you add them as you are developing the code, since your mind is focussed on the business problem in hand.

We recommend that you choose your logging API and policy before you develop any other code—even if you just start with an empty or trivial implementation, it is much easier if you log consistently from the start.

Initializing the log wrapper

Our log wrapper is implemented in the `itso.was4ad.helpers.LogHelper` class. Every application class must register itself with the logging framework by creating a static instance of the wrapper class. For convenience we add code to the application class to import the wrapper:

```
import itso.was4ad.helpers.LogWrapper;
```

We then create the wrapper instance and save it in a static member. The following example is taken from the `AccountBean` class that implements the PiggyBank `Account` EJB:

```
private static final LogHelper LOG = new LogHelper(AccountBean.class);
```

`LOG` can be declared as `final` since it will not be altered once set.

Logging messages

Application code uses the static `LOG` object to log messages. Figure 13-4 shows how the code for the `transfer` method of the PiggyBank `AccountManager` EJB makes use of the low wrapper to log various messages.

The very first debug message in the method builds a message that reports the message signature, complete with parameters. Because this involves string concatenation, which is a relatively expensive operation, we check to see if debug messages are enabled before building the message.

The other debug messages are simple strings that will be optimized to constants by the compiler, so we can rely on the code within the logging framework to check the logging level for us.

```
/**
 * Transfers money from one account to another
 * @param debitID int
 * @param creditID int
 * @param amount int
 */
public void transfer(int debitID, int creditID, int amount)
                       throws NonExistentAccount, InsufficientFunds {

    if (LOG.isDebugEnabled()) {
        LOG.debug("transfer(" + debitID + ", " + creditID + ", " + amount + ")");
    }
    try {
        // Locate the accounts
        LOG.debug("Looking up home");
        AccountHome home =
            (AccountHome) HomeHelper.getHome(ACCOUNT_HOME, AccountHome.class);
        LOG.debug("Locating debit account");
        AccountKey key = new AccountKey(debitID);
        Account debitAccount = home.findByPrimaryKey(key);
        LOG.debug("Locating credit account");
        key = new AccountKey(creditID);
        Account creditAccount = home.findByPrimaryKey(key);

        // Transfer the funds - debit first in case we get an InsufficientFunds
        //     exception
        LOG.debug("Debiting debit account");
        debitAccount.debit(amount);
        LOG.debug("Crediting credit account");
        creditAccount.credit(amount);
    } catch (FinderException e) {
        LOG.warn("transfer() caught finder exception", e);
        throw new NonExistentAccount(
            NonExistentAccount.WARNING,
            "Cannot locate accounts");
    } catch (NamingException e) {
        LOG.error("transfer() caught naming exception", e);
        throw new EJBException(e);
    } catch (RemoteException e) {
        LOG.error("transfer() caught remote exception", e);
        throw new EJBException(e);
    }
}
```

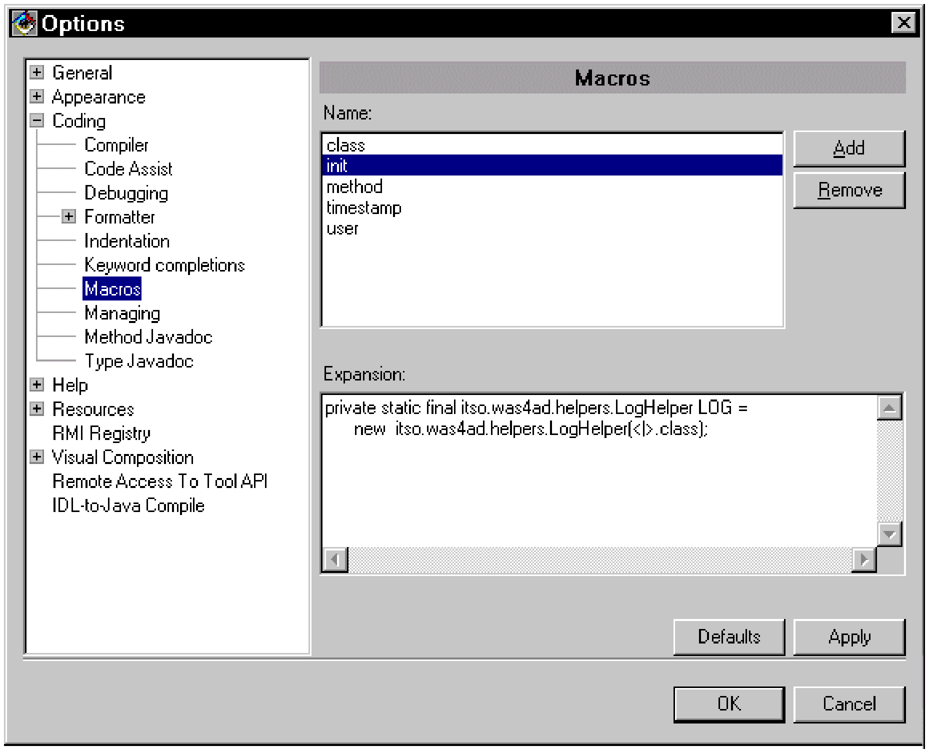
Figure 13-4 Logging messages using the log wrapper

Automation opportunities

Because every class in your application requires logging code that follows a similar pattern, you may find it convenient to automate some of the tasks involved. A simple starting point is to provide skeleton source files that already include the code to initialize logging for the component—all the developer would have to change would be the class name. A more sophisticated approach may involve scanning code as part of your build process, checking for and inserting logging code if necessary.

If you are using VisualAge for Java, you can quickly speed up the process of inserting logging code by defining macros. VisualAge for Java macros allow you define arbitrary text that can be inserted into source code using the code assist feature that is activated by pressing Ctrl-Space.

To create a new macro in VisualAge select *Window -> Options*, then in the Options window expand *Coding* and select *Macros*. Figure 13-5 shows the VisualAge for Java Options window being used to edit a macro called `init` that inserts code that initializes logging for a component.



Macros

Name:
class
init
method
timestamp
user

Expansion:

```
private static final itso.was4ad.helpers.LogHelper LOG =
    new  itso.was4ad.helpers.LogHelper(<|>.class);
```

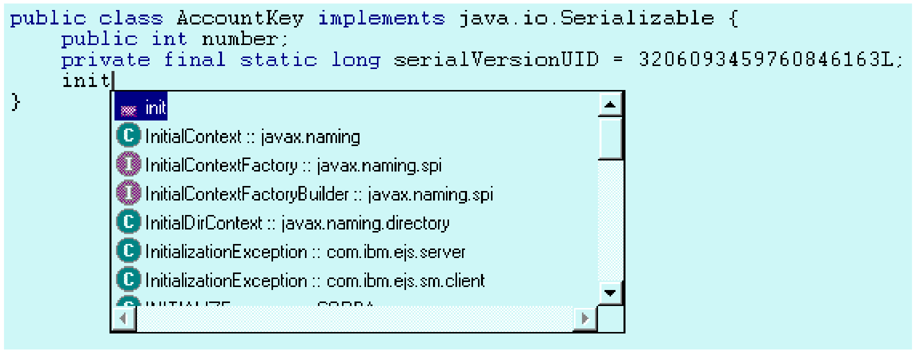
Figure 13-5   Creating a macro in VisualAge for Java

To create a macro click *Add* and enter a name for the new macro. This name identifies the macro in the code completion dialog. In the Expansion panel enter the code that will be inserted into the code when the macro is invoked. The character sequence <|> can be used in the macro to indicate where the cursor should be placed after the macro is inserted.

The code for our `init` macro is:

```
private static final itso.was4ad.helpers.LogHelper LOG =
   new  itso.was4ad.helpers.LogHelper(<|>.class);
```

To insert this macro into your code, type `init` then press Ctrl-Space to open the code completion window (Figure 13-6).

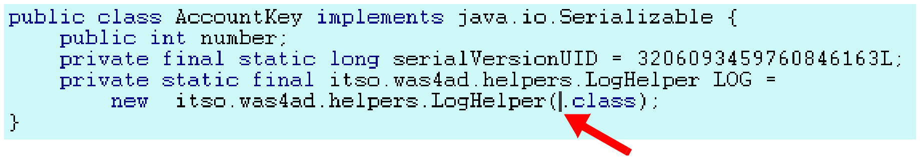


```
public class AccountKey implements java.io.Serializable {
    public int number;
    private final static long serialVersionUID = 3206093459760846163L;
    init
}
```

init
InitialContext :: javax.naming
InitialContextFactory :: javax.naming.spi
InitialContextFactoryBuilder :: javax.naming.spi
InitialDirContext :: javax.naming.directory
InitializationException :: com.ibm.ejs.server
InitializationException :: com.ibm.ejs.sm.client

Figure 13-6 Using code completion to insert a macro in VisualAge for Java

When you select the macro from the top of the list the macro code is inserted and the cursor is moved to the appropriate location to enter the new class name (Figure 13-7).



```
public class AccountKey implements java.io.Serializable {
    public int number;
    private final static long serialVersionUID = 3206093459760846163L;
    private static final itso.was4ad.helpers.LogHelper LOG =
        new  itso.was4ad.helpers.LogHelper(|.class);
}
```

Figure 13-7 Code inserted by VisualAge for Java macro

In addition to the `init` macro, we also created macros for each of the message types, and an `ifdebug` macro that inserts the code to log a debug message only if debug messages are enabled. The code for the `ifdebug` macro looks like this:

```
if (LOG.isDebugEnabled()) {
   LOG.debug("<|>");
}
```

Choosing a framework

There are a number of logging facilities available to WebSphere developers today. At a minimum you should ensure that the framework you choose meets the requirements described in “What do we need from a logging framework?” on page 331.

In this publication we discuss two options in more detail:

- The JRas facility provided with WebSphere Application Server
- Log4J from the Apache Jakarta project

Other frameworks you may want to investigate include the Logging Toolkit for Java (also known as JLog) from IBM, and Trace.Java, another open source package.

The Logging Toolkit for Java is available from the IBM AlphaWorks web site:

`http://alphaworks.ibm.com/`

Trace.Java is available from the Visible Workings Web site:

`http://visibleworkings.com/trace/`

You should also be aware that there are plans to introduce logging functionality into the core Java 2 API. The proposed API is described in Java Specification Request (JSR) 47. At the time of writing this specification has completed the public review stage and looks likely to be included in Version 1.4 of the Java 2 SDK. The API will be implemented in the `java.util.logging` package. For up to date information on the status of this specification request check the Java Community Process Web site at:

`http://jcp.org/`

Writing your own framework

You can of course choose to write your own framework, one that implements all the features that you need for your application.

We advise against this however; it is unlikely you’ll come up with a revolutionary system. You will simply end up spending more time implementing and maintaining your logging code—time that would be better spent working on your business logic.

If you really do find that none of the frameworks meets your requirements as-is, it will probably be easier to implement your needs as an extension to an existing framework such as Log4J. If you feel inclined to do so you can then submit your extension to the community and have other developers help maintain it with you.

Using the WebSphere JRas facility

Version 4.0 of WebSphere Application Server introduces support for the IBM JRas toolkit, a message logging and trace facility that is designed for use by application developers. JRas is a standalone IBM product that has been customized for use with WebSphere Application Server. The WebSphere JRas implementation is integrated with the internal logging system used by the application server code and has been present in the product for some time.

The JRas facility is powerful and flexible, and uses a ring buffer that can be dumped on command to store the most recently recorded events. The level of tracing and the components to be traced can be specified either when a process starts, or dynamically using the command line or GUI tools shipped with WebSphere. JRas also provides internationalization support for logged messages—the static text for messages are stored in text files that can be translated into different languages and the correct language version chosen at runtime.

For complete information on the WebSphere JRas implementation check the WebSphere documentation in the InfoCenter, available for download or browsing from:

`http://www.software.ibm.com/webservers/appserv/library.html`

To locate the JRas documentation navigate to the section "Using the JRas Message Logging and Trace Facility" in the master table of contents. The InfoCenter also includes JRas documentation in PDF format, and API documentation generated by the `javadoc` tool.

The decisive advantage that would lead you to use WebSphere trace to log application events is that messages from the application are fully integrated with messages from the application server. You use the same tools to control trace information from both sources, and the messages are collated in the same place.

Severe messages from the application are reported to the WebSphere administrator's console, and both WebSphere and your application can be integrated into your systems management infrastructure at the same time, simplifying the task and potentially halving the amount of work that needs to be done in that area.

This integration can also assist with debugging, particularly if you are concerned about how your application interacts with the application server, because messages from your application and WebSphere are interleaved in the same location, in the correct sequence and using the same message format.

Implementing the log wrapper using the JRas facility

The classes we need to implement our log wrapper using the WebSphere JRas facility are located in the `com.ibm.ras` and `com.ibm.websphere.ras` packages.

The first package contains the classes and interfaces that our logging code will use to log messages. The second package contains the singleton class `com.ibm.websphere.ras.Manager`. We use this class to create instances of the WebSphere-specific JRas implementation classes.

JRas message categories

JRas separates logged messages into two distinct categories:

Messages	Messages report significant events and are intended for the end-user of the application in day-to-day operations—they are enabled by default and support internationalization.
Trace	Trace reports low-level information to assist developers in the debugging of the application code—trace is not enabled by default, and does not support internationalization.

JRas provides a logging class for each of these two categories. Our debug messages fall into the JRas trace category, and our other message types into the message category, so our log wrapper has to manage an instance of the JRas trace logger for each of our components as well as an instance of the message logger class.

JRas supports a large range of different trace message types defined in the `RASITraceEvent` interface, whereas our simple log wrapper has only one. We map all our debug messages to the JRas message type `TYPE_MISC_DATA`.

The JRas message logger on the other hand defines three types of messages in the `RASIMessageEvent` interface—`TYPE_INFO`, `TYPE_WARN` and `TYPE_ERR`. These types can be mapped directly to the info, warning and error types provided by our log wrapper.

Writing the wrapper class

Figure 13-8 shows the outline of our log wrapper. We import the package that contains the JRas API, and declare instance variables to store the JRas objects that allow us to log messages and trace—the wrapper will manage these on behalf of the application components. We also store the class name and the package in instance variables, because these are required by the JRas API calls.

```
package itso.was4ad.helpers;

import com.ibm.ras.*;

/**
 * This class provides a log and trace facility to the PiggyBank
 * application. It is implemented as a wrapper around the
 * WebSphere JRas facility, to enable the underlying logging
 * framework to be changed without rewriting application code.
 */
public class LogHelper {
   // Instance variables
   private RASMessageLogger ml = null;
   private RASTraceLogger tl = null;
   private String className = null;
   private String packageName = null;

   // Statics
   private static boolean initialized = false;
   private static com.ibm.websphere.ras.Manager manager = null;

   // Constants
   private static final String ORGANIZATION = "PiggyBank Corporation";
   private static final String PRODUCT = "PiggyBank Application";
   private static final String DEFAULT_PACKAGE = "Default package";
}
```

Figure 13-8 Outline of the log wrapper class for WebSphere

The class defines two static variables; the first is a flag to indicate whether the logging system has been initialized, the second is a reference to the WebSphere singleton `Manager` class that we use to create JRas message and trace loggers. Finally, we declare some string constants that are used when we register components with JRas.

Log wrapper constructor

Each component in the application creates a new instance of the wrapper class to manage logging for that component. The class of the component is passed as a parameter to the log wrapper constructor.

First of all we make sure that we have initialized the log wrapper correctly. We then use the WebSphere JRas manager singleton to create a message and a trace logger for our component. We use constants to define the organization and product names for the loggers, and set the loggers' component name, specified in the third parameter, to the package name of the class. The complete code for the constructor is shown in Figure 13-9.

```
/**
  * Creates a new LogHelper instance for a component
  */
public LogHelper(Class component) {
   super();

   // Initialize JRas if necessary
   if (!initialized) {
      init();
   }

   // Set up the loggers for this component
   className = component.getName();
   int index = className.lastIndexOf(".");
   if (index > 0) {
      packageName = className.substring(0, index);
   } else {
      packageName = DEFAULT_PACKAGE;
   }
   ml =manager.createRASMessageLogger(ORGANIZATION, PRODUCT, packageName, className);
   tl =manager.createRASTraceLogger(ORGANIZATION, PRODUCT, packageName, className);
}
```

Figure 13-9   Log wrapper constructor using WebSphere JRas

Initializing JRas

Our log wrapper code performs initialization steps in the `init` method (Figure 13-10).

```
/**
  * Initialize the JRas logging system
  */
private static synchronized void init() {
   // Safeguard against race condition
   if (!initialized) {
      // Get a reference to the manager singleton
      manager = com.ibm.websphere.ras.Manager.getManager();

      // We have now initialized the logging system successfully
      initialized = true;
   }
}
```

Figure 13-10   Initializing the log wrapper using WebSphere JRas

The JRas version of the wrapper simply obtains and saves a reference to the WebSphere JRas `Manager` class.

Logging simple messages

We use two mechanisms to log messages without exceptions. Debug messages are logged using the `trace` method of the `RASTraceLogger` object managed by the log wrapper (Figure 13-11).

```
/**
  * Logs a debug message
  * @param o java.lang.Object The message to be written to the log
  */
public void debug(Object o) {
    if (isDebugEnabled()) {
        tl.trace(RASITraceEvent.TYPE_MISC_DATA, className, getCallingMethod(),
                 o.toString());
    }
}
```

Figure 13-11 Logging a simple debug message using WebSphere JRas

The `trace` method we use takes four parameters:

- The trace event type
- The name of the class logging the trace message
- The name of the method logging the trace message
- The trace message to be logged

There is an alternative method that allows the object logging the message to be specified rather than the class name—this would be useful during debugging in order to identify on which instance of a class a method is being invoked. Due to the architecture of our log wrapper we do not have this information available, however—if we had designed our log wrapper with JRas in mind, we may have chosen to include this information as a parameter.

The calling method name is determined by invoking the `getCallingMethod` method, shown in Figure 13-12.

This method parses the stack trace generated by creating a new instance of the `Throwable` class in order to determine the calling method's name. This is rather clumsy and inefficient, so we wrap the entire `trace` call in an `if` statement that determines whether debug messages are enabled.

```
/**
  * Returns the name of the method invoking the logger.
  * This method is very expensive, so try not to call it
  * unless absolutely necessary.
  */
private String getCallingMethod() {
   java.io.StringWriter sw = new java.io.StringWriter();
   java.io.PrintWriter pw = new java.io.PrintWriter(sw);
   new Throwable().printStackTrace(pw);
   String st = sw.toString();
   int start = st.indexOf(className) + className.length() + 1;
   int end = st.indexOf(")", start) + 1;
   return st.substring(start, end);
}
```

Figure 13-12 Determining the name of the method logging the message

The code for the other three simple logging methods is basically the same. Each method invokes a `textMessage` method on the JRas `RASMessageLogger` managed by the log wrapper, specifying the message type according to the mappings we described earlier.

The `textMessage` methods allows us to specify the text of the message to be logged—because our log wrapper is not designed to support internationalization, we cannot use the other methods that take message IDs and obtain the message text from message catalogs. If we want to take advantage of the internationalization support we will have to redesign our log wrapper.

The code for the `info` method is shown in Figure 13-13; the `warn` and `error` methods follow the same pattern, except that they do not check to see if the message type is enabled.

```
/**
  * Logs an informational message
  * @param o java.lang.Object The message to be written to the log
  */
public void info(Object o) {
   if (isInfoEnabled()) {
      ml.textMessage(RASIMessageEvent.TYPE_INFO, className, getCallingMethod(),
                     o.toString());
   }
}
```

Figure 13-13 Logging a simple information message using WebSphere JRas

Logging messages with exceptions

The JRas API provides methods for logging exceptions, but not for logging a message and an exception in a single method call. We work around this by logging two messages—the first logs the message text and the second the exception. Figure 13-14 shows the code for the `info` method—the other three methods follow the same pattern, except that the `debug` method uses the `RASTraceLogger` instead of the `RASMessageLogger` to log the message, and `warn` and `error` do not check to see if the message type is enabled.

```
/**
  * Logs an informational message including stack trace from an exception
  * @param o java.lang.Object The message to be written to the log
  * @param e java.lang.Exception The exception
  */
public void info(Object o, Exception e) {
   if (isInfoEnabled()) {
      ml.textMessage(RASIMessageEvent.TYPE_INFO, className, getCallingMethod(),
                     o.toString());
      ml.exception(RASIMessageEvent.TYPE_INFO, className, getCallingMethod(), e);
   }
}
```

Figure 13-14 Logging a message with an exception using WebSphere trace

Checking if debug and information messages are enabled

The last two messages in our JRas log wrapper allow code to check whether certain messages are enabled. Both the trace and message logger classes provide an `isLoggable` method that allows us to determine whether the message type is enabled. The code for the two methods can be seen in Figure 13-15.

```
/**
  * Returns true if debug level logging is enabled for this component
  * @return boolean
  */
public boolean isDebugEnabled() {
   return tl.isLoggable(RASITraceEvent.TYPE_MISC_DATA);
}

/**
  * Returns true if info level logging is enabled for this component
  * @return boolean
  */
public boolean isInfoEnabled() {
   return ml.isLoggable(RASIMessageEvent.TYPE_INFO);
}
```

Figure 13-15 Checking logging levels using WebSphere JRas

Building and deploying the log wrapper

The WebSphere JRas code is located in the archive `ras.jar`, which you will find in the `lib` directory under the directory where the application server is installed.

If you are developing code using VisualAge for Java you will find that the JRas code is already present in your workspace if you add the EJB development environment feature, so no further effort is required to build your code. If you are building your code outside of VisualAge, using the Java SDK or another IDE, you have to make sure that `ras.jar` is in the class path when you compile the log wrapper code.

Because the trace facility is used by the WebSphere runtime the relevant classes are already available during deployment and at runtime, so you do not have to include the `ras.jar` file in any other class path.

Controlling application logging with the JRas facility

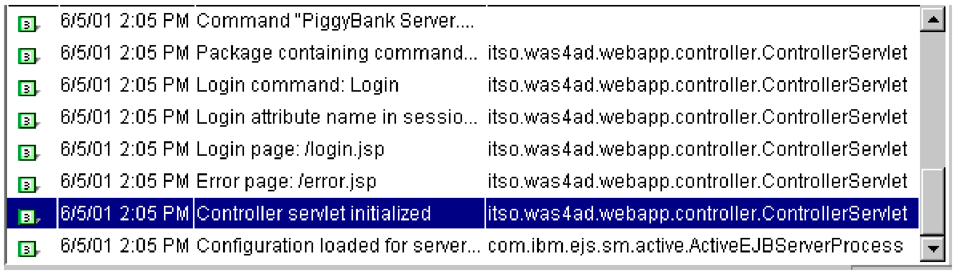
When we log application messages using the WebSphere JRas facility, we are able to use the same WebSphere tools to control the logging of messages from the application as we use to control logging of messages from WebSphere itself.

More complete information on how to control WebSphere trace is included in the product documentation—we use a small number of examples to illustrate some of the possibilities.

Logging using the administrator's console

One of the most useful side-effects of using JRas to log application messages is that informational, warning and error messages from code running in an application server are automatically posted to the console window.

Figure 13-16 shows part of the console window displaying messages logged using the `info` method of our class by a PiggyBank servlet on initialization.



```
6/5/01 2:05 PM Command "PiggyBank Server....
6/5/01 2:05 PM Package containing command...  itso.was4ad.webapp.controller.ControllerServlet
6/5/01 2:05 PM Login command: Login           itso.was4ad.webapp.controller.ControllerServlet
6/5/01 2:05 PM Login attribute name in sessio... itso.was4ad.webapp.controller.ControllerServlet
6/5/01 2:05 PM Login page: /login.jsp         itso.was4ad.webapp.controller.ControllerServlet
6/5/01 2:05 PM Error page: /error.jsp         itso.was4ad.webapp.controller.ControllerServlet
6/5/01 2:05 PM Controller servlet initialized itso.was4ad.webapp.controller.ControllerServlet
6/5/01 2:05 PM Configuration loaded for server... com.ibm.ejs.sm.active.ActiveEJBServerProcess
```

Figure 13-16 Application messages in the administrator's console window

The WebSphere administrator's console can also be used to dynamically modify the message types that will be logged by an application deployed in a running application server process.

To do this open the Trace dialog by selecting the appropriate application server in the console's tree view and selecting *Trace* from the pop-up menu (Figure 13-17).

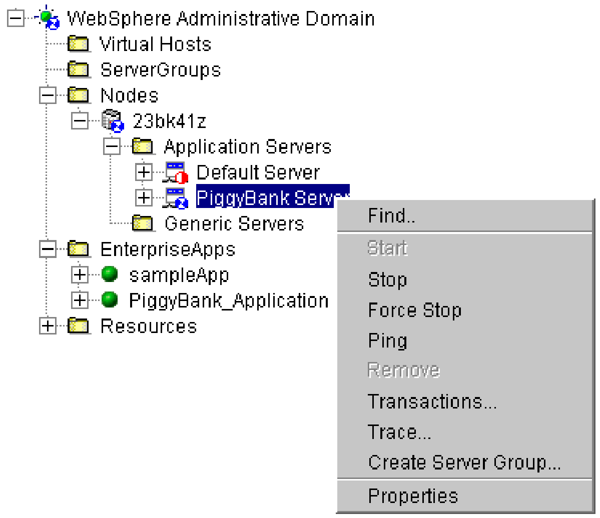


Figure 13-17   Opening the trace dialog for a running application server

The Trace dialog lists the application components alongside the WebSphere components. You can select an individual component or part or all of the component hierarchy and modify the message types that will be logged (Figure 13-18).

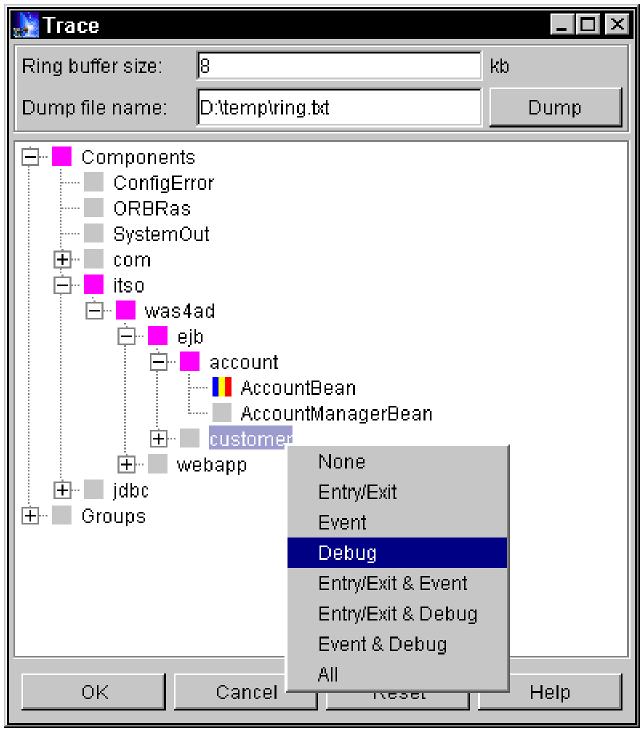


Figure 13-18 WebSphere trace dialog

Color coded boxes in the dialog indicate components for which message types have been activated. Once you have chosen the components and the message types you want them to log, click the *OK* button to enable the new settings.

The Trace dialog also allows you manage the application server's ring buffer, which maintains a circular log of the most recently recorded messages in memory:

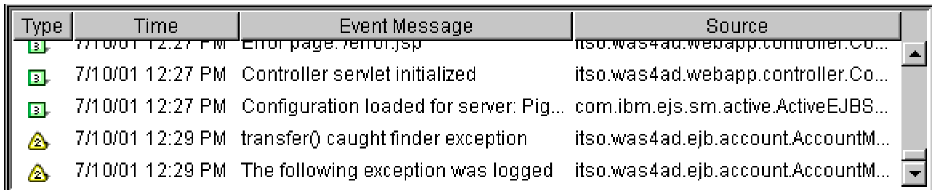
- To modify the size of the ring buffer enter the new size and click *OK*
- To dump the current contents of the ring buffer to a file, enter the file name and click *Dump*

Figure 13-19 shows an extract from a dumped ring buffer that shows debug messages logged by the PiggyBank code when the transfer method of the AccountManager EJB is invoked with an invalid account number.

```
[01.07.10 ......]  59ee96f AccountManage D itso.was4ad.ejb.account.AccountManagerBean
setSessionContext(AccountManagerBean.java:225) ORB.thread.pool:2 setSessionContext()
[01.07.10 ......]  59ee96f AccountManage D itso.was4ad.ejb.account.AccountManagerBean
ejbCreate(AccountManagerBean.java:159) ORB.thread.pool:2 ejbCreate()
[01.07.10 ......]  59ee96f AccountManage D itso.was4ad.ejb.account.AccountManagerBean
transfer(AccountManagerBean.java:236) ORB.thread.pool:2 transfer(123, 456, 1000)
[01.07.10 ......]  59ee96f AccountManage D itso.was4ad.ejb.account.AccountManagerBean
transfer(AccountManagerBean.java:240) ORB.thread.pool:2 Looking up home
[01.07.10 ......]  59ee96f HomeHelper    D itso.was4ad.helpers.HomeHelper
getHome(HomeHelper.java:24) ORB.thread.pool:2 Getting EJB home:
java:comp/env/ejb/Account
[01.07.10 ......]  59ee96f HomeHelper    D itso.was4ad.helpers.HomeHelper
getHome(HomeHelper.java:28) ORB.thread.pool:2 Getting InitialContext
[01.07.10 ......]  59ee96f HomeHelper    D itso.was4ad.helpers.HomeHelper
getHome(HomeHelper.java:30) ORB.thread.pool:2 Looking up home
[01.07.10 ......]  59ee96f AccountManage D itso.was4ad.ejb.account.AccountManagerBean
transfer(AccountManagerBean.java:244) ORB.thread.pool:2 Locating debit account
[01.07.10 ......]  59ee96f AccountBean   D itso.was4ad.ejb.account.AccountBean
setEntityContext(AccountBean.java:164) ORB.thread.pool:2 setEntityContext()
[01.07.10 ......]  59ee96f AccountBean   D itso.was4ad.ejb.account.AccountBean
ejbActivate(AccountBean.java:63) ORB.thread.pool:2 ejbActivate()
[01.07.10 ......]  59ee96f AccountBean   D itso.was4ad.ejb.account.AccountBean
unsetEntityContext(AccountBean.java:172) ORB.thread.pool:2 unsetEntityContext()
[01.07.10 ......]  59ee96f AccountManage W itso.was4ad.ejb.account.AccountManagerBean
transfer(AccountManagerBean.java:257) ORB.thread.pool:2 transfer() caught finder exc.
[01.07.10 ......]  59ee96f AccountManage W itso.was4ad.ejb.account.AccountManagerBean
transfer(AccountManagerBean.java:257) ORB.thread.pool:2 The following exception was
logged javax.ejb.ObjectNotFoundException: itso.was4ad.ejb.account.AccountKey@7b
```

Figure 13-19 Application debug messages from the WebSphere ring buffer

The warning and audit messages from this trace are also logged to the administrator's console window (Figure 13-20).

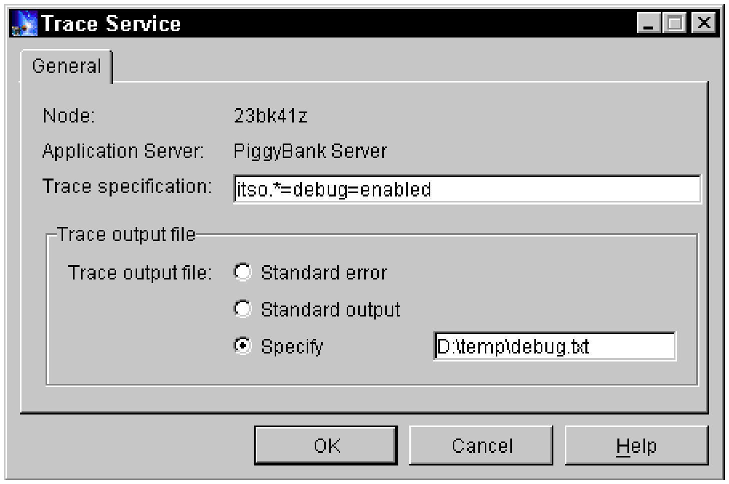


Type	Time	Event Message	Source
3	7/10/01 12:27 PM	Error page: /error.jsp	itso.was4ad.webapp.controller.Co...
3	7/10/01 12:27 PM	Controller servlet initialized	itso.was4ad.webapp.controller.Co...
3	7/10/01 12:27 PM	Configuration loaded for server: Pig...	com.ibm.ejs.sm.active.ActiveEJBS...
2	7/10/01 12:29 PM	transfer() caught finder exception	itso.was4ad.ejb.account.AccountM...
2	7/10/01 12:29 PM	The following exception was logged	itso.was4ad.ejb.account.AccountM...

Figure 13-20 Application warning messages in the administrator's console window

The administrators console can also be used to specify the initial WebSphere trace settings when an application server is started.

To do this open the Trace Service dialog (Figure 13-21). Select the application server in the console's tree view, then select *Properties* from the pop-up menu. In the application server Properties dialog select the *Services* tab, then select the *Trace Service*, and click the *Edit Properties* button.



Node: 23bk41z
Application Server: PiggyBank Server
Trace specification: itso.*=debug=enabled
Trace output file
Trace output file: ( ) Standard error ( ) Standard output [✓ Specify] D:\temp\debug.txt

Figure 13-21 WebSphere Trace Service dialog

Figure 13-21 shows the Trace Service dialog being used to edit the initial trace settings for the PiggyBank application server—debug messages from the entire application are to be sent to the file `D:\temp\debug.txt`. The format of the trace specification is described in full in the WebSphere documentation. To confirm the changes click the *OK* button.

Logging using the command line

The single server edition of Websphere Application Server does not feature the administrator's console supplied with the full edition of the software. With this edition the browser-based administration GUI offers similar functionality. As an alternative you may choose to use command line tools which work with all editions of the software.

Use the WebSphere `DrAdmin` command to administer WebSphere trace from the command line. This command communicates directly with a thread that runs in each WebSphere process that is dedicated to servicing the trace facility. The various command options are shown in Figure 13-22.

```
Usage:
  java com.ibm.ejs.sm.util.debug.DrAdmin [options]

Options:
  -help                 [Show this help message]
  -serverHost           <Server host name>
  -server               <Server name>
  -defaultConfiguration [Use default configuration file]
  -configurationFile    <configuration file>
  -serverPort           <Server port number>
  -testConnection       [Test Connection]
  -testVersions         [Test Connection and Versions]
  -retrieveTrace        [Retrieve the trace specification]
  -retrieveComponents   [Retrieve the trace components]
  -setTrace             <Trace specification>
  -setRingBufferSize    <Number of ring buffer entries, in K>
  -dumpRingBuffer       <Dump file> [default]
  -dumpState            <Dump string>
  -dumpThreads
  -dumpConfig           (all | server)
  -stopServer
  -stopNode

Either a server configuration file, or a server port number must
be specified.

Either the '-defaultConfiguration' option or the '-configurationFile' may
be provided.  The default configuration is 'server-cfg.xml'.

The '-testConnection' option may be used to test the connection
to the trace server; use -testVersions to test the connection
and to test program versions.

The '-dumpRingBuffer' option is executed if no other
options are specified.  In this case the dump file name is taken as
'JmonDump.' plus the current time in milliseconds.
```

Figure 13-22 DrAdmin command line options

The `DrAdmin` command has to know the number of the TCP/IP port that the server thread is listening on. This port number is written to the standard output of each WebSphere application server process soon after it starts. An example of this message is shown below:

```
[01.06.05 14:05:26:925 PDT] 3609a403 DrAdminServer I WSVR0053I: DrAdmin
available on port 1225
```

The port number used by the single server edition of WebSphere is defined in the application server's configuration file. The initial value specified in the default configuration file, `server-cfg.xml`, is 7000.

You may either specify the port number to `DrAdmin` using the `-serverPort` option, or, if you are using the single server edition of WebSphere, you can specify the configuration file which defines the port number for the application server process to use. The `-defaultConfiguration` flag tells the command to use the default configuration file.

If, for example, we want to enable all messages for our PiggyBank application in the server which logged the port number in the previous example, we issue the following command:

```
DrAdmin -serverPort 1225 -setTrace itso.*=all=enabled
```

If we are diagnosing a problem on a production system, we have to dump the log information into a file, then disable the trace again to reduce the performance impact of leaving debug messages enabled. We issue the following two commands:

```
DrAdmin -serverPort 1225 -setTrace itso.*=all=disabled
DrAdmin -serverPort 1225 -dumpRingBuffer ring.txt
```

The ring buffer is dumped into the `ring.txt` file in the working directory of the application server process.

Using Log4J

Log4J ia a subproject of the Apache Jakarta project, the same organization responsible for the Ant tool described in "Using Ant to build a WebSphere application" on page 197. Like Ant, Log4J is an open source tool that is maintained by a community of developers from many separate organizations.

In this section we describe only a very small subset of the features available in Log4J. For more information you should consult the Log4J Web site:

```
http://jakarta.apache.org/log4j/
```

The Log4J framework has been designed with performance and flexibility in mind. It comes with many standard extensions, and is easily extensible through the use of user written extensions which can be plugged in to the base framework.

Log4J also features a number of interesting and useful innovations, such as nested diagnostic contexts which allow context information to be associated with a thread and included in all messages logged by that thread. This is particularly useful in an application server where information such as a user name or client IP address can be used to easily identify events relating to a particular user.

Installing Log4J

We downloaded Version 1.1.1 of the Log4J distribution from the Log4J Web site. The download page is located at:

`http://jakarta.apache.org/log4j/docs/download.html`

We extracted the contents of the archive file onto our `D:` drive, creating a new directory, `D:\jakarta-log4j-1.1.1`.

Implementing the log wrapper using Log4J

The core Log4J API is implemented in the `org.apache.log4j` package. There are two classes in this package that we will use to implement our wrapper class using Log4J.

- The `Category` class is used to manage and perform logging for a particular component
- The `PropertyConfigurator` class is used to configure Log4J using a property file

Log4J message types

Log4J provides five message types, known as *priorities* in Log4J. These types are arranged in a hierarchy—enabling the logging of `WARN` messages for a component will also enable messages of the `ERROR` and `FATAL` priorities. Log4J also allows new priorities to be defined by extending the Log4J `Priority` class.

Because our simplified log wrapper supports only four message types, we map PiggyBank debug, warning, information and error messages to Log4J `DEBUG`, `INFO`, `WARNING` and `ERROR` messages, and disregard the Log4J `FATAL` priority.

Writing the wrapper class

Figure 13-23 shows the outline of our log wrapper. We import the package that contains the Log4J tracing API, and declare an instance variable to store the Log4J `Category` object that the wrapper will manage on behalf of the application components. We also declare a static `boolean` variable that we use to flag whether the Log4J framework has been initialized for the current process.

```
import org.apache.log4j.*;
/**
 * This class provides a log and trace facility to the PiggyBank
 * application. It is implemented as a wrapper around the
 * Jakarta Log4J logging facility, to enable the underlying logging
 * framework to be changed without rewriting application code.
 */
public class LogHelper {
   // Static variables
   static private boolean initialized = false;

   // Instance variables
   private Category category = null;
}
```

Figure 13-23 Outline of the log wrapper class for Log4J

Log wrapper constructor

Each component in the application creates a new instance of the wrapper class to manage logging for that component. The class of the component is passed as a parameter to the log wrapper constructor.

The constructor first checks to see if Log4J has been initialized, and initializes it if necessary. The code then creates a new Log4J `Category` object that manages logging for the component. The code for the constructor is shown in Figure 13-24.

```
/**
 * Creates a new LogHelper instance for a component
 */
public LogHelper(Class component) {
   super();

   // Initialize the logging system if required
   if (!initialized) {
      init();
   }

   // Register this component as a Log4J category
   category = Category.getInstance(component);
}
```

Figure 13-24 The log wrapper constructor for Log4J

Initializing Log4J

The Log4J framework is initialized by the `init` method, which is called by the constructor if Log4J is not already initialized. The code for this method is shown in Figure 13-25.

```
/**
 * Initialize the underlying logging system that this class wraps
 */
private static synchronized void init() {
   // Safeguard against possible race condition
   if (!initialized) {
      // Use a Log4J PropertyConfigurator to load logging information from
      // a properties file. configureAndWatch() will start a thread to
      // check the properties file every 60 seconds to see if it has changed
      // and reload the configuration if necessary.
      PropertyConfigurator.configureAndWatch("log4j.properties", 60000);

      // We have now initialized the logging system successfully
      initialized = true;
   }
}
```

Figure 13-25 Initializing Log4J

First we check to see if another wrapper class already initialized Log4J while we were waiting to enter the synchronized `init` method. If not, we use the Log4J `PropertyConfigurator` class to configure Log4J based on information in the file `log4j.properties`.

The `configureAndWatch` method searches for the file on the class path. It then checks every 60 seconds to see if the file has been modified, and reload the configuration if necessary. This rather crude mechanism allows the logging configuration to be altered dynamically in a running server.

Note: The PropertyConfigurator class is not the only way to manage the runtime configuration of Log4J. More sophisticated methods are described in the Log4J documentation.

Logging simple messages

The code for the four simple logging methods is basically the same. Each method invokes the appropriate method on the Log4J `Category` object managed by the wrapper, according to the mappings we described previously. The code for the `info` method is shown in Figure 13-26; the other three methods follow the same pattern.

```
/**
 * Logs an informational message
 * @param o java.lang.Object The message to be written to the log
 */
public void info(Object o) {
    category.info(o);
}
```

Figure 13-26 Logging a simple message using Log4J

The message object passed to the wrapper method is passed directly to the wrapped Log4J component. The Log4J framework only invokes `toString` on the message if logging is enabled for the message type.

Logging messages with exceptions

The Log4J API includes the facility to log an exception with a message, so the four methods to log exceptions are also virtually identical. Figure 13-27 shows the code for the `info` method.

```
/**
 * Logs an informational message including stack trace from an exception
 * @param o java.lang.Object The message to be written to the log
 * @param e java.lang.Exception The exception
 */
public void info(Object o, Exception e) {
    category.info(o, e);
}
```

Figure 13-27 Logging a message with an exception using Log4J

Checking if debug and information messages are enabled

The last two messages in our Log4J log wrapper allow code to check whether certain message types are enabled.

The Log4J `Category` object that we store in our wrapper class provides methods that provide this functionality and these methods simply delegate the request to the wrapped object. The code for the two methods is shown in Figure 13-28.

```
/**
 * Returns true if debug level logging is enabled for this component
 * @return boolean
 */
public boolean isDebugEnabled() {
   return category.isDebugEnabled();
}

/**
 * Returns true if info level logging is enabled for this component
 * @return boolean
 */
public boolean isInfoEnabled() {
   return category.isInfoEnabled();
}
```

Figure 13-28 Checking logging levels using Log4J

Building and deploying the log wrapper

The complete Log4J API is available in the archive `log4j.jar`, which you will find in the `dist\lib` directory under the directory where Log4J is installed. The core API is also provided in a smaller file, `log4j-core.jar`.

If you are developing code using VisualAge for Java you have to import at least the core API into your workspace in order to compile the wrapper class. You may find it convenient to import the Log4J source from the Log4J `src\java` directory into your workspace—this will enable you to step through the Log4J code in the VisualAge for Java debugger.

If you are building your code outside of VisualAge for Java , using the Java SDK or another IDE, you have to make sure that the `log4J.jar` file is in the class path when you compile the log wrapper code.

For deployment and runtime, we found it easiest to package the Log4J archive in the J2EE enterprise archive (EAR) file along with our application components. We then included `log4j.jar` in the class path entry of the manifest of the JAR file containing the log wrapper class.

Note: Although there can be only one instance of a Log4J `Category` class for a given component, multiple class loaders in the same process may load separate instances of the class. Because WebSphere can use different class loaders to load different components, there are circumstances where the logging of a component may behave unexpectedly. For more information on the WebSphere class loaders consult the WebSphere documentation in the InfoCenter.

Controlling application logging with Log4J

One of the strongest features of the Log4J framework is the degree to which it can be configured and extended. This discussion only touches briefly on some of the capabilities of the framework. If you are considering using Log4J in your own project, we encourage you to investigate the online Log4J documentation to gain a fuller picture. The documentation is available on the Web at:

http://jakarta.apache.org/log4j/docs/documentation.html

Our simple log wrapper uses a Log4J `PropertyConfigurator` to configure the logging framework using information supplied in a property file. The file uses the `name=value` format understood by the standard `java.util.Properties` class.

The contents of a sample configuration file is shown in Figure 13-29. We named the file `log4j.properties` and placed it in the `properties` directory of the WebSphere product directory, which is included in the class path of all WebSphere processes.

```
#
# Set root category priority to WARN and its only appender to FILE.
#
log4j.rootCategory=WARN,FILE

#
# Set the redbook EJB code priority to DEBUG
#
log4j.category.itso.was4ad.ejb=DEBUG

#
# FILE is a FileAppender that appends to D:\temp\trace.log
#
log4j.appender.FILE=org.apache.log4j.FileAppender
log4j.appender.FILE.File=D:/temp/trace.log

#
# FILE uses a PatternLayout
#
log4j.appender.FILE.layout=org.apache.log4j.PatternLayout
log4j.appender.FILE.layout.ConversionPattern=%d [%t] %-5p %c{1} - %m%n
```

Figure 13-29 Sample Log4J property configuration file

Log4J organizes components (categories in Log4J terminology) into a hierarchy, with each level separated by a `.` character, which in our case structures the hierarchy according to the Java package structure of the PiggyBank application. At the base of the hierarchy is the Log4J root category.

Enabled message priorities and message destinations are inherited through the hierarchy. The first line in the example in Figure 13-29 configures the root category, and hence all components that descend from it, to enable log messages of the Log4J `WARN` priority or higher. All messages in the hierarchy will be sent to the destination named `FILE`, which is defined later on in the configuration file.

The second configuration item enables `DEBUG` messages and higher—in other words all messages—for components in the hierarchy under `itso.was4ad.ejb`. This is all our EJB code. We could have chosen to specify one or more destinations for these messages—these would have been in addition to the destination inherited from the root category.

Message destinations are written to by Log4J *Appenders*. Appenders are implemented as Java classes. There are a number of standard appenders supplied with the Log4J framework, that can log to various destinations, such as rolling log files, UNIX system logs and Windows event logs. You can also implement your own appenders to meet your specific requirements. We are using a basic `FileAppender` that writes messages to a text file.

The final part of the configuration file specifies the format of the output message written to the file. It uses a standard Log4J layout—a `PatternLayout`—again you can implement your own layouts. The `PatternLayout` formats the log message by parsing a format string similar to that used by the `printf` function in C.

Our format string specifies that each log message includes the following items:

- A date and time stamp
- The ID of the thread that logs the message
- The message priority name
- The last part of the name of the component that logs the message
- The message itself
- A new-line character

An example of this message format can be seen in Figure 13-30, which shows the log statements that are written to the log file when the `transfer` method of the `AccountManager` EJB is invoked, passing in an invalid account ID.

```
2001-06-05 18:36:09,971 [ORB.thread.pool:1] DEBUG AccountManagerBean -
                                            setSessionContext()
2001-06-05 18:36:09,971 [ORB.thread.pool:1] DEBUG AccountManagerBean - ejbCreate()
2001-06-05 18:36:09,971 [ORB.thread.pool:1] DEBUG AccountManagerBean - transfer(432,
                                                                       604, 1000)
2001-06-05 18:36:09,971 [ORB.thread.pool:1] DEBUG AccountManagerBean - Looking up
                                                                       home
2001-06-05 18:36:09,982 [ORB.thread.pool:1] DEBUG AccountManagerBean - Locating debit
                                                                       account
2001-06-05 18:36:10,022 [ORB.thread.pool:1] DEBUG AccountBean - setEntityContext()
2001-06-05 18:36:10,022 [ORB.thread.pool:1] DEBUG AccountBean - ejbActivate()
2001-06-05 18:36:10,472 [ORB.thread.pool:1] DEBUG AccountBean - unsetEntityContext()
2001-06-05 18:36:10,512 [ORB.thread.pool:1] WARN  AccountManagerBean - transfer()
                                            caught finder exception
                                            javax.ejb.ObjectNotFoundException:
                                            itso.was4ad.ejb.account.AccountKey@1b0
```

Figure 13-30 Debug messages written using the Log4J log wrapper

Logging conclusions

In summary, we hope the discussions presented in this section have highlighted the need to include message logging as a fundamental component of a WebSphere application development project. We believe the benefits of implementing such a component at the earliest stages of development easily mitigate the effort involved, and ultimately lead to easier application deployment, and improved manageability in production. In real terms, for developers, this means fewer late nights and weekends getting systems live and fixing bugs in production.

Coding for performance

Every application has performance requirements. What is surprising however, is the number of projects that define no formal requirements in this area. In our experience as WebSphere consultants, we have encountered a frightening number of customer projects where the sole focus of the development team has been on application functionality, with little or no regard to the non-functional behavior of the application.

Although there many things you can do to tune an application once it is deployed into WebSphere, experience tells us that the most dramatic performance improvements—and performance problems—are driven by design and implementation of an application. We recommend you consider performance and scalability issues from the very beginning of your project, and set down development standards that encourage good practice.

In this section we outline some of the issues you should consider when building your application. Much of the information presented here comes from a WebSphere white paper, *Development Best Practices for Performance and Scalability*, by Harvey Gunther:

http://www-4.ibm.com/software/webservers/appserv/ws_bestpractices.pdf

General performance tips

The following performance tips are relevant to the development of any WebSphere application component.

Database connection pooling

Obtaining JDBC connections is a relatively expensive operation. WebSphere provides the facility to define and manage JDBC connections in pools that are made available to applications at runtime. In addition to performing better your application will also be easier to manage, because the connections can be defined and managed using the WebSphere tools.

An application obtains a pooled JDBC connection from a `javax.sql.DataSource` object obtained from a JNDI lookup, using code as shown in Figure 13-31.

```
import javax.naming.InitialContext;
import javax.sql.DataSource;
import java.sql.Connection;

...

// Lookup the DataSource and use it to obtain a Connection
InitialContext context = new InitialContext();
DataSource ds = context.lookup(“java:comp/env/jdbc/MyDataSource”);
Connection conn = ds.getConnection();

// Use the connection
...

// Release the connection back to the pool
conn.close();
```

Figure 13-31 Obtaining a JDBC connection from a connection pool

You have to declare the local JNDI reference to the JDBC resource `jdbc/MyDataSource` in the deployment descriptor of the application that uses this code. When the application is installed into WebSphere the reference is bound to a DataSource defined in the global JNDI namespace using the WebSphere administration tools.

You must also take care to ensure that every connection you obtain is returned to the pool promptly. Applications that 'leak' connections will quickly use up all available connections in the pool which will lead to significant problems in a production environment. Try to release the connection in the same method in which you obtain it—never hold on to a connection between user interactions. It is also good practice to put the `close` in a `finally` clause to ensure that the connection is always returned to the pool even if an exception is raised.

Using System.out

Invoking `System.out.println` can seriously degrade application throughput because the write to the standard output stream is synchronous. In the WebSphere environment standard error and output are redirected to disk files. The `println` method will not return until the information has been written to the file system. This can cause bottlenecks, because disk storage is relatively slow.

Instead of logging directly to standard out we recommend you develop another strategy for managing the logging of messages. We discuss this topic in detail in "Message logging" on page 330.

String concatenation

Manipulating `String` objects in Java is expensive, due to that fact that each string is represented by an immutable Java object. When you use the `+` or `+=` operators to concatenate strings temporary `String` objects are created and discarded. If you are going to perform string concatenation, a `java.util.StringBuffer` will perform better. For example the code:

```
String[] names = request.getParameters("name");
String msg = "Names:";
for (int i = 0; i < names.length; i++) {
   msg += " ";
   msg += names[i];
}
return msg;
```

Could be better written as:

```
String[] names = request.getParameters("name");
StringBuffer msg = new StringBuffer("Names:");
for (int i = 0; i < names.length; i++) {
   msg.append(" ");
   msg.append(names[i]);
}
return msg.toString();
```

JSP and servlet performance tips

The following performance tips are related to coding Web applications using servlets and JSPs.

Storing objects in HTTP sessions

The Java servlet API provides the `javax.servlet.http.HttpSession` class for use by applications wanting to store client-specific state information between requests from a client browser. In a single-server configuration this information may be stored in memory—in multi-server configurations WebSphere ensures this state information is available to all servers by persisting it to a database.

Although the application server provides mechanisms for caching session information and tuning the persistence mechanism, the best way to improve the performance of HTTP sessions is to minimize the amount of information you store in them. This leads us to the following guidelines for the use of HTTP sessions:

- Do not store large objects or object graphs in an HTTP session—if absolutely necessary store instead a reference to data persisted using JDBC or an entity EJB
- Use the session only for information required over multiple requests—use `setAttribute` on the `HttpRequest` object to pass results to JSPs
- Remove objects from sessions when they are no longer required
- Explicitly destroy sessions that are no longer required by calling `HttpSession.invalidate` from a logoff servlet

Using HTTP sessions in JSPs

By default a JSP always obtains a reference to a client's `HttpSession` object, even if the JSP does not use it. If the client does not already have an active session, the JSP will cause a new session to be created. This effect is particularly noticeable if your application uses frames in the client browser which use multiple JSPs. When you turn on session persistence WebSphere serializes access to the `HttpSession` object, so each JSP in the frameset has to wait for the page that currently has the session to complete before it can execute.

If a JSP does not use information stored in an HTTP session, prevent it from obtaining a session reference using the following JSP page directive:

```
<%@ page session="false" %>
```

Servlet multi-threading

By default WebSphere creates just one instance of each servlet class, and allows multiple threads to execute the servlet's `service` method concurrently. Because of this access to servlet instance variables must be controlled by Java synchronization mechanisms, which can quickly become a bottleneck in an application. Code your application to avoid such bottlenecks by keeping as many variables in your servlets as possible on the stack, declaring them at the method level. Although it may be tempting to use the `SingleThreadModel` feature of the servlet API to allow you to disregard these issues, try not to as it prevents WebSphere from pooling servlet resources in the most efficient manner.

EJB performance tips

The following tips relate to performance when coding with Enterprise Java Beans.

Accessing EJBs from client code

When you design your EJBs try to design your remote interfaces so that clients can perform all the processing required for a single user interaction by invoking a single method. This improves performance in two ways:

- First of all the work requested by the client can be performed, where appropriate, in a single EJB transaction. Managing transactions incurs a significant amount of overhead, especially if the transaction involves a two-phase commit between multiple participants. Reducing the number of transactions that WebSphere has to manage reduces the number of times this overhead is incurred. Entity EJBs also synchronize their persistent fields from and to the database once per transaction. Encapsulating the request into a single transaction may also be desirable, or even required, for your business logic.
- Second, the number of remote method calls may be reduced. With standalone clients, and potentially with servlets, the actual EJBs are deployed into a separate process from the EJB client. Under these circumstances communication between the EJB client and server code involves additional network overhead. When multiple EJBs are deployed into the same server process WebSphere is able to optimize out the network interaction.

You should also consider eliminating the getter and setter methods from entity EJBs that are generated using VisualAge for Java. Although these methods are convenient they can encourage the proliferation of RMI calls as clients of the EJB use each method in turn. Replace getters and setters with business methods that enforce correct behavior according to your business logic, rather than allow clients to modify data fields arbitrarily. If you find you do need to extract all of the data from an entity EJB, for displaying to a user, for example, consider the use of bulk getter methods. These are serializable data-only objects that can be obtained from an entity by a single method call.

Entity EJB read-only methods

WebSphere allows you to specify that individual methods of an entity EJB are read-only—that is to say they do not modify the bean's persistent fields. If read-only methods are the only methods invoked on an entity EJB instance during the course of a transaction WebSphere is optimizes out the `ejbStore` operation that stores the bean's persistent fields back into the database.

Read-only methods are specified using the WebSphere Application Assembly Tool (AAT). To specify a bean method as being read-only, load the module containing the EJB into AAT and expand the tree view in the left hand pane to reveal the Method Extensions item for the EJB (Figure 13-32).

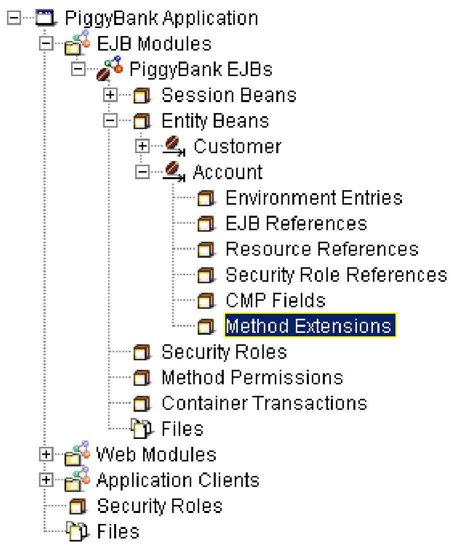
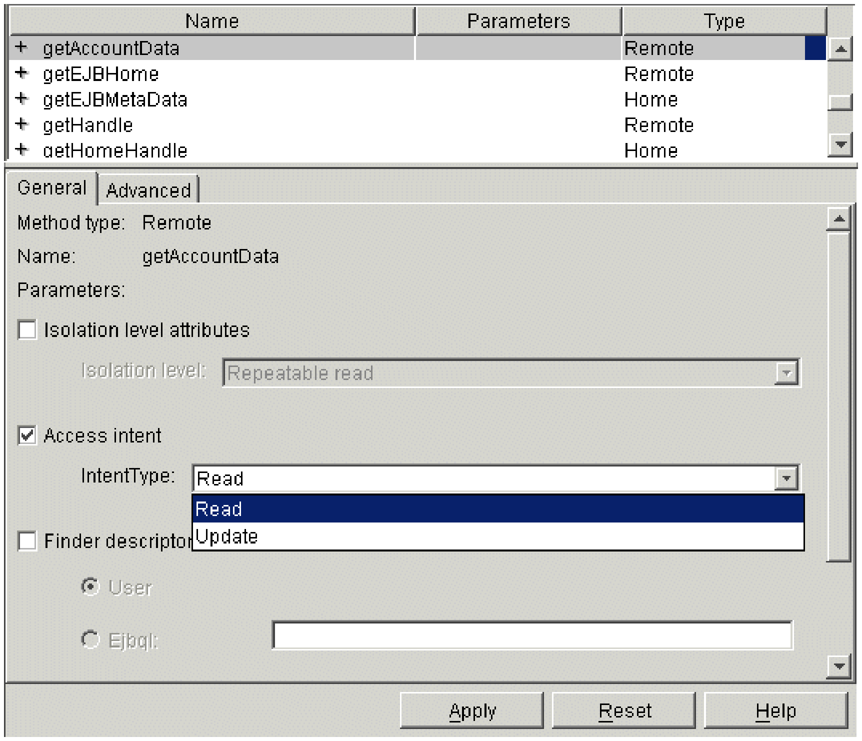


Figure 13-32 EJB Method Extensions item in the assembly tool tree view

Next, in the right hand pane, select the method you want to change, and check the *Access intent* box (Figure 13-33). Select *Read* from the drop-down box labeled *Intent Type* and click *Apply* to confirm the change.



Name	Parameters	Type
+ getAccountData		Remote
+ getEJBHome		Remote
+ getEJBMetaData		Home
+ getHandle		Remote
+ getHomeHandle		Home

General | Advanced

Method type: Remote

Name: getAccountData

Parameters:

☐ Isolation level attributes

Isolation level: Repeatable read

☑ Access intent

IntentType: Read

Read

Update

☐ Finder descriptor

● User

○ Ejbql:

Figure 13-33 Declaring the getAccountData method as being read-only

Because the read-only optimization is a WebSphere extension to the EJB specification, the information we enter is saved into the `ibm-ejb-jar-ext.xmi` deployment descriptor file saved in the EJB JAR file. Figure 13-34 shows the XML fragment that is inserted in this file as a result of setting the `getAccountData` method to be read-only.

```
<accessIntents xmi:id="AccessIntent_2" intentType="READ">
  <methodElements xmi:id="MethodElement_9" name="getAccountData" parms=""
                  type="Remote">
    <enterpriseBean xsi:type="ejb:ContainerManagedEntity"
                    href="META-INF/ejb-jar.xml#Account"/>
  </methodElements>
</accessIntents>
```

Figure 13-34 Read-only method information in the deployment descriptor

Isolation levels

Isolation levels are used to specify the degree to which an EJB transaction accessing a resource may be affected by other concurrent transactions accessing the same resource. For entity EJBs mapped to a relational database the isolation level determines the locking policy used when accessing a database. Isolation levels are discussed in detail in the WebSphere InfoCenter documentation.

The most strict isolation level is *Serializable*. An EJB that specifies this isolation level is guaranteed to get consistent results from the database for the duration of each transaction.

To achieve this behavior, every row that satisfies an SQL `SELECT` issued by the EJB or the underlying persistence layer is locked for the duration of the transaction. In development, where individual developers may have separate databases or separate schemas in a single database, this may not cause any problems. In a production system with multiple concurrent clients this strict locking can cause significant bottlenecks.

Note: The mapping of WebSphere isolation levels to actual behavior in the database is resource manager-specific. The same isolation level in WebSphere can produce different results with DB2 than with Oracle, for example.

The default isolation level assigned to EJBs by the WebSphere tools is *Repeatable read*. While not as strict as Serializable, this isolation level can still cause bottlenecks.

The isolation level *Read committed* is adequate for many applications. Although a more strict isolation level may appear to be a safer choice, you must consider the impact of such a change on the performance of the application in production.

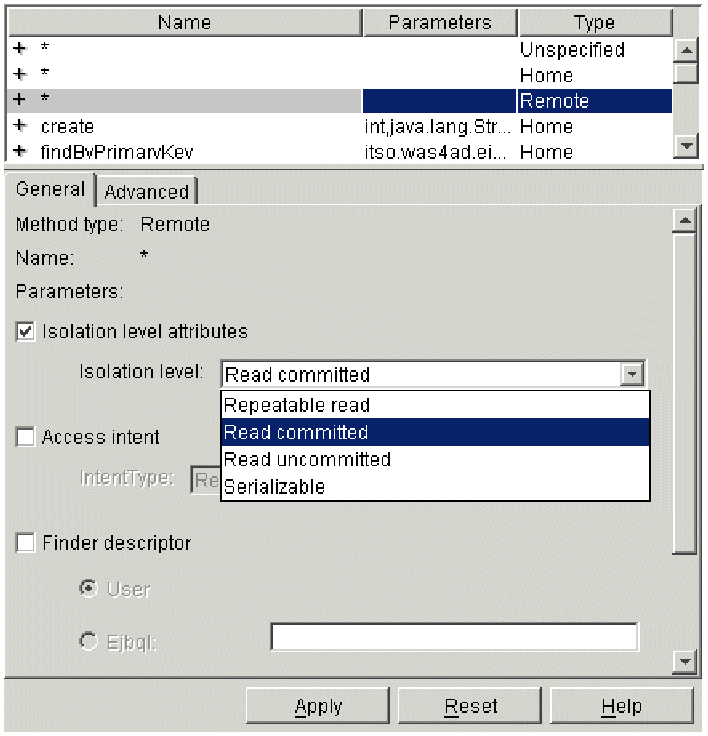
Therefore, do not sacrifice application integrity for the sake of performance by using an isolation level that does not provide the level of protection your business logic requires.

Important: Every participant in a transaction accessing the same resource manager must share the same isolation level. If isolation levels differ the transaction will be rolled back.

To modify the isolation level you must edit the EJB module using the WebSphere AAT tool:

- Select the *Method Extensions* item in the tree view for the EJB you want to modify, as described in “Entity EJB read-only methods” on page 367.
- To change the isolation level for a particular method, select that method from the list at the top of the right hand pane.
- To change the level for all methods in the home, remote, or both the home and remote interface, select the appropriate entry labeled with an asterisk (*) for the name.

Check the *Isolation level attributes* box and select the appropriate isolation level from the list (Figure 13-35). Once you have made your selection click *Apply* to save the changes.



Name	Parameters	Type
+ *		Unspecified
+ *		Home
+ *		Remote
+ create	int,java.lang.Str...	Home
+ findByPrimaryKey	itso.was4ad.ei...	Home

General | Advanced

Method type: Remote

Name: *

Parameters:

[✓] Isolation level attributes

Isolation level: Read committed

- Repeatable read
- Read committed
- Read uncommitted
- Serializable

[ ] Access intent

IntentType: Re

[ ] Finder descriptor

(•) User

( ) Ejbql:

Apply | Reset | Help

Figure 13-35 Modifying the isolation level for all methods in the remote interface

Managing application versions

It is often desirable to be able to run two versions of the same application inside the same WebSphere cluster, or even on the same server. There are two scenarios which usually drive this requirement:

- A single large server—often but not necessarily a UNIX server—must be shared between multiple developers, each of which needs his own independent WebSphere environment in which to work.
- Two or more copies of the same application, either the same or different versions, must be hosted in the same environment—examples include to allow parallel running and final user acceptance testing of a new release, or to allow two different user communities access to the same application accessing separate data.

The introduction of the single server edition of WebSphere Application Server with Version 4.0 goes some way to addressing these problems—these lightweight editions do not require a database as an administrative repository and are generally easier to manage, making it much more feasible to have a copy on every developer's desktop machine.

There are still valid scenarios, however, where you want to be able to host multiple applications in the same instance or cluster of the advanced edition of WebSphere. The single server edition does not provide any workload management or clustering capabilities, for example. If your developers have Windows desktops but your deployment environment is UNIX it is a good idea to test your application on the target platform from early in the development cycle—in some cases it may be absolutely necessary, perhaps because of other software components that are only available in the deployment environment.

In this section we describe how to install multiple versions of the same application into separate application server processes running in the same WebSphere cluster.

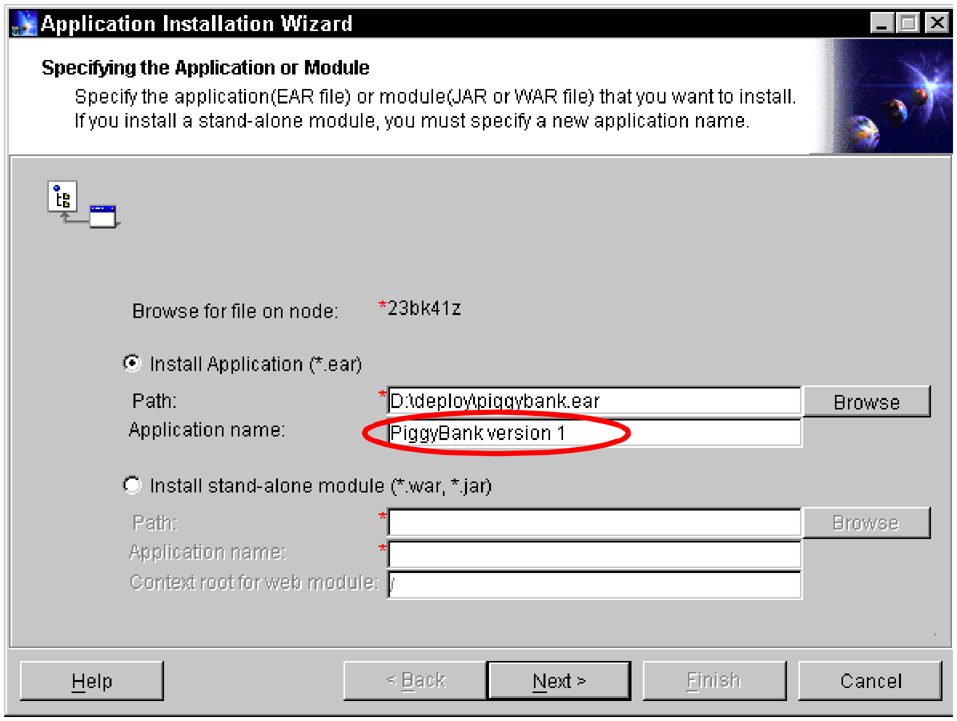
There are four areas where two instances of the same application may conflict when running in the same WebSphere cluster:

- The application name
- The Web application in the URI namespace
- The EJBs in the JNDI namespace
- Access to database and other resources

While we cover the case where the two instances have to connect to different databases. The scenario where the two versions accessing the *same* database require different database schemas is beyond the scope of this discussion.

Specifying the application name

The two versions of the application installed in WebSphere may not share the same name. To separate the names you must enter a new name into the first panel of the WebSphere application installation wizard (Figure 13-36).



Specifying the Application or Module

Specify the application(EAR file) or module(JAR or WAR file) that you want to install.
If you install a stand-alone module, you must specify a new application name.

Browse for file on node: *23bk41z

● Install Application (*.ear)

Path: *D:\deploy\piggybank.ear

Application name: PiggyBank version 1

○ Install stand-alone module (*.war, *.jar)

Path: *

Application name: *

Context root for web module: /

Figure 13-36 Specifying a unique name for the application

Partitioning Web applications in the URI namespace

There are two approaches to partitioning the URI namespace:

- Use the same Web application context root, and different virtual hosts
- Use the same virtual host, but different Web application context roots

In general the first approach may be more appropriate for a production environment, and the second more appropriate in a development environment. This is due mainly to the fact that you will have many developers and over time new staff will be recruited and existing staff move on to other projects. Managing aliases in DNS for each developer in this environment is an unnecessary burden.

Partitioning using virtual hosts

In this scenario we create a new virtual host for each version of the application. If we call our two versions `version1` and `version2`, for example, we need to perform the following actions:

- Define host aliases for both `version1` and `version2`, that resolve to the Web server IP address, or the address of the load balancing component in a multi-server configuration
- Use the WebSphere administrator's console to define the two virtual hosts to WebSphere
- Specify binding information for the Web application that binds the two versions of the application to separate virtual hosts

Defining host aliases

In a production environment we would ask our DNS administrator to create the two aliases for our production machine. In this example we simply create aliases by editing the file `C:\WINNT\system32\drivers\etc\HOSTS`, adding the two additional lines shown below:

```
127.0.0.1    version1
127.0.0.1    version2
```

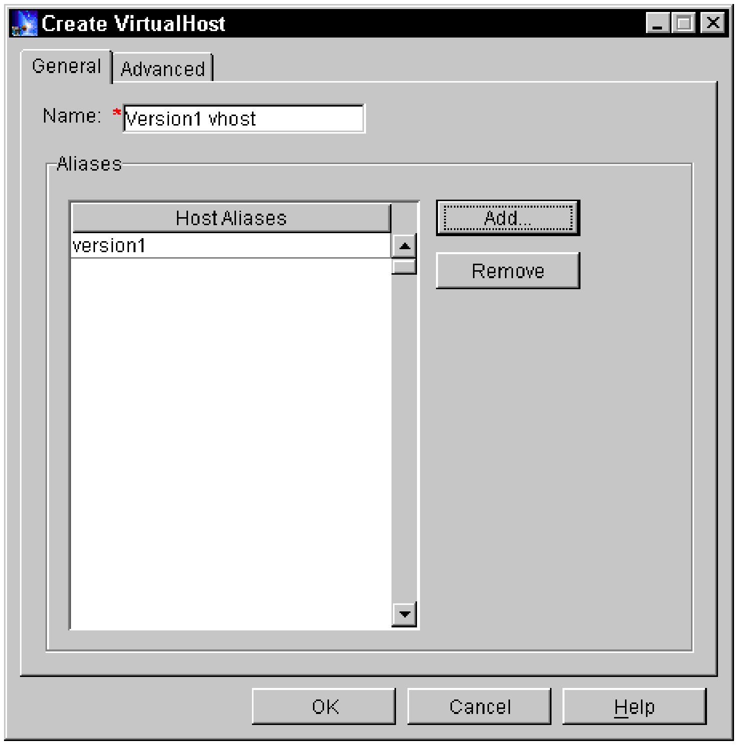
Tip: If you follow these instructions you will only be able to use the virtual hosts from the local machine where they are defined. To correctly define the host aliases they must be defined globally.

Important: You must also remember to configure your Web server to recognize the new virtual hosts.

Defining WebSphere virtual hosts

We have to define two new virtual hosts to WebSphere to allow WebSphere to route requests to each virtual host correctly.

To define a new virtual host, click the *Virtual Hosts* folder in the administrator's console tree view, and select *New* from the pop-up menu. This opens the Create Virtual Host dialog (Figure 13-37).



Name: *Version1 vhost

Host Aliases
version1

Figure 13-37 Defining a new WebSphere virtual host

In the Create Virtual Host dialog, enter a name for the new virtual host in the Name field. This is the name that will appear in the console. To add a host alias for this virtual host, click the *Add* button, enter the host name in the dialog that pops up, then click *OK*.

Remember to include the port number if the Web server listens on a port other than port 80; if the web server is using SSL the port is usually set to port number 443.

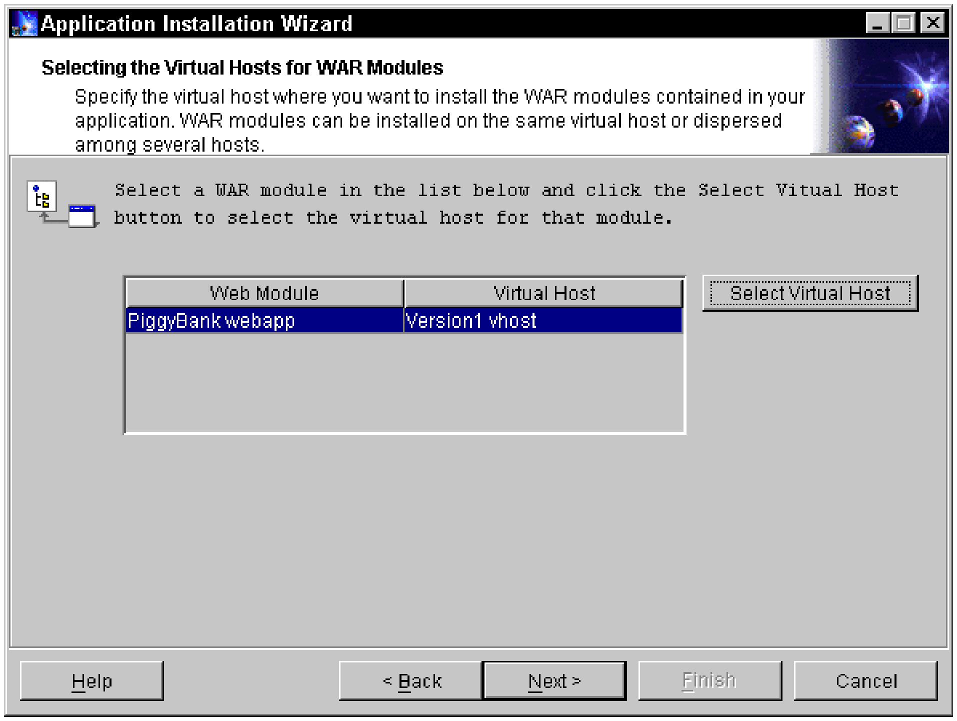
Specifying binding information

You can specify the binding between a Web application and the virtual host in one of two ways:

- At installation time using the installation wizard
- Prior to deployment using the Application Assembly Tool (AAT)

Specifying binding information using the installation wizard

Use the administrator's console to install the module containing the application. When you get to the page that specifies the virtual host to use (Figure 13-38), select the Web module, and click *Select Virtual Host*.



Application Installation Wizard

Selecting the Virtual Hosts for WAR Modules

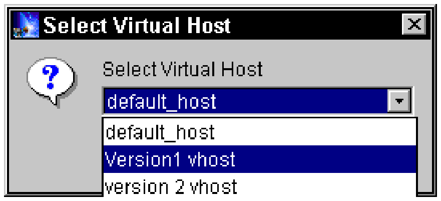
Specify the virtual host where you want to install the WAR modules contained in your application. WAR modules can be installed on the same virtual host or dispersed among several hosts.

```
Select a WAR module in the list below and click the Select Vitual Host
button to select the virtual host for that module.
```

Web Module	Virtual Host
PiggyBank webapp	Version1 vhost

Figure 13-38 Specifying the virtual host using the deployment wizard

In the dialog that pops-up (Figure 13-39), select the appropriate virtual host from the drop-down list, and click *OK*.



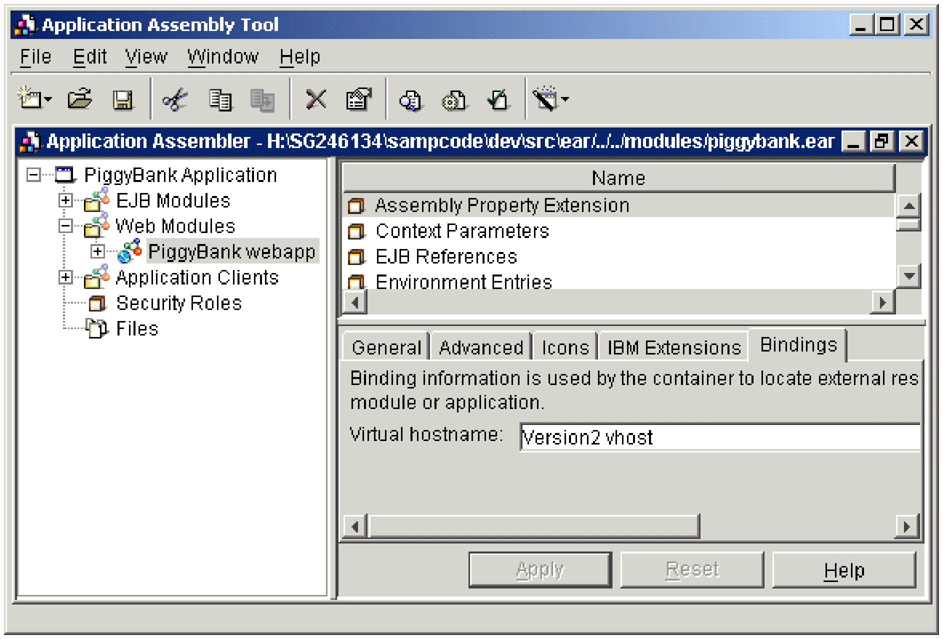
Select Virtual Host

Select Virtual Host: default_host

Figure 13-39 Selecting the virtual host

Specifying binding information using the assembly tool

You may also specify the binding information using AAT. Start the assembly tool and open the module containing your Web application. Use the tree view to select the Web application (Figure 13-40). Click the Bindings tab, enter the virtual host name in the virtual host name field, and click *Apply*.



Binding information is used by the container to locate external res module or application.

Virtual hostname: Version2 vhost

Figure 13-40 Specifying the virtual host name using the assembly tool

When you deploy the module into WebSphere, the virtual host you set using AAT is automatically selected.

Partitioning using the Web application context root

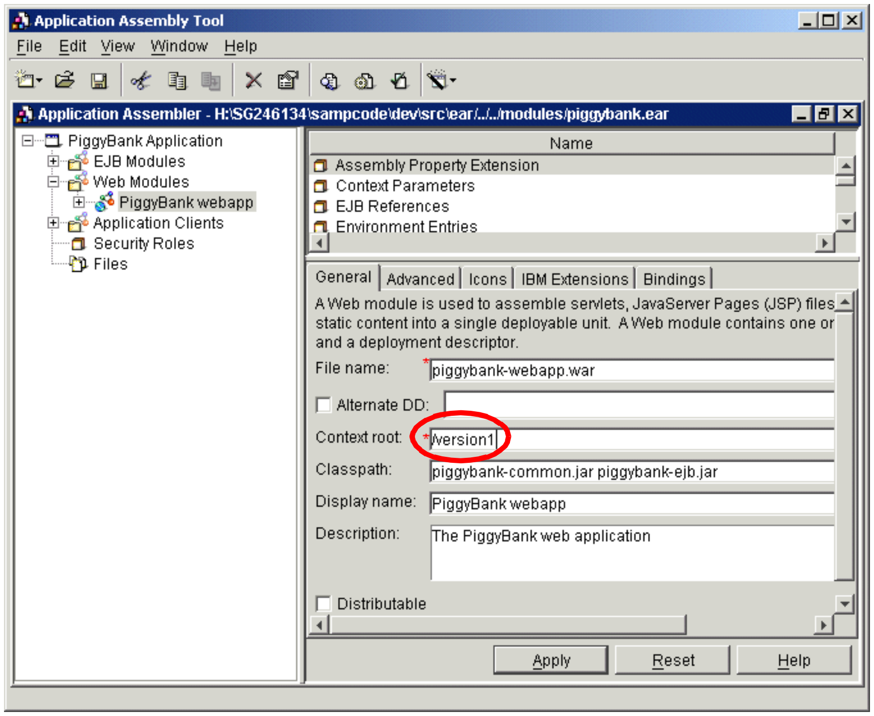
The context root of a Web application module defines the base URI of the application relative to the base of the Web server URI. If we modify the context root for our Web application to include the version identifier, we can separate the namespace using the same virtual host, so for example, one version of our application can occupy the URIs under:

```
http://hostname.domainname.com/version1/
```

The other version uses URIs under:

```
http://hostname.domainname.com/version2/
```

The context root binding for a Web application is specified using the WebSphere assembly tool. We start the tool and open the EAR file containing our Web application. First we use the tree view to navigate to the Web application (Figure 13-41). We then select the General tab and enter the new context root in the field labeled Context root. Finally we click *Apply* and save the modified module file.



A Web module is used to assemble servlets, JavaServer Pages (JSP) files static content into a single deployable unit. A Web module contains one or and a deployment descriptor.

File name: piggybank-webapp.war

Alternate DD:

Context root: version1

Classpath: piggybank-common.jar piggybank-ejb.jar

Display name: PiggyBank webapp

Description: The PiggyBank web application

Distributable

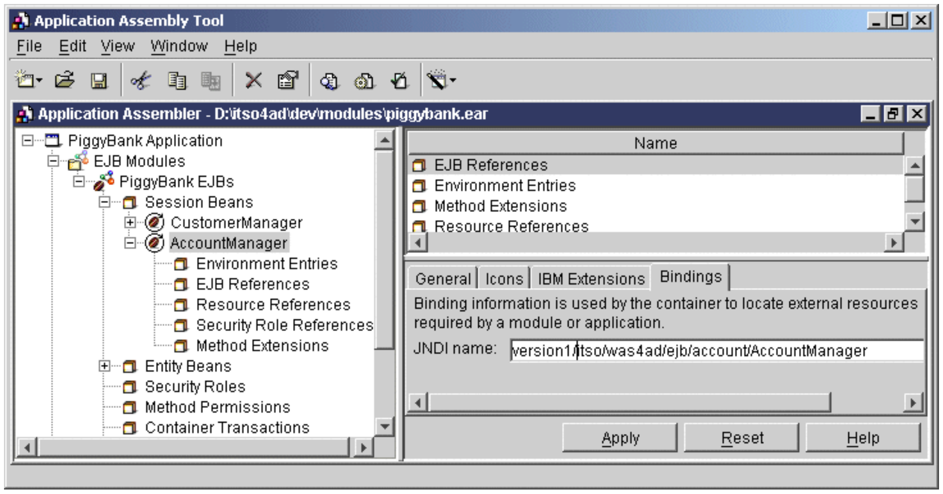
Figure 13-41 Changing the Web application context root using the assembly tool

Partitioning EJBs in the JNDI namespace

When we install the EJBs into the application server we specify the names that the beans will use in the global JNDI namespace—for example the `AccountManager` EJB might use the global JNDI name:

```
itso/was4ad/ejb/account/AccountManager
```

We can define the global JNDI name to use for each EJB using AAT or the WebSphere application installation wizard. Figure 13-42 shows how AAT can be used to specify the JNDI name for an EJB.

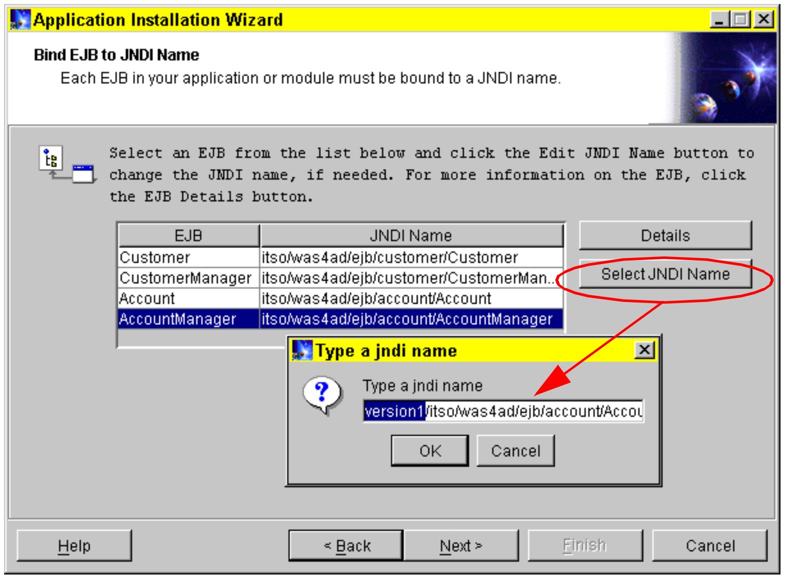


Binding information is used by the container to locate external resources required by a module or application.

JNDI name: version1/itso/was4ad/ejb/account/AccountManager

Figure 13-42 Specifying an EJB JNDI name using AAT

The global JNDI name can also be specified when the application is installed into the WebSphere environment. Figure 13-43 shows how the application installation wizard invoked from the administrator's console can be used to specify or override the binding of an EJB to a JNDI name.



Bind EJB to JNDI Name

Each EJB in your application or module must be bound to a JNDI name.

```
Select an EJB from the list below and click the Edit JNDI Name button to
change the JNDI name, if needed. For more information on the EJB, click
the EJB Details button.
```

EJB	JNDI Name
Customer	itso/was4ad/ejb/customer/Customer
CustomerManager	itso/was4ad/ejb/customer/CustomerMan...
Account	itso/was4ad/ejb/account/Account
AccountManager	itso/was4ad/ejb/account/AccountManager

Type a jndi name

version1/itso/was4ad/ejb/account/Accou

Figure 13-43 Specifying an EJB JNDI name in the application installation wizard

If we have to run two versions of the same EJBs concurrently in the same cluster, we must ensure the global JNDI names are unique. In our example with two versions, we could decide to install the first version of the account manager as:

```
version1/itso/was4ad/ejb/account/AccountManager
```

and the second version as:

```
version2/itso/was4ad/ejb/account/AccountManager
```

Our application's EJB client code uses *EJB references* to locate EJB home objects. All of our applications that use EJBs look for them in the EJB client's local JNDI namespace, under `java:comp/env`, for example:

```
java:comp/env/ejb/AccountManager
```

These names are coded as constants in the classes that are EJB clients. When we use the assembly tool to build our application we declare the references used by each application client, Web application and EJB component. Different components may use different local names to refer to the same EJB, or the same local name to refer to different EJBs. When a component looks up an EJB the WebSphere runtime maps the local EJB reference to the correct global JNDI name for the EJB that particular component needs.

When we install client code that uses the EJB, we simply specify the correct binding for the version we require, without any need to modify our own code (Figure 13-44).

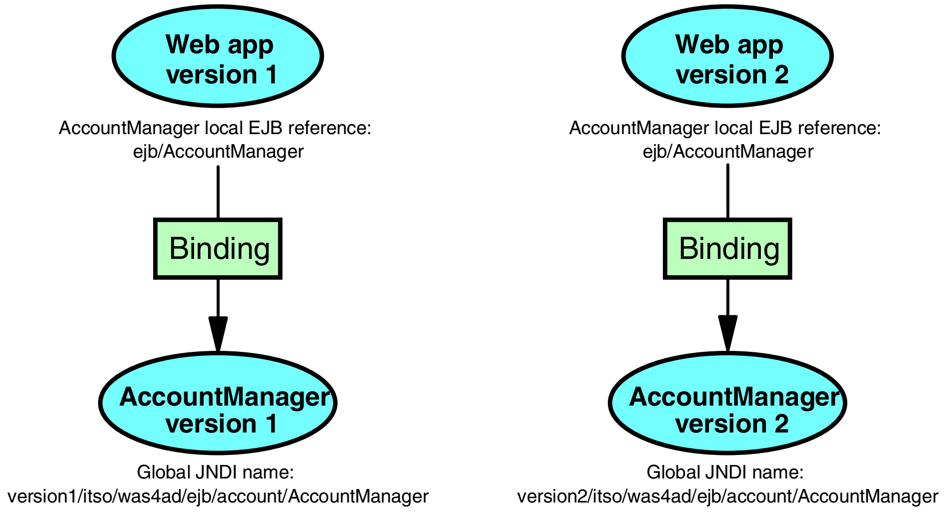
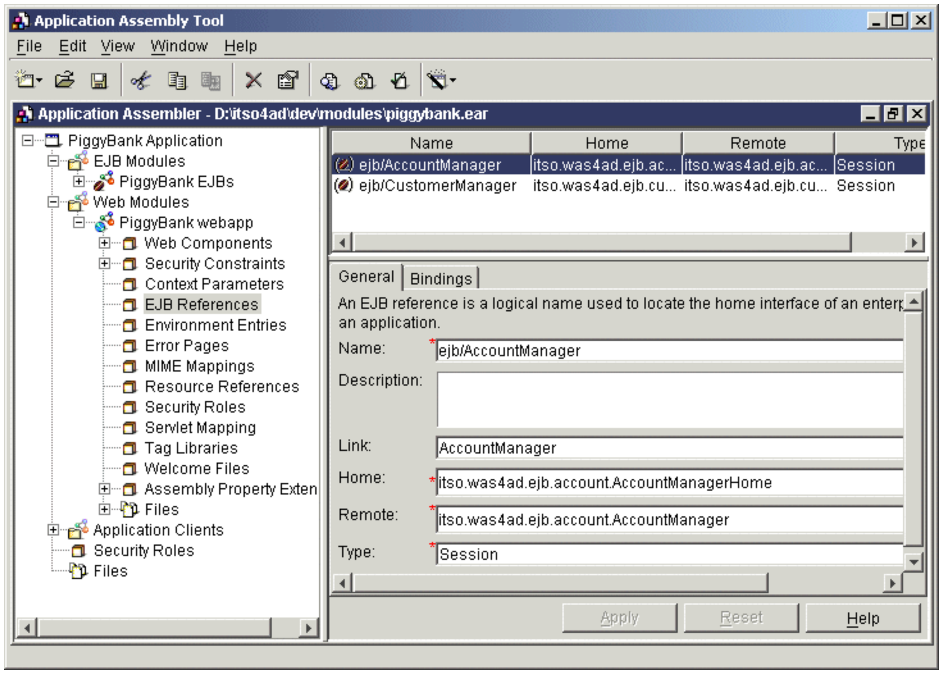

Web app version 1
AccountManager local EJB reference: ejb/AccountManager
Binding
AccountManager version 1
Global JNDI name: version1/itso/was4ad/ejb/account/AccountManager
Web app version 2
AccountManager local EJB reference: ejb/AccountManager
Binding
AccountManager version 2
Global JNDI name: version2/itso/was4ad/ejb/account/AccountManager

Figure 13-44 Binding EJB references to JNDI names

The local EJB references used by a module are described in the module's deployment descriptor.

We use the WebSphere AAT tool to manage EJB references (Figure 13-45).



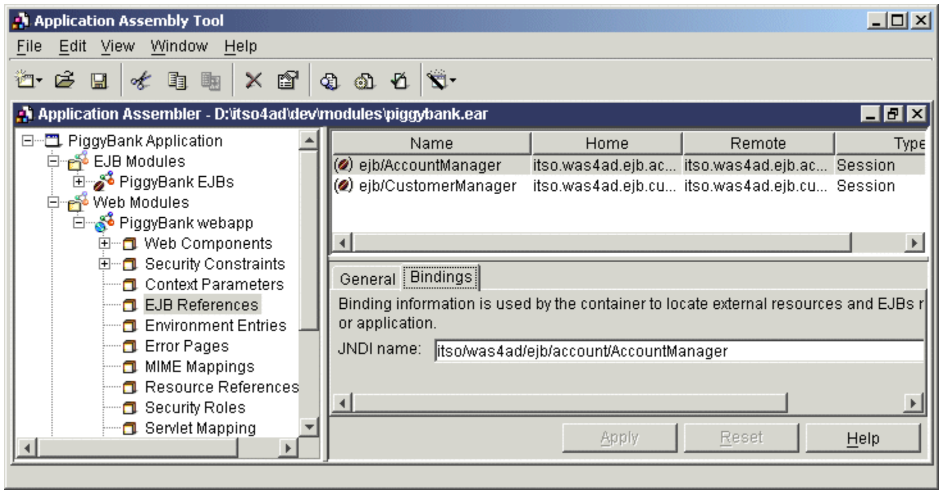
Name	Home	Remote	Type
ejb/AccountManager	itso.was4ad.ejb.ac...	itso.was4ad.ejb.ac...	Session
ejb/CustomerManager	itso.was4ad.ejb.cu...	itso.was4ad.ejb.cu...	Session

An EJB reference is a logical name used to locate the home interface of an enterp an application.

Name: ejb/AccountManager
Description:
Link: AccountManager
Home: itso.was4ad.ejb.account.AccountManagerHome
Remote: itso.was4ad.ejb.account.AccountManager
Type: Session

Figure 13-45 Managing EJB references using AAT

We can also define the binding that maps each EJB reference to a JNDI name using AAT, using the Binding tab (Figure 13-46).



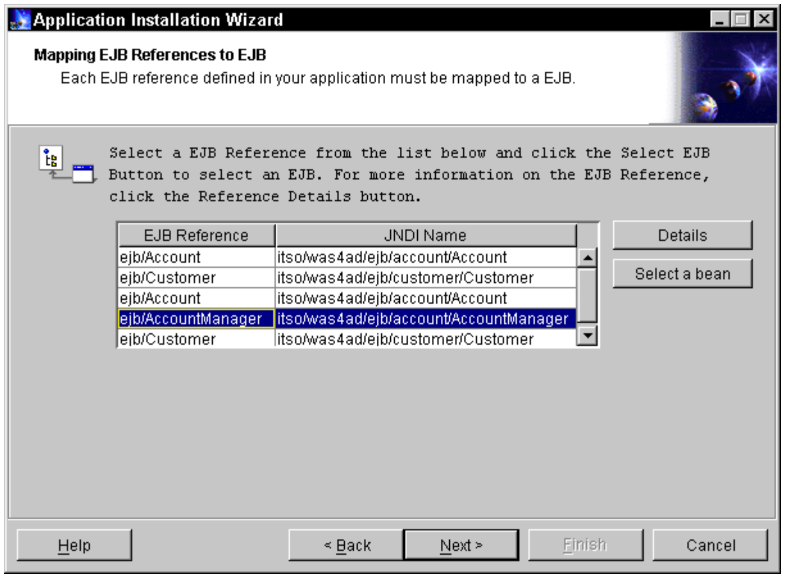
Name	Home	Remote	Type
ejb/AccountManager	itso.was4ad.ejb.ac...	itso.was4ad.ejb.ac...	Session
ejb/CustomerManager	itso.was4ad.ejb.cu...	itso.was4ad.ejb.cu...	Session

Binding information is used by the container to locate external resources and EJBs r
or application.

JNDI name: itso/was4ad/ejb/account/AccountManager

Figure 13-46 Specifying an EJB reference binding using AAT

Alternatively we can specify or modify the binding when we install the application using the application installation wizard (Figure 13-47).



Mapping EJB References to EJB

Each EJB reference defined in your application must be mapped to a EJB.

Select a EJB Reference from the list below and click the Select EJB Button to select an EJB. For more information on the EJB Reference, click the Reference Details button.

EJB Reference	JNDI Name
ejb/Account	itso/was4ad/ejb/account/Account
ejb/Customer	itso/was4ad/ejb/customer/Customer
ejb/Account	itso/was4ad/ejb/account/Account
ejb/AccountManager	itso/was4ad/ejb/account/AccountManager
ejb/Customer	itso/was4ad/ejb/customer/Customer

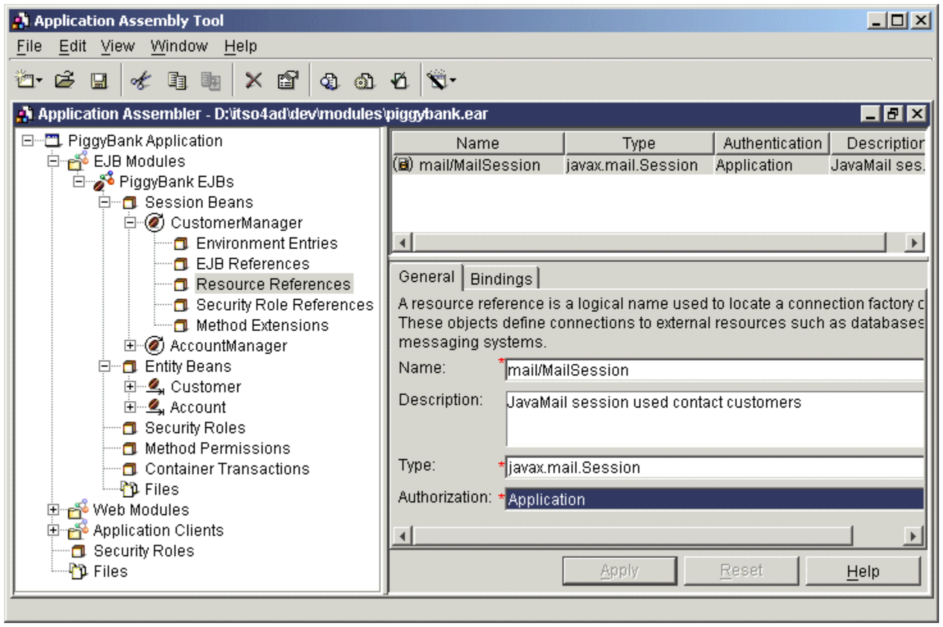
Figure 13-47 Specifying an EJB reference binding in the installation wizard

Partitioning access to database and other resources

We can use a similar approach to that used for EJBs to define bindings to other resources, including JDBC DataSources and JMS queue connection factories. In each case the code that requires the resource should perform the JNDI lookup in the local JNDI namespace, under `java:comp/env`. For example, to lookup a JavaMail session you could write the following code:

```
InitialContext context = new InitialContext();
Session sess = (Session) context.lookup("java:comp/env/mail/MailSession");
```

The local JNDI name used must be defined in the component's deployment descriptor, which can be edited using AAT (Figure 13-48).



Name	Type	Authentication	Descriptior
mail/MailSession	javax.mail.Session	Application	JavaMail ses.

A resource reference is a logical name used to locate a connection factory c
These objects define connections to external resources such as databases
messaging systems.

Name: * mail/MailSession

Description: JavaMail session used contact customers

Type: * javax.mail.Session

Authorization: * Application

Figure 13-48 Editing a JavaMail resource reference using AAT

When you install a new application version define the resources that are unique to the new version with unique global JNDI names in the WebSphere environment, using the administrator's console or other appropriate WebSphere tools such as `wscp`. You can then specify the bindings for the new application version using AAT, or during installation using the application installation wizard.

Automation opportunities

There are two opportunities for automation that can assist you in managing an environment with multiple application versions:

- Creating ready-bound modules
- Scripting installation of application versions

These opportunities are outlined below.

Creating ready-bound modules

If you are using Ant to create your modules, perhaps in a manner similar to that described in "Using Ant to build a WebSphere application" on page 197, you can extend your build process to automatically insert version information into the binding information stored in the J2EE modules.

One way to do this is to use the Ant built-in `replace` task. This task copies a source file to a new location, replacing occurrences of named tags with a specified value. Tags are specified in the source file using the syntax:

```
@TAG_NAME@
```

If we define a tag `@VERSION@` in our source deployment descriptors, we can use the `replace` task to automatically insert a version identifier into the files that specify deployment bindings, as we build the modules that make up our application.

Scripting installation of application versions

You may find that you plan to install new application versions on a regular basis, because, for example, you have a single large machine that is used as a shared development or test machine. Under these circumstances it will probably be worth creating scripts that can automatically install and uninstall application versions, creating all the appropriate resources such as JDBC DataSources and JMS queue connection factories that each application version requires.

WebSphere provides tools which enable you to perform these administration tasks from scripts or the command line, most notably the `wscp` tool. A description of this tool is out of the scope of this publication, however.

14

Software Configuration Management

In this chapter we touch on the challenging area of Software Configuration Management (SCM), and how it relates to WebSphere Studio and VisualAge for Java.

While the starting point of customer involvement with SCM varies, no customer can afford to ignore this area. In fact, after implementing SCM processes, the resulting improvements in IT reaction times to meet business demands could well prove to be a key success factor for being successful with e-business.

Introduction

Definition: The U.S. Department of Defense, in its standard on software development, DOD-STD-2167A, defines SCM as follows:

> *Software Configuration Management is the discipline of identifying the configuration of software systems at discrete points in time for the purpose of controlling changes and maintaining traceability of changes throughout the software life cycle.*

SCM is one of the key areas that has to be addressed when developing and maintaining applications. This is not only true for managing the software configuration within your development environment, but also applies to the software configuration within the production environment.

Application architectures, methodologies, technologies, and associated tools put into a development process context potentially fail on delivering the appropriate functionality to the business if SCM processes and supporting tools are not in place.

Pressure to deliver faster and more complex applications makes it more urgent to implement SCM. At the same time, businesses that are developing and deploying applications in the e-business space may find themselves open to exposure when SCM problems occur.

This calls for an end-to-end (E2E) approach for SCM throughout the complete application life cycle. However, addressing all aspects of SCM would be a book in itself.

Reference

When writing this redbook we ran out of time to investigate and test a complete SCM approach.

We therefore refer to Chapter 9. Software Configuration Management in the redbook *Servlet and JSP Programming with IBM WebSphere Studio and VisualAge for Java*, SG24-5755.

In the referenced redbook we documented an SCM approach and showed how to implement SCM with Rational ClearCase for WebSphere Studio.

Part 4

Unit testing the application

In this part of the book we provide information to assist you in unit testing and debugging a WebSphere Version 4 application.

First we describe how to use the WebSphere Application Assembly Tool (AAT) to assemble the application into J2EE modules that can be deployed into a WebSphere environment for testing. We then describe the deployment process itself, explaining how to deploy into both the full and single-server versions of WebSphere Application Server.

After that we discuss application debugging, and describe how we can use the facilities provided by VisualAge for Java and the IBM Online Trace (OLT) and Distributed Debugger products to debug WebSphere applications.

Finally we introduce JUnit, an open source framework for unit testing Java applications. We describe how to use JUnit to create test cases and discuss how to use the tool to unit test EJB components running in WebSphere.

15

Assembling the application

In this chapter we describe the process of assembling a J2EE application using the tools available for WebSphere Application Server 4.0 Advanced Edition (AE) and Single Server Edition (AEs).

We describe the assembly process using the following tools:

- Application Assembly Tool (AAT)
- `ejbdeploy` command line tool

Both of them are new features in WebSphere Application Server 4.0.

We focus on the basic aspects of the assembly process, skipping through options that are not necessary in a unit test scenario.

We also discuss how the `ejbdeploy` can be used to help validate and migrate version 1.0 EJBs to the new 1.1 specification level, and describe the new XML-based CMP persistence mapping supported by the tool.

Application Assembly Tool (AAT)

The Application Assembly Tool is a new feature introduced in WebSphere 4.0, both for Advanced Edition and Single Server Edition. It allows packaging and assembling an enterprise application and its modules according to the J2EE specification.

For details about J2EE packaging, see Chapter 8, “Setting up a development environment” on page 179.

Note: The assembly tool allows us to specify installation-specific information in addition to its role in providing a GUI facility for defining J2EE modules. These entries are known as *bindings*, because they bind an application to a specific WebSphere installation, specifying global JNDI names, and user IDs and passwords for databases, for example.

This is a useful shortcut for administrators who do not want to specify the global JNDI name for every EJB every time they install a new version of an application. Binding information is supplemental to the J2EE deployment information, however, and is not portable to other application server vendors’ products.

Starting the Application Assembly Tool

From the command line

The bat file that starts the Application Assembly Tool, `assembly.bat`, is located in

```
d:\WebSphere\AppServer\bin
```

If we start the tool from a command window, this window remains in the background, and it must not be closed, or the assembly tool is closed as well. This window displays tracing information for changes in the properties of the module’s elements.

From the WAS Administrative Console (AE only)

Before we can start the Administrative Console, we must first start the AdminServer: go to *Start -> Programs -> IBM WebSphere -> Application Server v4.0 -> Start AdminServer*, or either start the *IBM WS AdminServer* service from the Windows Services panel.

It is also possible to start the AdminServer from the command line, typing:

```
net start “IBM WS AdminServer“
```

Then start the Application Server's Administrator's Console, by selecting *Start -> Programs -> IBM WebSphere -> Application Server v4.0 -> Administrator's Console.*

To start the Application Assembly Tool from the Administrative Console, select *Tools -> Application Assembly Tool*.

Using the interface

The first view onto the Application Assembly Tool is a general menu that allows the user to create any of the J2EE modules:

- Application (EAR file)
- Web module (WAR file)
- EJB module (EJB JAR file)
- Application client (JAR file)

It is possible to perform these creation tasks either using the property dialogs or the corresponding wizard. Wizards require minimum information to complete the process. They ask for the required information and fill in the rest with the default values.

The welcome screen is shown in Figure 15-1.

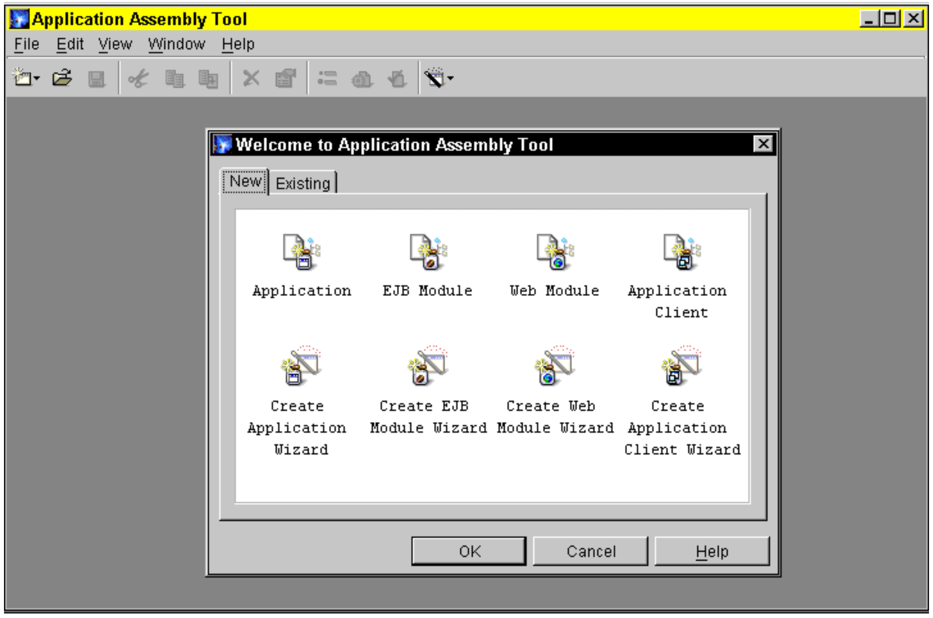
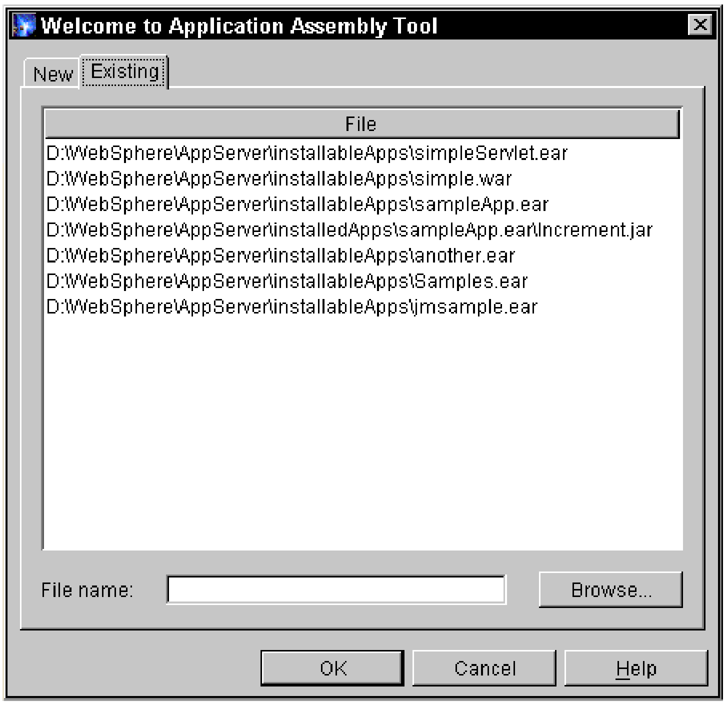

Application Assembly Tool
File Edit View Window Help
Welcome to Application Assembly Tool
New Existing
Application
EJB Module
Web Module
Application Client
Create Application Wizard
Create EJB Module Wizard
Create Web Module Wizard
Create Application Client Wizard
OK
Cancel
Help

Figure 15-1 Application Assembly Tool welcome screen

It is also possible to open an existing module (of any of the previously listed types), by selecting the *Existing* tab, as it is shown in Figure 15-2.



D:\WebSphere\AppServer\installableApps\simpleServlet.ear
D:\WebSphere\AppServer\installableApps\simple.war
D:\WebSphere\AppServer\installableApps\sampleApp.ear
D:\WebSphere\AppServer\installedApps\sampleApp.ear\Increment.jar
D:\WebSphere\AppServer\installableApps\another.ear
D:\WebSphere\AppServer\installableApps\Samples.ear
D:\WebSphere\AppServer\installableApps\jmsample.ear

Figure 15-2 Opening an existing file

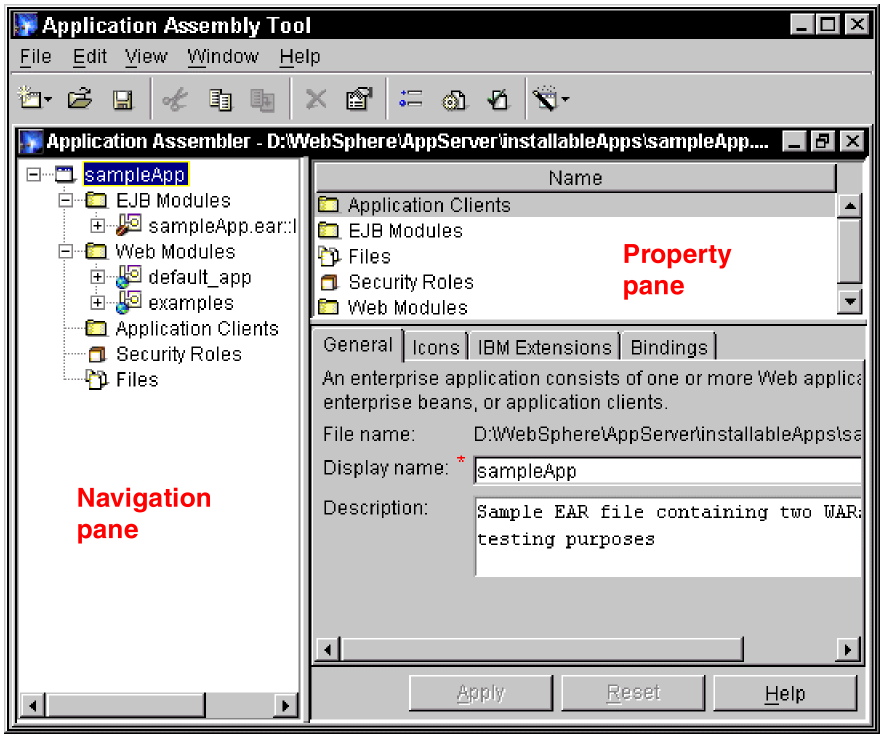
The interface for any module has two parts:

- Navigation pane
- Property pane

The navigation pane is a tree view of the module's contents, assembly properties, and files. It is used to select components or property groups and to review the content and structure of the module.

The property pane displays the properties for the element selected in the navigation pane. It is possible to hide this pane by selecting *View -> Show Property Pane.* Fields indicating required properties are signaled with a red asterisk.

The AAT user interface is shown in Figure 15-3.



An enterprise application consists of one or more Web applica
enterprise beans, or application clients.

File name: D:\WebSphere\AppServer\installableApps\sa

Display name: * sampleApp

Description: Sample EAR file containing two WAR:
testing purposes

Figure 15-3 AAT user interface

In addition to the menu bar, the toolbar provides access to the main functions related to module creation and administration (Figure 15-4).

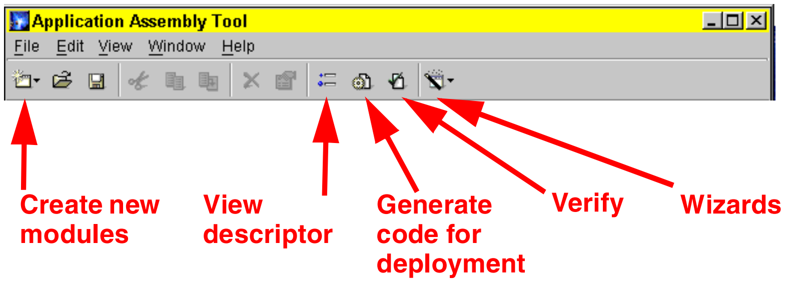

Application Assembly Tool
File Edit View Window Help
Create new modules
View descriptor
Generate code for deployment
Verify
Wizards

Figure 15-4 AAT toolbar

Creating a Web module

The Application Assembly Tool allows the user to create a J2EE Web module from pieces, that is, creating an empty module and incorporating the components, such as servlets, JSPs, or static content (HTML pages, images).

In Figure 15-5 we can see the structure of the navigation pane for a Web module.

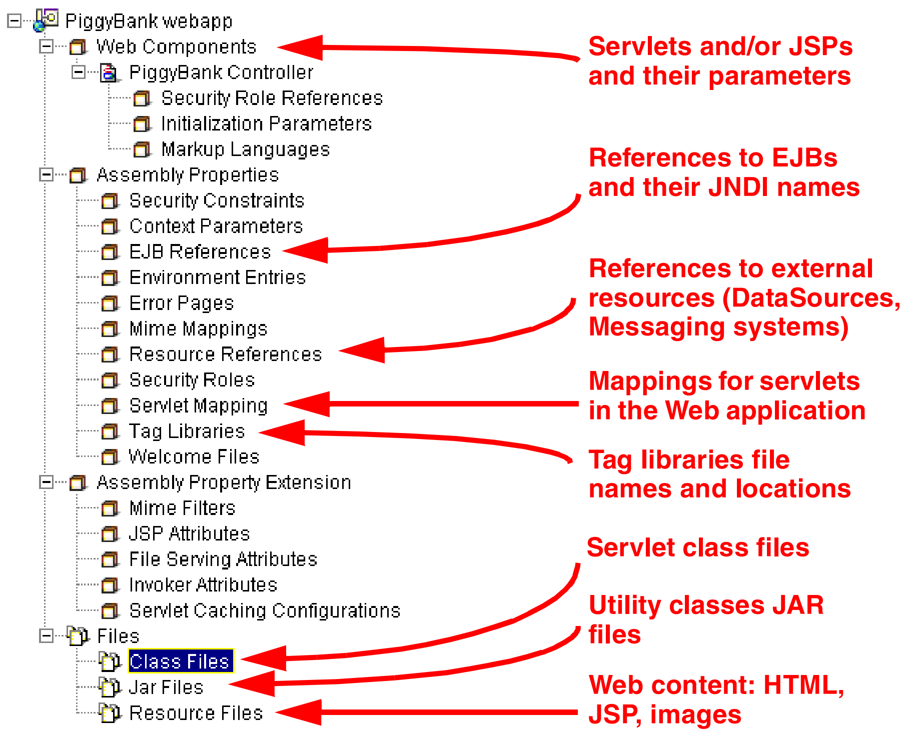


Figure 15-5 AAT navigation panel for a Web module

It is possible to drag-and-drop files from other Web modules (opening them in the AAT), so that both the configuration and the file are copied.

If we use the *Import* option, only the configuration is copied, and we have to add the file manually to the *Files* folder. The same applies to the *New* option, though in this case we have to type in the configuration for the new file.

When expanding the WAR file, the files are placed as shown in Figure 15-6.

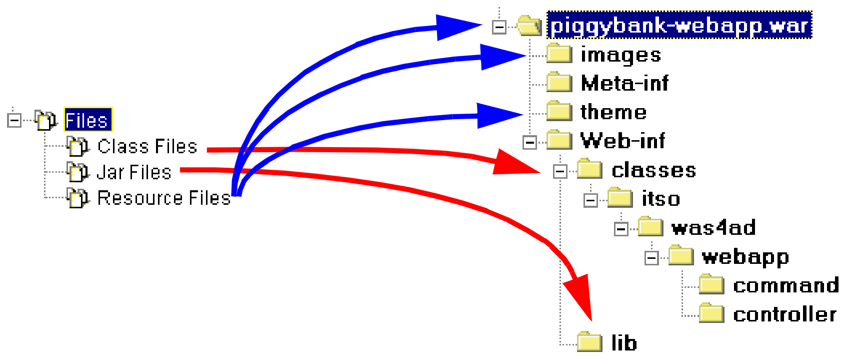


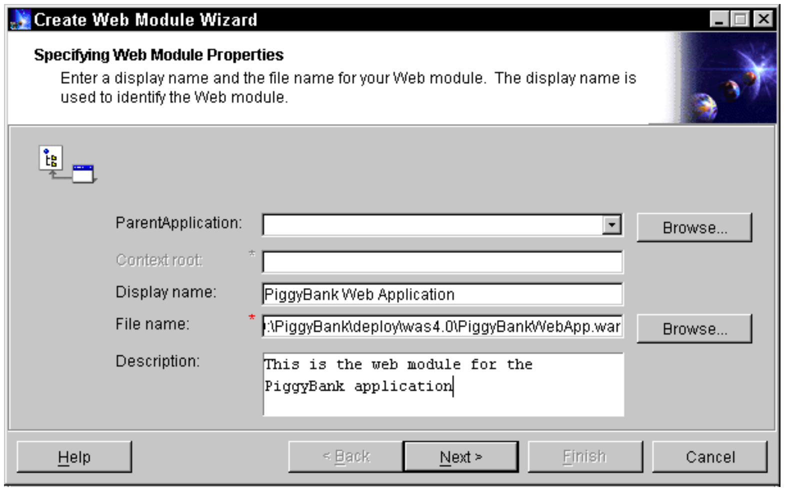
Figure 15-6   Files in the expanded Web application

Using the Web module creation wizard

We now describe the process of creating a new Web module for the PiggyBank application using the *Create Web Module Wizard*, as an example to illustrate the main configurable features of Web applications.

To launch the wizard, select *File -> Wizards -> Create Web Module Wizard*, or either click on the last button on the right (the *Wizards* button), and then select *Create Web Module Wizard.*

The first window lets the user specify the basic properties of the Web module (Figure 15-7).



Specifying Web Module Properties

Enter a display name and the file name for your Web module. The display name is used to identify the Web module.

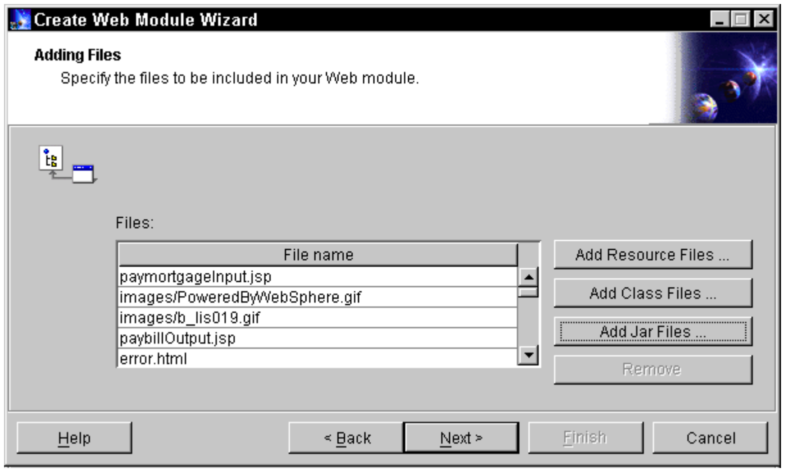
ParentApplication:	
Context root: *	
Display name:	PiggyBank Web Application
File name: *	:\PiggyBank\deploy\was4.0\PiggyBankWebApp.war
Description:	This is the web module for the PiggyBank application

Figure 15-7   Web module creation wizard

A description of the fields follows:

- *Parent application*—the enterprise application to which the module belongs (if any). For a stand-alone Web module, it is not necessary to fill in this field (or if we are creating the Web module previous to the creation of the enterprise application).
- *Context root*—only applicable when selecting a *parent application* (then, the context root is the root of the Web module inside the enterprise application)
- *Display name*—the Web module name (as it is displayed in the navigation panel).
- *File name*—the name of the WAR file (required)
- *Description*—description of the module

After completing this information, the next step is to add files to the module (Figure 15-8).


Create Web Module Wizard

Adding Files

Specify the files to be included in your Web module.

Files:

File name
paymortgageInput.jsp
images/PoweredByWebSphere.gif
images/b_lis019.gif
paybillOutput.jsp
error.html

Figure 15-8 Adding files to the Web module

As we have described before, each type of file available corresponds to the following:

- *Resource files*—JSPs, HTML pages, images, etc. (Web content)
- *Class files*—compiled classes that the application uses (servlets or other).
- *Jar files*—JAR files containing utility classes used by the Web components, for example, the Log4j classes of the PiggyBank application.

We can choose to add the common code as a JAR file or as individual classes. If our application only has a Web component (no EJBs or application client) and the utility classes are not many, we can choose this approach, but it is not recommended when the common code is shared across the application by all the component modules. In this case it is better to add the common JAR file to the enterprise application and indicate it in the class path of the individual modules.

For the PiggyBank application we have chosen a mixed configuration: we have packaged the shared common code in a JAR file, `piggybank-common.jar`, but in the Web module, `piggybank-webapp.war`, we have included individual class files for the command classes that the servlet uses, as they are not shared with the other component modules.

Thus, we have two utility classes JAR files for PiggyBank:

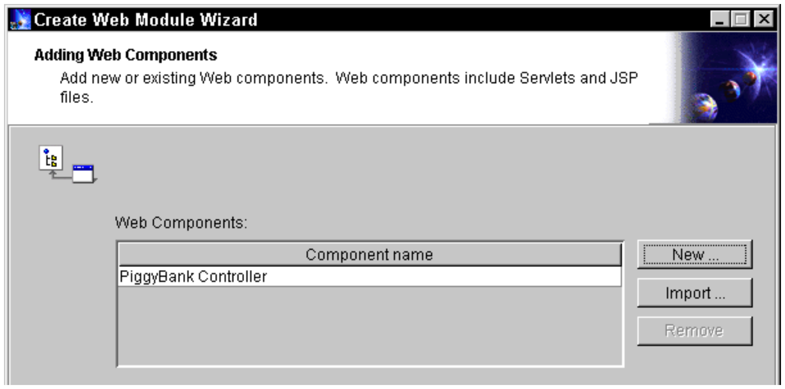
- `piggybank-common.jar`
- `Log4J.jar` (used for the logging part of the application)

To add the files, we open the folder that contains them in the browser window and add them.

To add the static resources files (JSPs, HTML, images), we select *Add Resource Files* and select `D:\PiggyBank\src\web`, the folder containing the static Web content.

Skipping through the *Icons* screen, the next step we have to care about is *Adding Web Components*. Here, we are able to register the servlets and JSPs in our Web module.

Select *New* to register new components or *Import* to get components from other Web modules or enterprise applications (Figure 15-9).



Adding Web Components

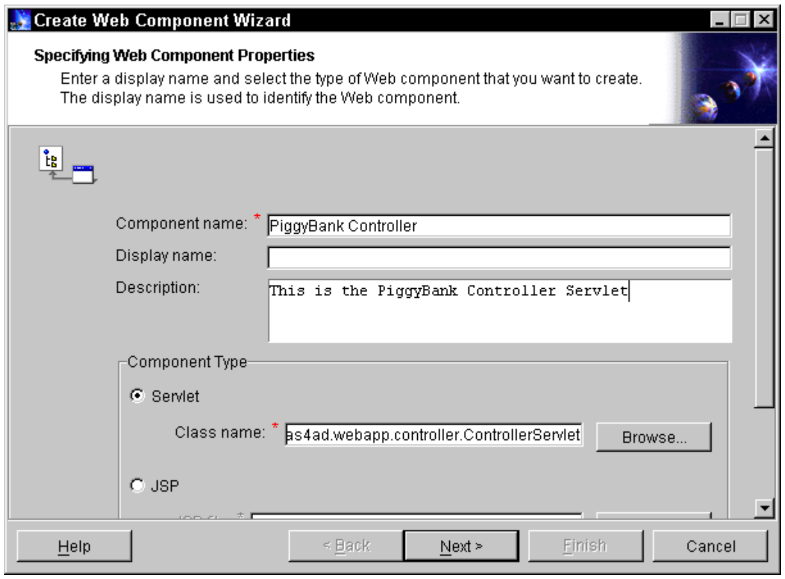
Add new or existing Web components. Web components include Servlets and JSP files.

Web Components:

Component name
PiggyBank Controller

Figure 15-9 Adding Web components

The following screen capture shows an example for the PiggyBank Controller servlet (Figure 15-10).



Specifying Web Component Properties

Enter a display name and select the type of Web component that you want to create. The display name is used to identify the Web component.

Component name: * PiggyBank Controller

Display name:

Description: This is the PiggyBank Controller Servlet

Component Type

[✓ Servlet]

Class name: * as4ad.webapp.controller.ControllerServlet

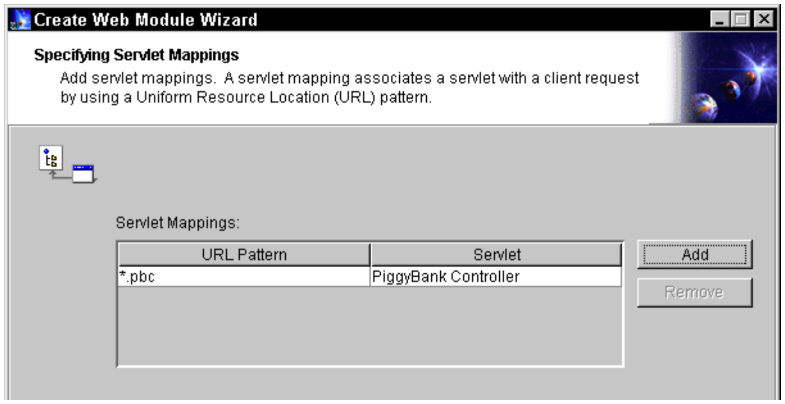
JSP

Figure 15-10 Adding a new servlet

For every component added, we have to type at least the information regarding the *Component name* (as it appears in the Web module navigation tree if no display name is introduced) and the appropriate file.

Also, it is possible to specify security roles and initialization parameters (servlets) for the component.

Skipping the security roles definition screen, we show next the *Specifying Servlet Mappings* screen. Here, for example, we add the mappings for the PiggyBank Controller (Figure 15-11).



Create Web Module Wizard

Specifying Servlet Mappings

Add servlet mappings. A servlet mapping associates a servlet with a client request by using a Uniform Resource Location (URL) pattern.

Servlet Mappings:

URL Pattern	Servlet
*.pbc	PiggyBank Controller

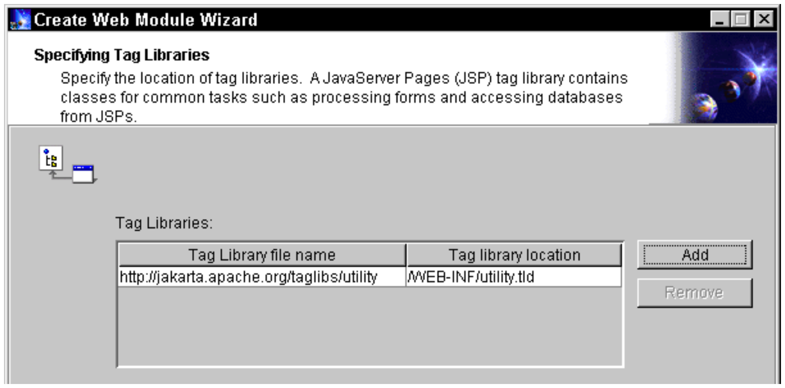
Figure 15-11 Adding servlet mappings

By clicking on *Add*, a new panel is displayed and we select the servlet name and type the URL mapping.

The next two panels allow the assembler to specify references to external resources (for example, databases or messaging systems) and context parameters for the servlets running in the Web application.

The next steps specify default *Error pages* and *MIME mappings*.

To configure Tag libraries in the next step we specify both the name of the file and its location within the WAR file (Figure 15-12).



Create Web Module Wizard

Specifying Tag Libraries

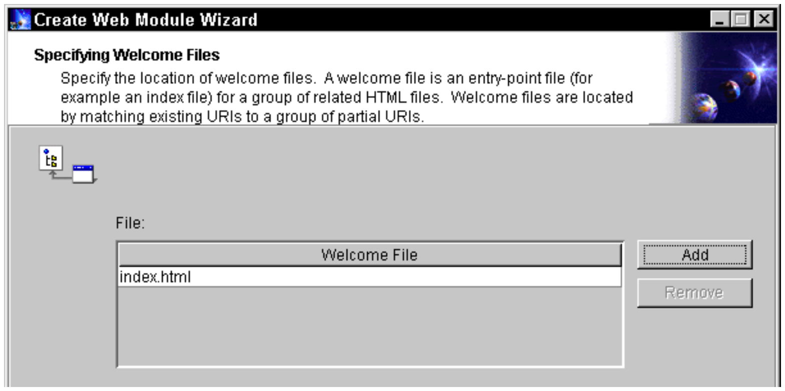
Specify the location of tag libraries. A JavaServer Pages (JSP) tag library contains classes for common tasks such as processing forms and accessing databases from JSPs.

Tag Libraries:

Tag Library file name	Tag library location
http://jakarta.apache.org/taglibs/utility	/WEB-INF/utility.tld

Figure 15-12 Specifying tag libraries for JSPs

In the next panel, we can specify the welcome file for our application, `index.html` (Figure 15-13).


Create Web Module Wizard

Specifying Welcome Files

Specify the location of welcome files. A welcome file is an entry-point file (for example an index file) for a group of related HTML files. Welcome files are located by matching existing URIs to a group of partial URIs.

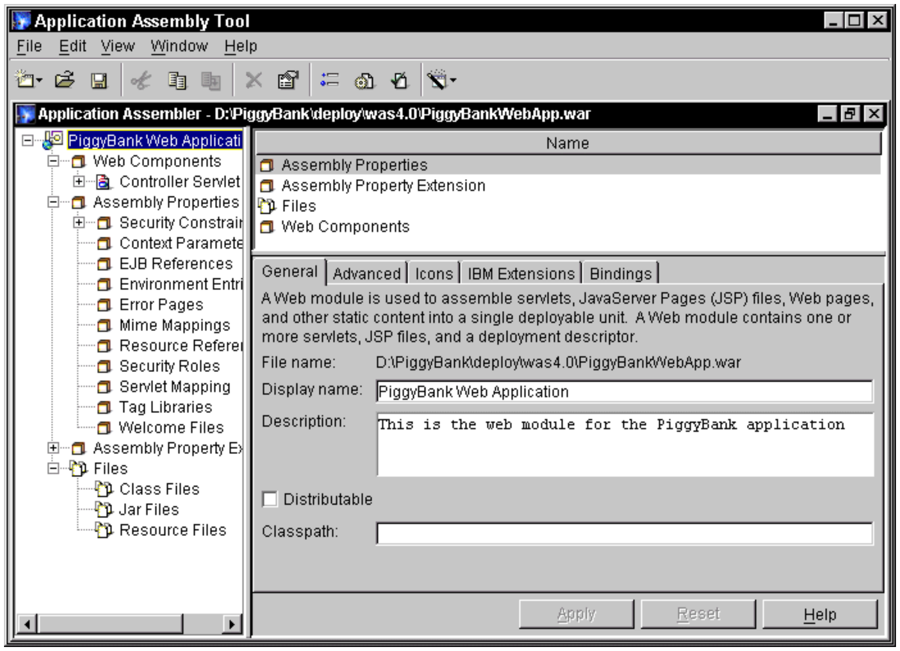
File:

Welcome File
index.html

Figure 15-13 Specifying welcome files

In the last panel of the wizard, we specify the EJB references used by elements of the Web module.

The Web module creation is finished when we click on the *Finish* button, and we can see the module structure in the standard AAT interface (Figure 15-14).



A Web module is used to assemble servlets, JavaServer Pages (JSP) files, Web pages, and other static content into a single deployable unit. A Web module contains one or more servlets, JSP files, and a deployment descriptor.

File name: D:\PiggyBank\deploy\was4.0\PiggyBankWebApp.war

Display name: PiggyBank Web Application

Description: This is the web module for the PiggyBank application

Distributable: [ ]

Classpath:

Figure 15-14 PIggyBank Web application module

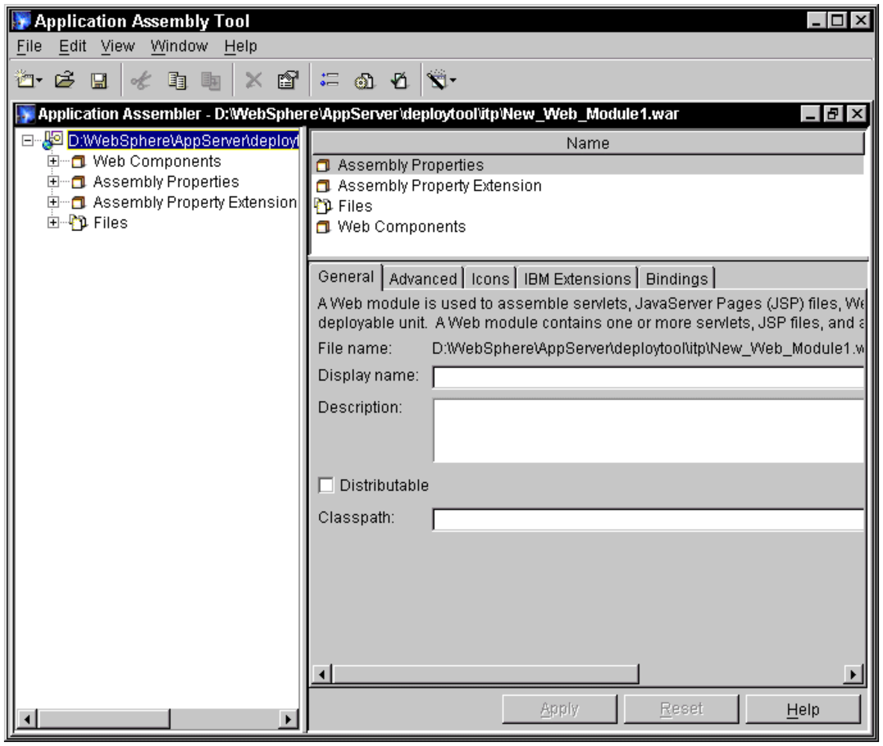
To save, select *File -> Save As* and navigate to the desired saving location.

Creating a Web module without the wizard

Wizards are useful to begin using the tool and become familiar with the options, as well as providers of a systematic module build process, but in general direct assembly of the modules is faster when we know which options we want to configure.

We describe now how to setup a Web module with its basic features, using again the PiggyBank application.

In the welcome screen (shown in Figure 15-1 on page 391), select Web Module or select *File -> New -> Web Module*. The skeleton of the module is shown in the navigation pane (Figure 15-15).


Application Assembly Tool
File Edit View Window Help
Application Assembler - D:\WebSphere\AppServer\deploytool\itp\New_Web_Module1.war
D:\WebSphere\AppServer\deploy
Web Components
Assembly Properties
Assembly Property Extension
Files
Name
Assembly Properties
Assembly Property Extension
Files
Web Components
General
Advanced
Icons
IBM Extensions
Bindings
Apply
Reset
Help

A Web module is used to assemble servlets, JavaServer Pages (JSP) files, We
deployable unit. A Web module contains one or more servlets, JSP files, and a

File name: D:\WebSphere\AppServer\deploytool\itp\New_Web_Module1.w

Display name:

Description:

☐ Distributable

Classpath:

Figure 15-15 Creating an empty Web module

It is possible to build WAR files from existing Web modules, by importing the required files to the new archive.

Consider, for example, that we have several modules with different parts of the Web application (let's say, for the case of an online banking application as the PiggyBank, account management, funds management, and so forth). At some moment we might want to test the interaction between these modules, so we have to assemble parts of them together (perhaps not all the files, but the ones that gives us key information about that interaction). We can take two approaches in this case:

- Group all the components we want to assemble from the repository (Java code in VisualAge for Java) or their storage locations and create the new Web module
- Create the Web module by importing the components directly from another pre-built modules.

Following with the second scenario, to import the Web components to the new module we have to:

- Add the files (class, JAR, etc.) from the existing module
- Import the configuration into the `Web Components` folder

Add files to the module by selecting *Files* and then *Class Files* in the navigation pane, and *Add Files* from the context menu. The process is the same that the performed using the wizard:

- Open the JAR (or ZIP) file where the compiled code for the servlet is
- Select the appropriate class file, navigating within the folders if necessary
- Click *Add* and then *OK*

The added files are placed under the `\WEB-INF\classes` directory of the web module.

The resource files (utility classes used by the other elements of the module), added as a JAR file, are placed under `\WEB-INF\lib`.

We add the JSP and static files (HTML, images) to the *Resource Files* section too. They are placed under the directory `\WEB-INF`.

Import the configuration details by selecting *Web Components* and *Import* from the context menu.

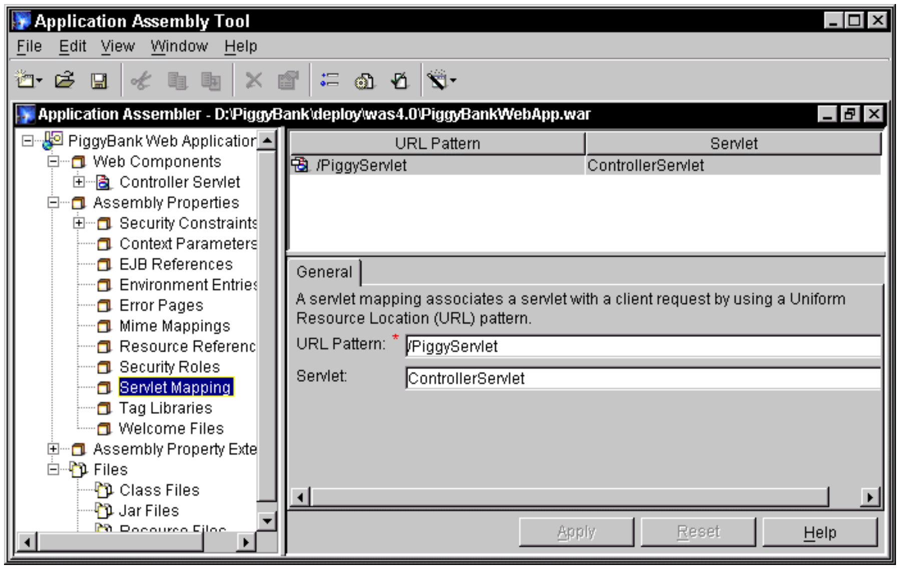
In the case of adding new files (not previously configured in other modules), we select the *New* option. Importing is useful when we have setup “complicated” configurations for the components (with long lists of initialization parameters that are tedious to rewrite), and it acts like a copy-paste mechanism between modules.

Importing is equivalent to dragging and dropping components from other modules (we can do this by opening the second module, and dragging the required component to the *Web Components* folder; the properties and the files are copied automatically).

Again, the screens shown are the same as the screens displayed for the wizard section (see Figure 15-10 on page 398). The JSP files are added in the same way.

After having all the required files added to the module, the next step is to create a mapping for the servlets: in the navigation pane, select *Assembly Properties -> Servlet Mapping -> (Right button)* and select *New*. A window is displayed where we must enter the URL pattern name and the servlet associated to it (see Figure 15-11 on page 399)

After clicking *OK*, the new servlet mapping is displayed in the *Servlet Mappings* list (Figure 15-16).



URL Pattern	Servlet
/PiggyServlet	ControllerServlet

A servlet mapping associates a servlet with a client request by using a Uniform Resource Location (URL) pattern.

URL Pattern: * /PiggyServlet

Servlet: ControllerServlet

Figure 15-16 Servlet mappings list

For the PiggyBank application, the addition of the other features (tag libraries, error and welcome pages) is done through the node in the navigation pane, in a similar way as how it is done by the wizard.

Creating an EJB module

WebSphere 4.0 supports only the EJB 1.1 specification, so that if our EJBs have been created with the 1.0 specification, we have to convert them to 1.1.

With VisualAge for Java 4.0 we can export the EJBs to a 1.1 undeployed JAR file (see Chapter 11, “Development using VisualAge for Java” on page 259), and then generate the deployment code either with the AAT or with the command line EJB deployment tool (see “EJB deployment tool” on page 418), but it may be necessary to fix some details of the code (the aspects that have to do with the differences between the 1.0 and 1.1 specifications).

The structure of the navigation tree for an EJB module is shown in Figure 15-17.

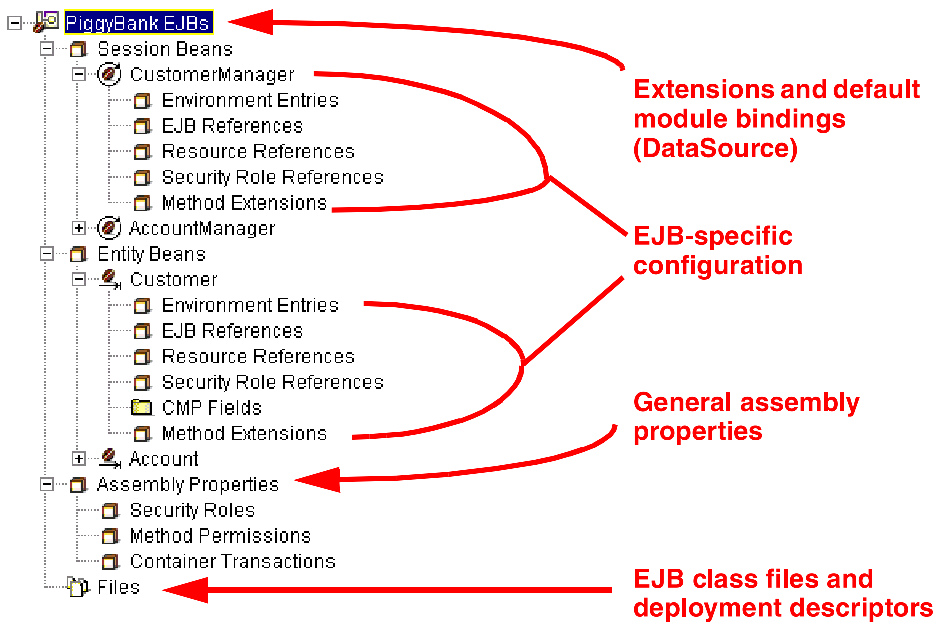


Figure 15-17 AAT navigation panel for an EJB module

Setting up the configuration data

Here we describe how the deployment descriptor information matches the AAT property panels and how to configure EJB data using only the AAT.

Figure 15-18 shows schematically how the deployment descriptor information fits in the AAT tree structure.

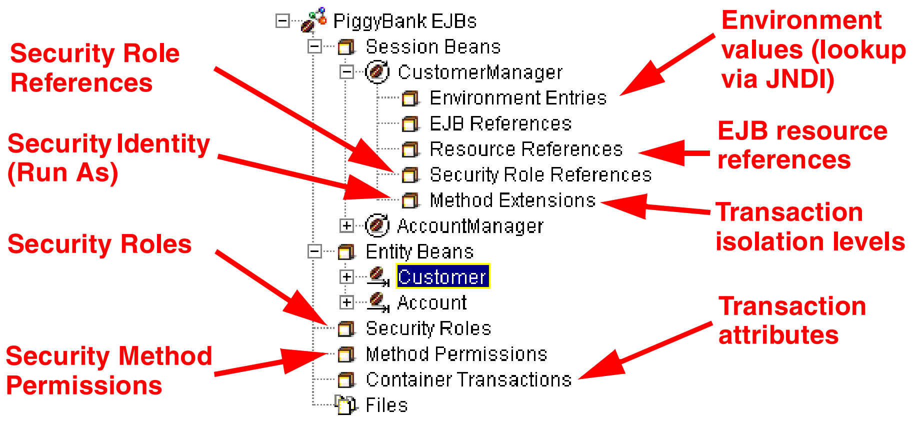
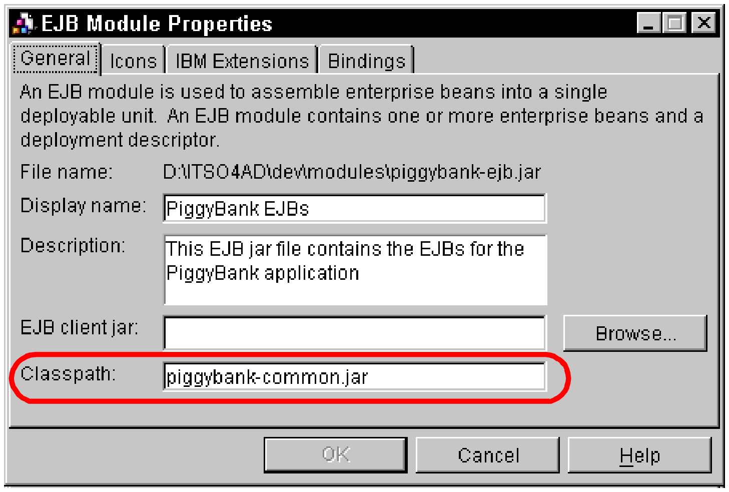


Figure 15-18 Deployment descriptor data in the AAT

Security roles are also configurable through the Security Roles panel in the enterprise application (EAR) file.

Class path information added through the module main property panel is written to the `manifest.mf` file. Class path entries must be separated by spaces (Figure 15-19).

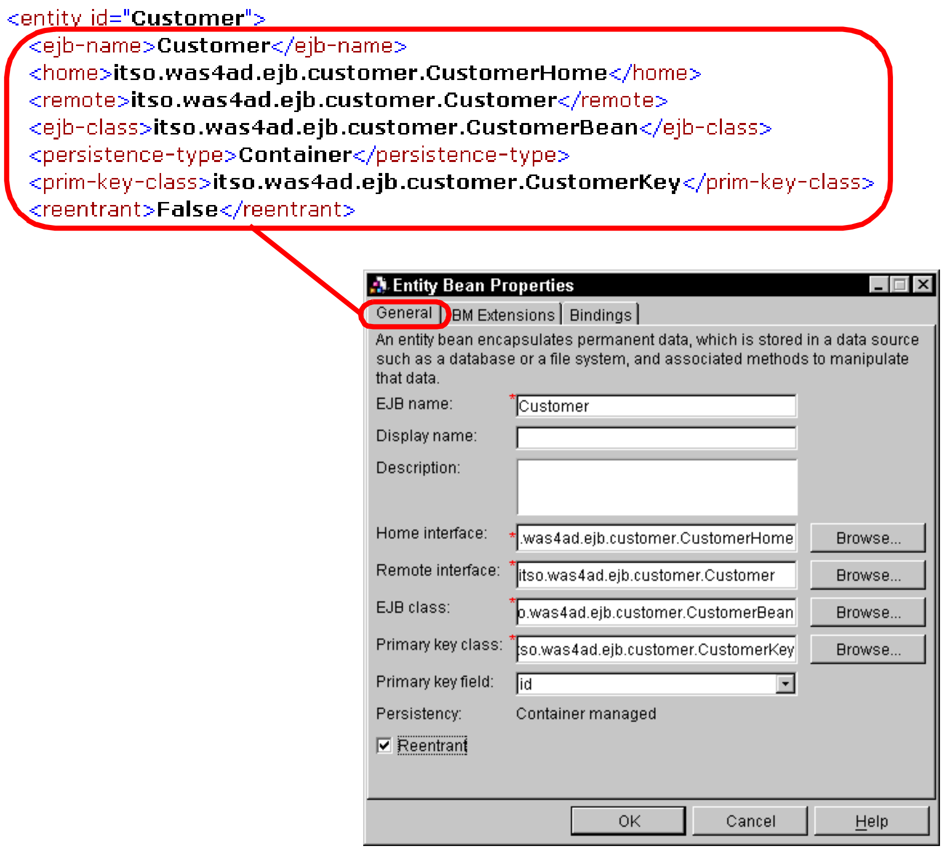


An EJB module is used to assemble enterprise beans into a single deployable unit. An EJB module contains one or more enterprise beans and a deployment descriptor.

File name:	D:\ITSO4AD\dev\modules\piggybank-ejb.jar
Display name:	PiggyBank EJBs
Description:	This EJB jar file contains the EJBs for the PiggyBank application
EJB client jar:	
Classpath:	piggybank-common.jar

Figure 15-19 Setting up the class path for the module

Basic EJB properties, such as the home and remote interface names are configured through the General panel of each bean (Figure 15-20).


```
<entity id="Customer">
  <ejb-name>Customer</ejb-name>
  <home>itso.was4ad.ejb.customer.CustomerHome</home>
  <remote>itso.was4ad.ejb.customer.Customer</remote>
  <ejb-class>itso.was4ad.ejb.customer.CustomerBean</ejb-class>
  <persistence-type>Container</persistence-type>
  <prim-key-class>itso.was4ad.ejb.customer.CustomerKey</prim-key-class>
  <reentrant>False</reentrant>
```

Entity Bean Properties

General | IBM Extensions | Bindings

An entity bean encapsulates permanent data, which is stored in a data source such as a database or a file system, and associated methods to manipulate that data.

EJB name: *	Customer
Display name:	
Description:	
Home interface: *	.was4ad.ejb.customer.CustomerHome
Remote interface: *	itso.was4ad.ejb.customer.Customer
EJB class: *	o.was4ad.ejb.customer.CustomerBean
Primary key class: *	tso.was4ad.ejb.customer.CustomerKey
Primary key field:	id
Persistency:	Container managed

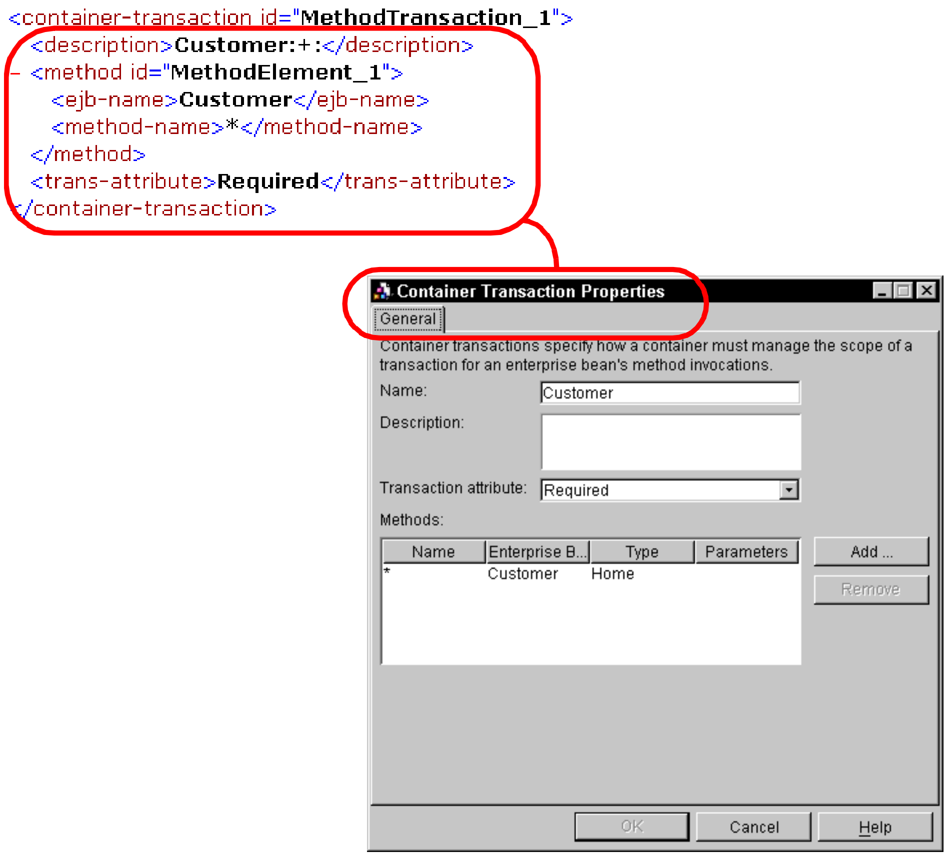
☑ Reentrant

Figure 15-20 Setting up general EJB configuration data

The *Bindings* pane is the place to detail the JNDI name of the bean, as well as the default DataSource to be used in case of an Entity bean.

CMP for entity EJBs can be setup in the *CMP Fields* pane under each EJB.

Transaction data (such as isolation levels) has to be setup under the *Container Transactions* node (Figure 15-21).



```
<container-transaction id="MethodTransaction_1">
  <description>Customer:+:</description>
- <method id="MethodElement_1">
    <ejb-name>Customer</ejb-name>
    <method-name>*</method-name>
  </method>
  <trans-attribute>Required</trans-attribute>
</container-transaction>
```

Container Transaction Properties

General

Container transactions specify how a container must manage the scope of a transaction for an enterprise bean's method invocations.

Name: Customer

Description:

Transaction attribute: Required

Methods:

Name	Enterprise B...	Type	Parameters
*	Customer	Home	

Figure 15-21 Setting up container transaction properties

Working with EJB 1.1 JAR files

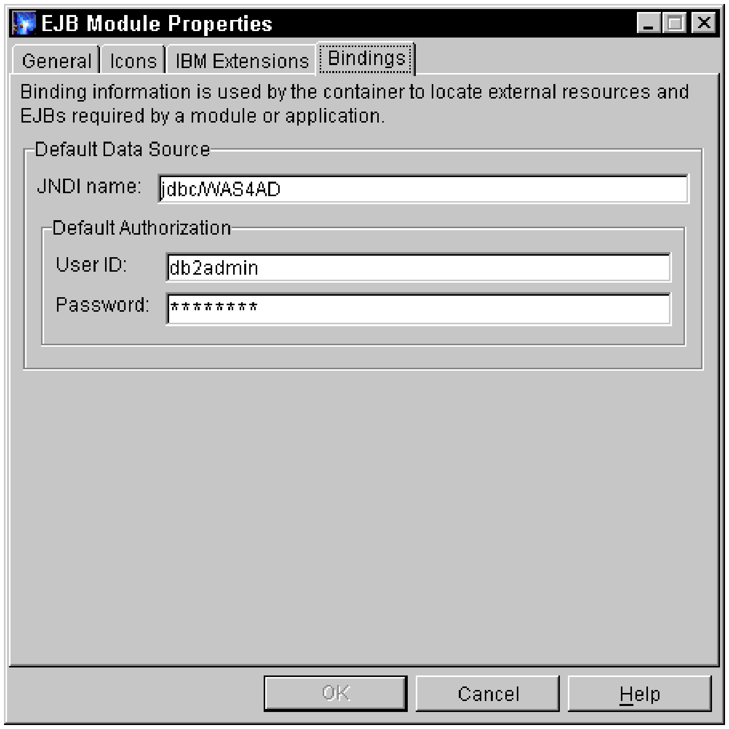
When using 1.1 EJBs, we only have to care about generating the deployed code (if necessary), and specifying the deployment descriptor information.

Let’s pick up the example of PiggyBank again: we describe how to generate deployment code for the EJBs created using the Java 2 SDK (see Chapter 9, “Development using the Java 2 Software Development Kit” on page 183).

The property pane shows the deployment descriptor information that we configured previously.

In the PiggyBank example, we have added the common code JAR file to the EAR archive (as we are packaging all the modules together in a single enterprise application file, we do not want to include the common JAR file in each single module, but only at the top level, and reference it through the class path inside the modules). For the EJBs to find the utility classes, we specify the JAR file name in the *Classpath* entry (see Figure 15-19 on page 405).

The *Bindings* tab lets us specify the default DataSource for the module, but if we specify a DataSource for each bean, the default configuration is overridden (Figure 15-22).



Binding information is used by the container to locate external resources and EJBs required by a module or application.

Default Data Source

JNDI name: jdbc/WAS4AD

Default Authorization

User ID: db2admin

Password: ********

Figure 15-22 Specifying binding information for the EJB module

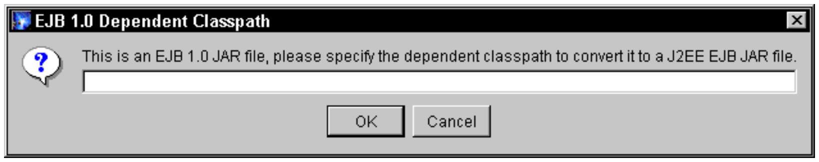
We generate the deployed code once we have setup the deployment descriptor information.

Working with EJB 1.0 JAR files

If we have developed 1.0 EJBs, we adapt them to the 1.1 specification prior to deployment (we fix the code differences and recompile the bean classes).

Let's pick up the EJB 1.0 undeployed file from the PiggyBank application, `PiggyBankEJBs10.jar`, developed in VisualAge for Java, and open it in the AAT.

We are prompted to specify the *dependent classpath* (used to specify classes that are not included in the EJB JAR file but are needed to inspect the EJB classes), as shown in Figure 15-23.



This is an EJB 1.0 JAR file, please specify the dependent classpath to convert it to a J2EE EJB JAR file.

Figure 15-23 EJB 1.0 dependent class path

We click *OK* on this window (our EJBs do not require dependent class path specification), and the EJB file is opened in the AAT interface.

At this point, we can edit the deployment descriptor to configure the module properties. By selecting the JAR file in the navigation pane, we can see its properties in the property pane (setting this pane to be visible as described in "Using the interface" on page 391), and we can also specify properties for every EJB in the JAR file.

Let's review the configuration options with more detail:

- *Environment entries*—defined to customize the business logic of the application at runtime, so that no modifications of the source code are necessary. These variables are available through an JNDI name context.

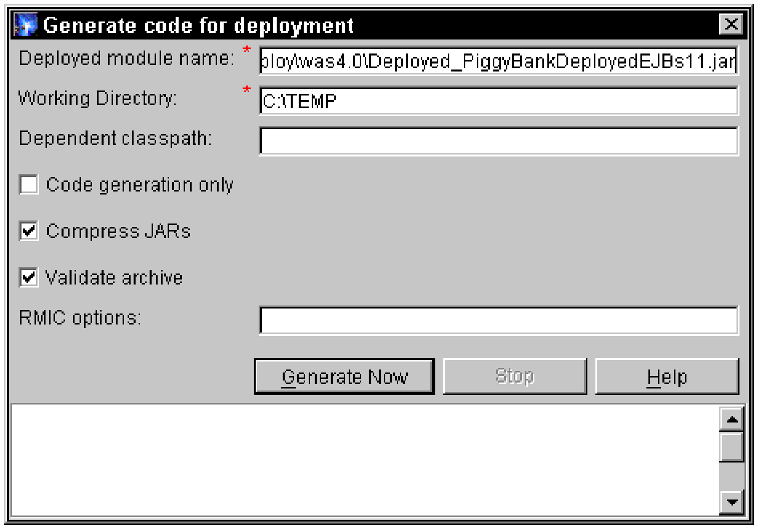
  For example, we can setup an environment variable that defines the activation or not of the tracing facility implemented in our application.
- *EJB references*—used to reference an EJB's home interface that the component uses (this option is general for all modules, as EJBs can be referenced from the Web, client or EJB tier).
- *Resource references*—used to specify logical names that define a connection to an external resource (database, messaging system, and so forth).
- *Security role references*—linking to security roles used in the code (for example, an EJB's method can be performed based on the user's role).
- *Method extensions*—security and transaction information for EJB methods, (read-only methods, isolation levels, and so forth).

If we save the file now, WebSphere generates the deployment descriptor (`ejb-jar.xml`) and the extension files (`ibm-ejb-jar-bnd.xmi` and `ibm-ejb-jar-ext.xmi` and the database files for entity beans). The next task to be performed is the generation of the deployed code.

Generating deployed code

AAT allows us to generate deployed code for an EJB module, or all of the EJB modules in an EAR file, by providing a GUI interface to the `ejbdeploy` tool. ejbdeploy is discussed in "EJB deployment tool" on page 418.

To perform this task in the Application Assembly Tool, select *File -> Generate Code for Deployment* (Figure 15-24).



Deployed module name: * ploy\was4.0\Deployed_PiggyBankDeployedEJBs11.jar
Working Directory: * C:\TEMP
Dependent classpath:
☐ Code generation only
☑ Compress JARs
☑ Validate archive
RMIC options:

Figure 15-24 Generating deployment code for EJB 1.1 JAR files

A description of the fields and available options follows:

- *Deployed module name*—the name of the deployed mode resulting from the code generation process
- *Working directory*—folder to store temporary files created during the process
- *Dependent classpath*—when the EJBs require classes that are not present in the JAR file
- *Code generation only*—select this to generate the Java files only, without running the RMI compiler `rmic` and `javac`
- *Compress JARs*—check this box to compress the contents of the generated JAR

- *Validate archive*—checks that the archive is complete and that the deployment descriptor properties and references contain the appropriate values:
  - All classes referenced in the deployment descriptor must exist in the JAR
  - Method signatures must be compliant with the EJB 1.1 specification

Errors appearing during the code generation process are shown in the text area. If any errors or exceptions appear while executing RMIC, the deployment process is aborted and the deployed code is not generated.

We have to generate the deployment code only when the EJBs have been fixed to be 1.1 compliant.

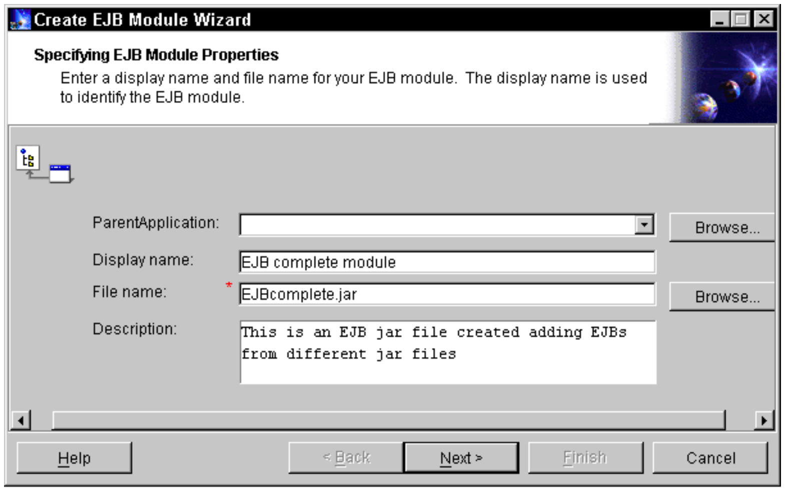
Once the code generation process is completed, we have an EJB 1.1 JAR file prepared to be added to an enterprise application.

It is also possible to deploy EJBs using the command line tool *EJBDeploy* directly (see “EJB deployment tool” on page 418).

Using the EJB module creation wizard

We now review briefly the use of the EJB module creation wizard. This feature is useful when first using the tool, to familiarize oneself with the options and structure of the configuration.

Figure 15-25 shows the *Create EJB Module Wizard*.


Create EJB Module Wizard

Specifying EJB Module Properties

Enter a display name and file name for your EJB module. The display name is used to identify the EJB module.

ParentApplication:

Display name: EJB complete module

File name: * EJBcomplete.jar

Description: This is an EJB jar file created adding EJBs from different jar files

Figure 15-25 EJB module creation wizard

As in the case of the Web module wizard, we have to type at least the required field, *File name* (the name of the EJB module file). We select a parent application, if we are creating this EJB module inside an enterprise application, otherwise we leave these fields blank.

In the *Adding Files* screen, we select utility files used by the EJBs (libraries). Next (skipping the icons section), we add the EJBs to the module (*Adding Enterprise Beans*). Click on *New* to add new beans to the module or on *Import* to import existing EJBs from another module.

If, for example, we are combining EJBs from two modules into one, we can just drag-and-drop the beans from the modules, so that both the configuration and the files are copied, and the process is easier. However, it is not possible to do this from the wizard screen.

After finishing, we can modify any of the properties of the JAR file (that are written to the deployment descriptor), by selecting them from the navigation pane and editing them on the property pane.

Creating an application client module

A difference between a standard Java program and an application client following the J2EE model is that the client accesses the resources (EJBs, DataSources, messaging systems, and so forth) through JNDI lookup (so no “physical” resource data is hard coded in the program). Storing the actual resource data outside the client program provides flexibility and reusability. This configuration data is set up using two tools in WebSphere environment:

- *Application Assembly Tool*—to build the client JAR file (or the client EAR file if we are assembling the client code separately to distribute it across the client machines) and configure the EJB references, environment entries and other non-client specific resources.
- *Application client resource configuration tool*—to set up the client-specific resources (databases, messaging systems). We talk about this tool in Chapter 16, “Deploying to the test environment” on page 431, in “Application client resource configuration tool” on page 458.

All these references are resolved by the application server at runtime.

Assembling the module

The structure of an application client module in the AAT is shown in Figure 15-26.

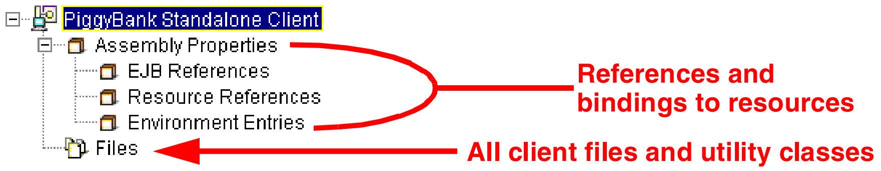

PiggyBank Standalone Client
Assembly Properties
EJB References
Resource References
Environment Entries
Files
References and bindings to resources
All client files and utility classes

Figure 15-26 Application client navigation tree

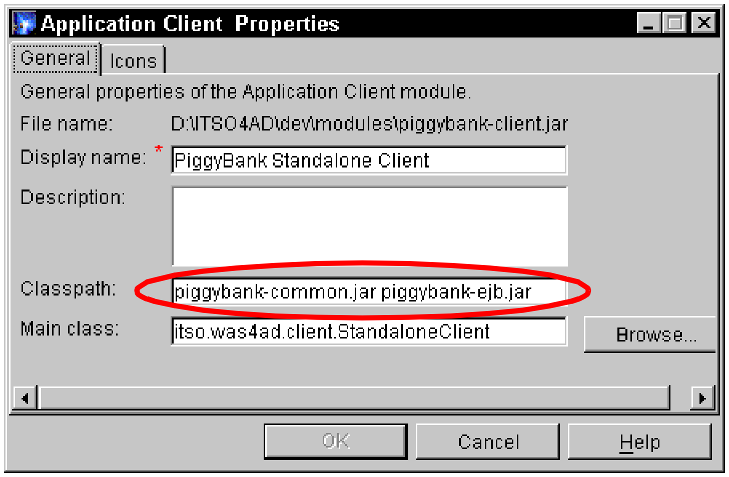
When we create a new application client in the AAT, several steps have to be followed to set up the module (Figure 15-27):

- Add the client classes to *Files*: for PiggyBank, we have only one class, `itso.was4ad.client.StandaloneClient` for the command line client.

  For the Swing client, we add all the classes in the `itso.was4ad.client.swing` package.

- Specify the executable class (the one that contains the `main` method). This class appears in the property pane in the *Main Class* field.

- Set up the class path: we add any JAR files that contain classes used by the client.



General properties of the Application Client module.

File name: D:\ITSO4AD\dev\modules\piggybank-client.jar

Display name: * PiggyBank Standalone Client

Description:

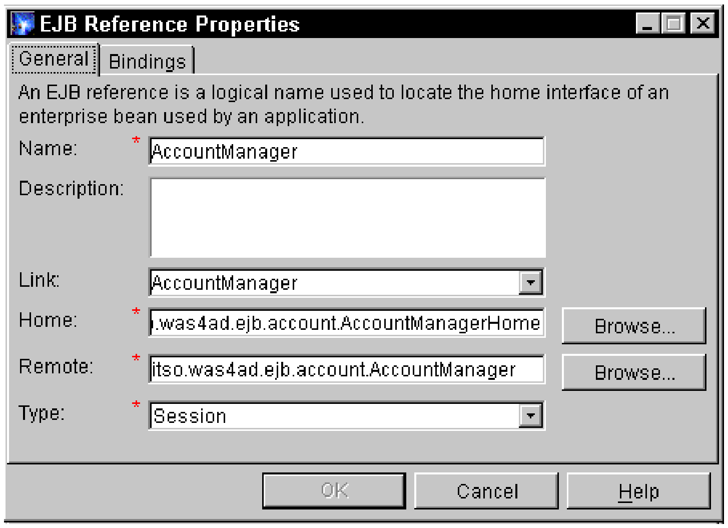
Classpath: piggybank-common.jar piggybank-ejb.jar

Main class: itso.was4ad.client.StandaloneClient

Figure 15-27 Setting up the application client class path

For the PiggyBank, as we are assembling all the modules in a single EAR file. We do not add the common code and EJB client JAR files to the client application module, but we include references to them in the class path.

- Set up the references to resources (EJBs and DataSources) and environment entries (Figure 15-28):


EJB Reference Properties

General | Bindings

An EJB reference is a logical name used to locate the home interface of an enterprise bean used by an application.

Name: * AccountManager

Description:

Link: AccountManager

Home: * .was4ad.ejb.account.AccountManagerHome

Remote: * itso.was4ad.ejb.account.AccountManager

Type: * Session

Figure 15-28 Setting up EJB references in the client module

  – The PiggyBank standalone client references two EJBs: `AccountManager` and `CustomerManager`. We include both under the *EJB References* node.
  – In the *General* tab we enter the name of the reference, as well as the type of the bean (session or entity) and the home and remote interfaces. We use se the *Browse* button to open the JAR file containing the classes and select them (they are not included in our application client JAR file).
  – In the *Binding*s tab, we can enter the global JNDI name of the EJB:

    ```
    itso/was4ad/ejb/account/AccountManagerHome
    ```

Note: For more information about using JNDI see “Using JNDI” on page 326.

We setup resource references under the *Resource References* node. These references have to be configured also in the application client resource configuration tool (see “Application client resource configuration tool” on page 458), and several details have to be taken in account:

- The *JNDI name* specified in the *Bindings* tab has to match the JNDI name specified in the application client resource configuration tool (as they refer to the same resource).
- The *Name* field in the *General* tab and the *JNDI name* in the *Bindings* tab have to be the same.

The *Name* field in the *General* tab is used to bind a reference to the object in the JNDI namespace and to retrieve client specific resource information configured in the application client resource configuration tool.

Application client creation wizard

The Application Assembly Tool provides a wizard to guide us in the process of creating an application client JAR file. Essentially, the steps of the wizard are the ones we have just described:

- Introduce the file name and display name for the module
- Add class and/or resource files
- Select the executable class and set up the class path
- Add icons for the module
- Configure EJB references
- Configure external resources references
- Configure environment entries

Assembling the complete application: the EAR file

Once we have built all the J2EE modules, we can assemble the PiggyBank enterprise application in an EAR file. The structure of an EAR file in the AAT is shown in Figure 15-29.

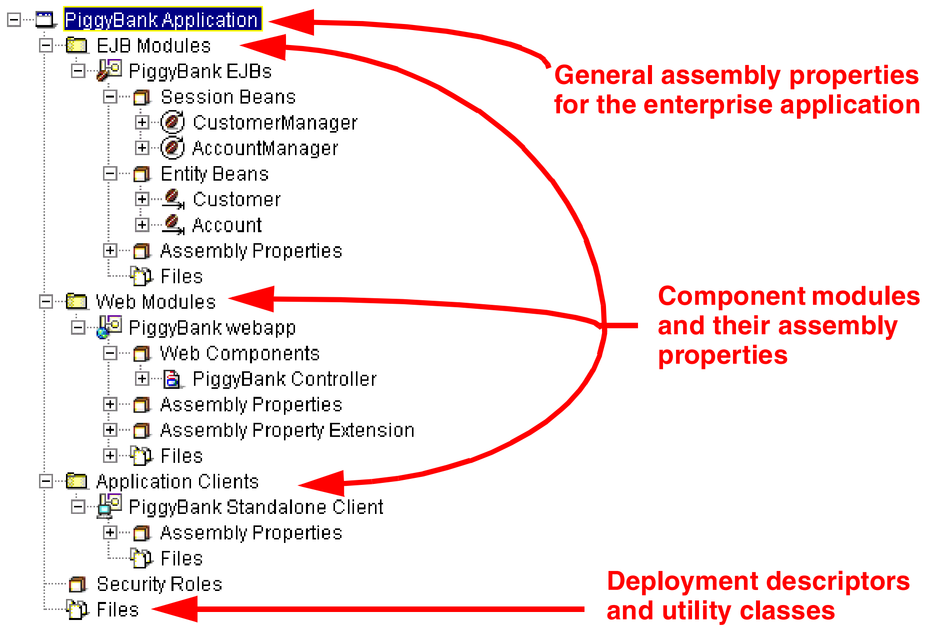


Figure 15-29 Enterprise application navigation tree

Assembling options for enterprise applications

In the PiggyBank example, we have created a single EAR file containing all the application code. However, in a real scenario, we would package the client code apart from the Web application (so that it can be distributed and installed across the client machines). Then we would build two EAR files containing the following:

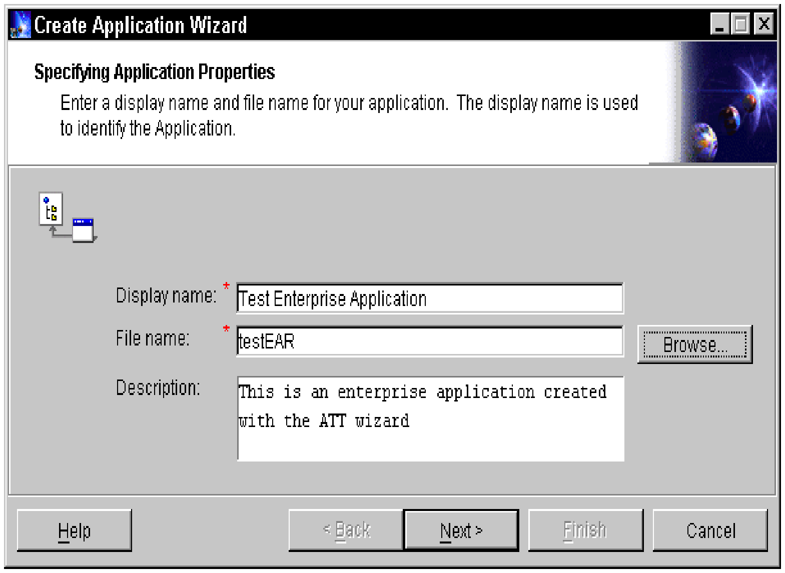
- Web application, EJBs and common code
- Client application, EJB client files and common code

Other configurations might be appropriate depending on the application structure, for example, assembling the EJBs in a separate file in case they are shared across several Web applications. Then, each Web application should include references to the EJBs as external resources.

Using the enterprise application wizard

Let's use the *Create Application Wizard* for this task.

In the first panel (Figure 15-30) we have to enter the application name and file name (in a similar way as to the other modules).



Specifying Application Properties

Enter a display name and file name for your application. The display name is used to identify the Application.

Display name: * Test Enterprise Application

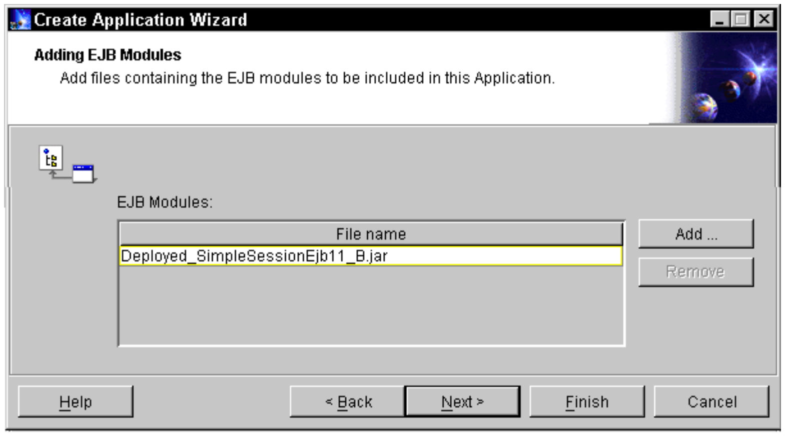
File name: * testEAR

Description: This is an enterprise application created with the ATT wizard

Figure 15-30 Create application wizard

The next step is to add supplementary files that the enterprise application uses. These can be icon libraries or other utilities. The icons used to represent the file are selected in the next panel.

Then we come to the panel where we must add the EJB modules that our enterprise application uses. We add here the 1.1 JAR files created in the section: “Creating an EJB module” on page 404. After clicking on *Add*, navigate to the directory that contains the JAR file and add it to the module (Figure 15-31).


Create Application Wizard
Adding EJB Modules
Add files containing the EJB modules to be included in this Application.
EJB Modules:
File name
Add ...
Remove
Help
< Back
Next >
Finish
Cancel

Deployed_SimpleSessionEjb11_B.jar

Figure 15-31 Adding EJB modules to the enterprise application

The Web modules and application clients are added in the next two steps in a similar way (we add the modules that we have created in previous sections).

The last step is the specification of security roles for the whole enterprise application.

After completing the wizard tasks, and prior to the installation in the application server, we have to configure some details such as the binding information (resolving the JNDI names for the EJBs).

In case we have not generated the deployed code for our EJBs, we do this prior to the installation of the enterprise application in WAS (though the application server provides the option of deploying the code when installing).

Now, the PiggyBank application is ready to be installed in WAS 4.0.

We can also assemble the EAR file by dragging and dropping the previously created modules (so that we copy both the configuration and the files) from another windows of the AAT, speeding up the creation process.

EJB deployment tool

In this section we describe the command line EJB deployment tool, `ejbdeploy`. We discuss the functions performed by the tool, and explain when and how you might want to use it.

Important: The original release of WebSphere AEs included an early version of the `ejbdeploy` tool that does not provide all of the functionality described here. The version of the tool we describe was made available for download from the VisualAge Developer Domain Web site at the same time that VisualAge for Java Version 4.0 became generally available. This newer version of the tool includes support for enhanced CMP EJB mapping. The updated tool will be included in the first release of WebSphere AE, and updated in a PTF (fixpack) for WebSphere AEs.

What does the EJB deployment tool do?

Some readers may find the name of the tool a little confusing—although the tool is named `ejbdeploy` it does not actually deploy EJBs into the application server, that is to say it does not *install* them into an instance of the application server. It does however perform a task that is essential to deployment—it generates *deployed code* for our EJBs.

The deployed code is the WebSphere-specific code that hooks your J2EE-compliant EJB into the WebSphere container—if for some reason we wished to deploy to another vendor's EJB container there would be an equivalent process that would generate code specific to that vendor's container.

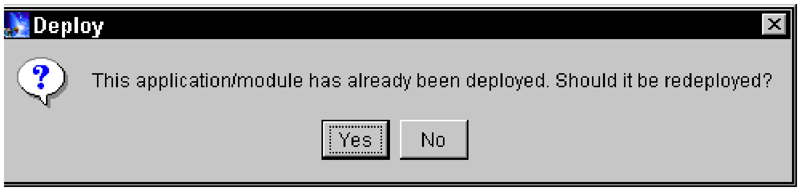
The deployed code includes the Remote Method Invocation (RMI) stub code generated by the `rmic` compiler, as well as persistence mapping code for container managed persistent (CMP) EJBs. We describe the generation of persistence mapping code in more detail shortly.

When is the EJB deployment tool executed?

The WebSphere deployment tools invoke `ejbdeploy` automatically when you install an EJB module (either standalone or as part of an EAR file) under the following circumstances:

- You install any EJB module into WebSphere AEs using `SEAppInstall` (see "SEAppInstall command line tool" on page 433) and you do not specify the command option `-ejbdeploy false`.

- You use the AE administrator's console to install an EJB module that has not already been processed by the `ejbdeploy` tool into WebSphere (see "Installing new applications" on page 450).
- You use the AE administrator's console to install an EJB module that has already been processed by the `ejbdeploy` tool into WebSphere and you click *Yes* when prompted with the dialog in Figure 15-32.


Deploy

This application/module has already been deployed. Should it be redeployed?

Figure 15-32 AE administrator's console redeployment dialog

The AAT also invokes `ejbdeploy` when you choose the *File -> Generate Code for Deployment* menu option, as described in "Generating deployed code" on page 410.

Why would I want to run the EJB deployment tool myself?

We envisage several scenarios where you must run `ejbdeploy` directly, either from the command line, from AAT, or from a script such as a build script:

- You want to generate default CMP persistence mapping files so that you can customize them to provide meet-in-the-middle mapping (see ""Customizing CMP persistence mapping" on page 420").
- You are using WebSphere AEs with the original version of the `ejbdeploy` tool and you want to install an EJB JAR with customized CMP mapping using the standalone EJB deployment tool downloaded from VisualAge Developer Domain.
- You are creating an EJB JAR that will be installed into WebSphere many times—for example to support individual developer environments—and you want to incur the cost of generating deployed code once only (see "Packaging EJBs and generating deployed code" on page 217).
- You want to create and extract XML EJB 1.1 deployment descriptors from an EJB 1.0 JAR file which contains serialized deployment descriptors (see "Migrating and validating EJB JAR files" on page 426).
- You want to test a newly-created EJB JAR file for EJB 1.1 compliance.

Under other circumstances, while there is no harm caused by generating the deployed code directly, there is no real benefit in not allowing WebSphere to execute the tool for you.

Customizing CMP persistence mapping

When the `ejbdeploy` tool is used against a JAR file that contains CMP entity EJBs, it looks for two meta-data files in the EJB JAR file that tell it how to generate the code that persists the container managed fields in the EJB to the target database. These two files are stored in the JAR file at the following locations:

`META-INF/Schema/Schema.dbxmi`	Contains descriptions of the tables in the database and defines their columns
`META-INF/Map.mapxmi`	Connects the object model description with the specified database schema

These files are created by VisualAge for Java Version 4.0 when we use the *Export -> EJB 1.1 JAR* menu option to create an EJB JAR file. VisualAge for Java creates these files using the database schema and mapping information stored in the repository and managed using the schema and map browser tools that enable us to perform meet-in-the-middle mapping between CMP EJBs and the database tables.

We anticipate that future, yet-to-be-released WebSphere EJB development tools will also offer the ability to create and maintain these files.

If the schema and map files do not exist in the EJB JAR file, `ejbdeploy` creates a new schema and map based upon its default mapping rules. We can see this if we run `ejbdeploy` against a version of our PiggyBank EJB JAR that does not contain a schema or map (Figure 15-33).

```
D:\DeployTool>ejbdeploy d:\itso4ad\dev\modules\piggybank-ejb.jar d:\temp
                        d:\temp\deployed-piggybank-ejb.jar
Starting workbench.
Creating the project.
Importing file..
Starting Validation.
Creating Top-Down Map...
Generating deployment code
Building: /deployed-piggybank-ejb.jar.
Invoking RMIC.
Building: /deployed-piggybank-ejb.jar.
Generating DDL
Shutting down workbench.
EJBDeploy complete.
0 Errors, 0 Warnings, 0 Informational Messages
```

Figure 15-33 Running ejbdeploy on a JAR without a schema or map

If we examine the deployed EJB JAR file generated by `ejbdeploy`, we can see the new schema and map files (Figure 15-34).

```
D:\DeployTool>jar tf D:\temp\deployed-piggybank-ejb.jar META-INF
META-INF/Map.mapxmi
META-INF/Schema/Schema.dbxmi
META-INF/Table.ddl
META-INF/ejb-jar.xml
META-INF/ibm-ejb-jar-bnd.xmi
META-INF/ibm-ejb-jar-ext.xmi
META-INF/MANIFEST.MF
```

Figure 15-34 Generated schema, map, and DDL files

The tool also generated a `Table.ddl` file—this contains DDL that we can use to create our database tables (Figure 15-35).

```
-- Generated by Relational Schema Center on Thu Jul 26 14:34:20 PDT 2001

CREATE TABLE "CUSTOMER"
  ("ID" INTEGER NOT NULL,
   "NAME" VARCHAR(32));

ALTER TABLE "CUSTOMER"
  ADD CONSTRAINT "CUSTOMERPK" PRIMARY KEY ("ID");

CREATE TABLE "ACCOUNT"
  ("NUMBER" INTEGER NOT NULL,
   "BALANCE" INTEGER,
   "CHECKING" SMALLINT,
   "CUSTOMERID" INTEGER);

ALTER TABLE "ACCOUNT"
  ADD CONSTRAINT "ACCOUNTPK" PRIMARY KEY ("NUMBER");
```

Figure 15-35 Contents of the DDL file to create the tables

If we run the same command against a JAR that does contain a schema and map, the default map is missing, and the message highlighted in Figure 15-33 is not reported, as seen in Figure 15-36.

```
D:\DeployTool>ejbdeploy d:\itso4ad\dev\modules\piggybank-ejb.jar d:\temp
                        d:\temp\deployed-piggybank-ejb.jar
Starting workbench.
Creating the project.
Importing file..
Starting Validation.
Generating deployment code
Building: /deployed-piggybank-ejb.jar.
Invoking RMIC.
Building: /deployed-piggybank-ejb.jar.
Generating DDL
Shutting down workbench.
EJBDeploy complete.
0 Errors, 0 Warnings, 0 Informational Messages
```

Figure 15-36 Running ejbdeploy on a JAR with a schema and map

If the default mapping is not suitable for our target database we can extract the schema and map files from the deployed JAR file, edit them to suit our needs, then create a new JAR file which we can then run through `ejbdeploy` a second time to generate new persistence mapping code for us, based on the modified files.

This enables us to modify the default CMP persistence mapping, even if we do not have access to Version 4.0 of VisualAge for Java. As we will see, however, the XML files are somewhat cryptic, so we still recommend you use VisualAge for Java Version 4 if the option is available to you.

The standalone deployment tool available for download from VisualAge Developer Domain includes some documentation on how to modify the schema and map files—we anticipate this documentation will be included in some future revision of the WebSphere InfoCenter. We refer you to the documentation for a full description of the files.

Tip: If you decide you want to edit the generated schema and map files yourself, you may find it helpful to use an XML-aware editor.

Schema file

An extract from the PiggyBank schema file for DB2 showing the `CUSTOMER` table is shown in Figure 15-37.

The XML first defines the database and the SQL types available for the target database—in this case DB2 UDB Version 7. It then goes on to define the `CUSTOMER` table, and the primary key constraint on the `ID` column. Each column in the table is described, in the order in which they are defined in the database. In this case we see the ID column defined as an `INTEGER` that may not be null. The name column is a `VARCHAR(32)` and may be null.

```
<xmi:XMI xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
         xmlns:RDBSchema="RDBSchema.xmi">
  <RDBSchema:RDBDatabase xmi:id="RDBDatabase_1" name="TopDownDB"
                         tableGroup="CUSTOMER ACCOUNT">
    <dataTypeSet href="UDBV7_Primitives.xmi#SQLPrimitives_1"/>
  </RDBSchema:RDBDatabase>
  <RDBSchema:RDBTable xmi:id="CUSTOMER" name="CUSTOMER"
                      primaryKey="SQLReference_1" database="RDBDatabase_1">
    <namedGroup xmi:type="RDBSchema:SQLReference" xmi:id="SQLReference_1"
                members="RDBColumn_1" table="CUSTOMER"
                constraint="Constraint_CUSTOMERPK"/>
    <constraints xmi:id="Constraint_CUSTOMERPK" name="CUSTOMERPK"
                 type="PRIMARYKEY" primaryKey="SQLReference_1"/>
    <columns xmi:id="RDBColumn_1" name="ID" allowNull="false"
             group="SQLReference_1">
      <type xmi:type="RDBSchema:SQLExactNumeric" xmi:id="SQLExactNumeric_1">
        <originatingType xmi:type="RDBSchema:SQLExactNumeric"
                         href="UDBV7_Primitives.xmi#SQLExactNumeric_1"/>
      </type>
    </columns>
    <columns xmi:id="RDBColumn_2" name="NAME">
      <type xmi:type="RDBSchema:SQLCharacterStringType"
            xmi:id="SQLCharacterStringType_1" characterSet="800"
            length="32">
        <originatingType xmi:type="RDBSchema:SQLCharacterStringType"
                      href="UDBV7_Primitives.xmi#SQLCharacterStringType_3"/>
      </type>
    </columns>
  </RDBSchema:RDBTable>
  [ ACCOUNT TABLE NOT SHOWN ........... ]
</xmi:XMI>
```

Figure 15-37 Extract from the PiggyBank schema file

If we want to modify the database schema we must modify this file. Changing the table and column names is fairly straightforward—we simply locate the names and alter them as required. Changing the type of columns is a little more involved—we must alter the `href` to refer to the correct SQL primitive for the column type we want.

The SQL primitives for each database are defined in XML documents stored in:

```
DeployTool\plugins\com.ibm.etools.sqlmodel\runtime\primitives
```

Where, `DeployTool` is the location where you installed the standalone tool downloaded from VADD. In the AE and updated AEs products we expect, but cannot confirm, that this directory will be located in the WebSphere install directory.

Figure 15-38 shows an extract from the document that defines the primitives for Oracle.

```
<RDBSchema:SQLPrimitives xmi:version="2.0"
                         xmlns:xmi="http://www.omg.org/XMI"
                         xmlns:RDBSchema="RDBSchema.xmi"
                         xmi:id="SQLPrimitives_1" domain="ORACLE_V8">
  <types xmi:type="RDBSchema:SQLBinaryLargeObject"
         xmi:id="SQLBinaryLargeObject_1" externalName="BINARY LARGE OBJECT"
         name="BLOB" jdbcEnumType="2004" domain="ORACLE_V8"
         requiredUniqueInstance="false" renderedString="BLOB"
         typeEnum="BINARYLARGEOBJECT"
         formatterClassName=
                   "com.ibm.etools.rdbschemagen.formatter.oracle.SimpleTextFormatter"
         length="4" multiplier="G"/>
  <types xmi:type="RDBSchema:SQLCharacterStringType"
         xmi:id="SQLCharacterStringType_1" externalName="CHARACTER"
         name="CHAR" jdbcEnumType="1" domain="ORACLE_V8"
         requiredUniqueInstance="true" renderedString="CHAR"
         typeEnum="CHARACTER"
         formatterClassName=
                "com.ibm.etools.rdbschemagen.formatter.oracle.CharacterTextFormatter"
         characterSet="800" length="1"/>
  <types xmi:type="RDBSchema:SQLCharacterStringType"
         xmi:id="SQLCharacterStringType_2" externalName="CHARACTER VARYING"
         name="VARCHAR2" jdbcEnumType="12" domain="ORACLE_V8"
         requiredUniqueInstance="true" renderedString="VARCHAR2"
         typeEnum="CHARACTERVARYING"
         formatterClassName=
                "com.ibm.etools.rdbschemagen.formatter.oracle.CharacterTextFormatter"
         characterSet="800" length="1"/>
  <types xmi:type="RDBSchema:SQLNumeric" xmi:id="SQLNumeric_6" externalName="DECIMAL"
         name="NUMBER" jdbcEnumType="3" domain="ORACLE_V8"
         requiredUniqueInstance="true" renderedString="NUMBER" typeEnum="DECIMAL"
         formatterClassName=
                  "com.ibm.etools.rdbschemagen.formatter.oracle.NumericTextFormatter"
         precision="5" scale="0"/>
...
```

Figure 15-38 Extract from the Oracle SQL primitives document

Map file

An extract from the PiggyBank map file that defines the mapping between the customer EJB and the table defined in the schema is shown in Figure 15-39.

```
<ejbrdbmapping:EjbRdbDocumentRoot xmi:version="2.0"
               xmlns:xmi="http://www.omg.org/XMI"
               xmlns:ejbrdbmapping="ejbrdbmapping.xmi" xmlns:ejb="ejb.xmi"
               xmlns:RDBSchema="RDBSchema.xmi" xmlns:Mapping="Mapping.xmi"
               xmi:id="EjbRdbDocumentRoot_1" outputReadOnly="false"
               topToBottom="true">
  <helper xmi:type="ejbrdbmapping:RdbSchemaProperies"
          xmi:id="RdbSchemaProperies_1" primitivesDocument="DB2UDBNT_V71">
    <vendorConfiguration
                    href="RdbVendorConfigurations.xmi#DB2UDBNT_V71_Config"/>
  </helper>
  <inputs xmi:type="ejb:EJBJar" href="META-INF/ejb-jar.xml#ejb-jar_ID"/>
  <outputs xmi:type="RDBSchema:RDBDatabase"
           href="META-INF/Schema/Schema.dbxmi#RDBDatabase_1"/>
  <nested xmi:type="ejbrdbmapping:RDBEjbMapper" xmi:id="RDBEjbMapper_1">
    <helper xmi:type="ejbrdbmapping:PrimaryTableStrategy"
            xmi:id="PrimaryTableStrategy_1">
      <table href="META-INF/Schema/Schema.dbxmi#CUSTOMER"/>
    </helper>
    <inputs xmi:type="ejb:ContainerManagedEntity"
            href="META-INF/ejb-jar.xml#Customer"/>
    <outputs xmi:type="RDBSchema:RDBTable"
             href="META-INF/Schema/Schema.dbxmi#CUSTOMER"/>
    <nested xmi:id="Customer_id---CUSTOMER_ID">
      <inputs xmi:type="ejb:CMPAttribute"
              href="META-INF/ejb-jar.xml#Customer_id"/>
      <outputs xmi:type="RDBSchema:RDBColumn"
               href="META-INF/Schema/Schema.dbxmi#RDBColumn_1"/>
      <typeMapping href="JavatoDB2UDBNT_V71TypeMaps.xmi#int-INTEGER"/>
    </nested>
    <nested xmi:id="Customer_name---CUSTOMER_NAME">
      <inputs xmi:type="ejb:CMPAttribute"
              href="META-INF/ejb-jar.xml#Customer_name"/>
      <outputs xmi:type="RDBSchema:RDBColumn"
               href="META-INF/Schema/Schema.dbxmi#RDBColumn_2"/>
      <typeMapping href="JavatoDB2UDBNT_V71TypeMaps.xmi#String-VARCHAR"/>
    </nested>
  </nested>
  [ ACCOUNT TABLE NOT SHOWN ............]
</ejbrdbmapping:EjbRdbDocumentRoot>
```

Figure 15-39 Extract from the PiggyBank map file

The following XML elements are used:

`helper`	Defines database-specific helper objects used to set schema properties, table strategy, and as converters between attribute and column types
`inputs`	Used to identify CMP references in the EJB deployment descriptor—uses an `href` tag that uses a path relative to the `META-INF` directory
`outputs`	Used to identify database table references in the schema document, also using an `href` tag that uses a path relative to the `META-INF` directory
`nested`	Defines child mappings for each input and output
`typeMapping`	Defines the type mapper to be used for a CMP field—the mappers are defined in documents in the directory `DeployTool\plugins\com.ibm.etools.ejbdeploy\typemaps`

The `inputs` and `outputs` tags are paired—in our example there is a pair that maps from the JAR file to the database in the schema, another that maps from the EJB in the deployment descriptor to the table in the schema document, and two further pairs that map each of the two CMP fields for the customer EJB to the appropriate columns in the database schema.

Migrating and validating EJB JAR files

We can use the `ejbdeploy` tool to help us migrate existing EJB 1.0 JAR files to the new EJB 1.1 specification. There are three areas of concern where the tool can assist us:

- Validating our EJB code to complies with the EJB 1.1 specification
- Converting EJB 1.0 serialized deployment descriptors to EJB 1.1 XML deployment descriptors
- Migrating CMP beans to work without modifying our database schema

We discuss each in turn, using the standard sample EJBs that are delivered with VisualAge for Java in our examples.

Validating EJB code

In “Developing EJBs in VisualAge for Java” on page 266 we discussed some of the differences between EJB code developed for Versions 1.0 and 1.1 of the EJB specification. As we work through our code modifying it to comply with the new specification level we can use `ejbdeploy` to check that we have made all the required changes—it is very easy to miss out the odd `RemoteException` or `ejbCreate` method that needs updating.

To demonstrate this we run the EJB deployment tool on `sample11.jar`—this contains the unaltered VisualAge for Java samples exported as a 1.1 JAR file using the VisualAge *Export -> EJB 1.1 JAR* menu option.

The output of the `ejbdeploy` command is shown in Figure 15-40.

```
D:\DeployTool>ejbdeploy D:\temp\sample11.jar D:\temp D:\temp\deployed-sample.jar
Starting workbench.
Creating the project.
Importing file..
Starting Validation.
[*Warning] /deployed-sample.jar(0): CHKJ2031W: Home interface IncrementHome defines a create
method, Public abstract com.ibm.ivj.ejb.samples.increment.Increment
com.ibm.ivj.ejb.samples.increment.IncrementHome.create(com.ibm.ivj.ejb.samples.increment.Increment
Key) throws javax.ejb.CreateException,java.rmi.RemoteException, but there is no matching
ejbPostCreate method. While the deployment code can be generated if this method is absent, the
generated code does not invoke ejbPostCreate as it should. The EJB 1.1 specification states that
this method must be implemented, and the EJB 1.0 specification intended for this method to be
implemented, but did not state that it was a requirement. Read section 9.2.4 of the EJB 1.1
specification for details.
[*Warning] /deployed-sample.jar(0): CHKJ2406W: The method Public void
com.ibm.ivj.ejb.samples.increment.IncrementBean.ejbCreate(com.ibm.ivj.ejb.samples.increment.Increm
entKey) throws javax.ejb.CreateException,java.rmi.RemoteException declared in bean
com.ibm.ivj.ejb.samples.increment.IncrementBean should return the primary key type
com.ibm.ivj.ejb.samples.increment.IncrementKey. Read section 9.2.3 (BMP beans) or 9.4.2 and
9.4.7.3 (CMP beans) of the EJB 1.1 specification for details. The EJB 1.1 specification has
changed the method signature required on CMP beans' ejbCreate method.
[*Warning] /deployed-sample.jar(0): CHKJ2400W: Public void
com.ibm.ivj.ejb.samples.increment.IncrementBean.ejbCreate(com.ibm.ivj.ejb.samples.increment.Increm
entKey) throws javax.ejb.CreateException,java.rmi.RemoteException throws a
java.rmi.RemoteException. This is permitted in the EJB 1.0 specification, but deprecated in the
EJB 1.1 specification. In the EJB 1.1 specification, a javax.ejb.EJBException (or another
java.lang.RuntimeException) should be thrown to indicate non-application exceptions.
[*Warning] /deployed-sample.jar(0): CHKJ2002W: IncrementBean implements an ejbCreate method with
signature Public void
com.ibm.ivj.ejb.samples.increment.IncrementBean.ejbCreate(com.ibm.ivj.ejb.samples.increment.Increm
entKey) throws javax.ejb.CreateException,java.rmi.RemoteException but does not implement the
matching ejbPostCreate method. While the deployment code can be generated if this method is
absent, the generated code does not invoke ejbPostCreate as it should. The EJB 1.1 specification
states that this method must be implemented, and the EJB 1.0 specification intended for this
method to be implemented, but did not state that it was a requirement. Read section 9.2.4 of the
EJB 1.1 specification for details.
[*Warning] /deployed-sample.jar(0): CHKJ2400W: Public void
com.ibm.ivj.ejb.samples.helloworld.HelloWorldBean.ejbCreate() throws
javax.ejb.CreateException,java.rmi.RemoteException throws a java.rmi.RemoteException. This is
permitted in the EJB 1.0 specification, but deprecated in the EJB 1.1 specification. In the EJB
1.1 specification, a javax.ejb.EJBException (or another java.lang.RuntimeException) should be
thrown to indicate non-application exceptions.
Generating deployment codeBuilding: /deployed-sample.jar.
Invoking RMIC.
Building: /deployed-sample.jar.
Generating DDL
Shutting down workbench.
EJBDeploy complete.
0 Errors, 5 Warnings, 0 Informational Messages
```

Figure 15-40 Using ejbdeploy to validate an EJB JAR file

As you can see from the figure the tool generated five warnings from the two EJBs in the JAR file. A quick scan through the messages reveals the problems:

- The `Increment` entity bean does not define an `ejbPostCreate` method to match its `ejbCreate` method.
- The `Increment` entity bean `ejbCreate` method returns `void` (as required by the 1.0 specification) instead of the primary key type.
- The Increment `ejbCreate` method throws a `RemoteException`, deprecated in the 1.1 specification.
- Another missing `ejbPostCreate` in `Increment`.
- Another `RemoteException` thrown by `Increment`.

In this case we only found warnings, so we can run the EJBs in WebSphere 4.0 unmodified if we desire. If we want to make our beans truly J2EE compliant, however, we must make the changes suggested by the tool.

Converting serialized deployment descriptors

The 1.0 EJB specification required bean providers to store deployment information in serialized Java objects, using the `SessionDescriptor` and `EntityDescriptor` classes in the `javax.ejb.deployment` package. The 1.1 EJB specification, however, now requires us to store this same information in XML files.

If we are using Version 4.0 of VisualAge for Java, we can easily create these XML deployment descriptors using the *Export -> EJB 1.1 JAR* menu option on an EJB group. If we do not have the latest version of VisualAge for Java, however, that option is not available to us.

We can use the EJB deployment tool to save us the effort of manually creating the XML descriptors, either by hand in an editor, or by entering the information into AAT.

We start with a simple undeployed EJB 1.0 JAR, `sample10.jar`, and run it through `ejbdeploy` (Figure 15-41).

Then we extract the XML deployment descriptors from the deployed JAR file (Figure 15-42).

```
D:\DeployTool>ejbdeploy D:\temp\sample10.jar D:\temp D:\temp\deployed-sample.jar
Starting workbench.
Creating the project.
Importing file..
Importing MOF Resource..
Importing MOF Resource..
Importing MOF Resource..
Starting Validation.
[ SKIPPED WARNINGS ]
Creating Top-Down Map...
Generating deployment code
Building: /deployed-sample.jar.
Invoking RMIC.
Building: /deployed-sample.jar.
Generating DDL
Shutting down workbench.
EJBDeploy complete.
0 Errors, 5 Warnings, 0 Informational Messages
```

Figure 15-41 Running ejbdeploy on an EJB 1.0 JAR

```
D:\temp>jar xvf deployed-sample.jar META-INF
extracted: META-INF/Map.mapxmi
extracted: META-INF/Schema/Schema.dbxmi
extracted: META-INF/Table.ddl
extracted: META-INF/ejb-jar.xml
extracted: META-INF/ibm-ejb-jar-bnd.xmi
extracted: META-INF/ibm-ejb-jar-ext.xmi
extracted: META-INF/MANIFEST.MF
```

Figure 15-42 Extracting generated deployment descriptors

Migrating CMP beans

If we created our 1.0 EJBs using meet-in-the-middle or bottom-up mapping using VisualAge for Java, the best and by far the easiest way to migrate our schema and mapping information is to use the EJB 1.1 export feature of VisualAge version 4. The only alternative, described earlier in "Customizing CMP persistence mapping" on page 420, is not for the faint-hearted.

If we did not use VisualAge for Java to create our 1.0 EJBs, however, but we did deploy them into Version 3.5 of WAS, we would have had no choice but to accept the default top-down CMP persistence mapping in that version of the product.

The default top-down mapping strategy has changed in WAS Version 4—if we have been running our 1.0 EJBs in a Version 3.5 production system all our application data will be stored in a database with a different schema from the one the default Version 4 mappings would produce. On the face of it, this would leave us with two choices, neither of which may seem attractive:

- Migrate our database schema to match the Version 4 mapping
- Manually customize our CMP mapping, as described earlier

Fortunately, the EJB deployment tool provides an option that can help solve this problem. We can specify `-35` as an option on the `ejbdeploy` command line.

When we use this option to generate the schema and map files for the EJB 1.0 entity bean in the VisualAge for Java samples, we get a schema file that contains the following extract:

```
<RDBSchema:RDBTable xmi:id="INCREMENTBEANTbl" name="INCREMENTBEANTbl"
primaryKey="SQLReference_1" schema="RDBSchema_1" database="RDBDatabase_1">
```

We have highlighted the table name in the example—readers familiar with Version 3.5 of WebSphere will recognize the format of the table name.

16

Deploying to the test environment

In this chapter we describe the process of deployment for the following environments:

- WebSphere Application Server Version 4 **Advanced Edition** (AE)
- WebSphere Application Server Version 4 **Single Server Edition** (AEs)

Relating to these environments, we describe the following tools:

- EARExpander, to expand/collapse enterprise application (EAR) files
- SEAppInstall, to install applications in AEs
- Administrative Console, both for AEs (Web-based console) and AE (standalone console)
- Application Client Resource Configuration, to configure resources associated to application clients
- XMLConfig, to export/import configuration data from the administrative repository
- WSCP, the command-line and scripting interface for administering WebSphere applications

EARExpander command line tool

A new feature of WebSphere Version 4.0, the EARExpander command line tool's function is to expand enterprise application files (.ear) into the format desired by the application server runtime. It can also generate EAR files from a suitable directory structure (reverse operation).

The usage of the EARExpander tool is shown in Figure 16-1.

```
Usage: java com.ibm.websphere.install.se.EARExpander
-ear <ear file or directory>
-expandDir <directory in which to expand ear>
-operation <expand | collapse>
[-expansionFlags <all | war>]
```

Figure 16-1 Usage of the EARExpander tool

An example of expanding an application is shown in Figure 16-2.

```
D:\WebSphereSSE\AppServer\bin>earexpander
       -ear ..\installableApps\PiggyBank.ear -expandDir ..\temp\PiggyBank
       -operation expand
IBM WebSphere Application Server Standard Edition, Release 4.0
J2EE Application Expansion Tool, Version 1.0
Copyright IBM Corp., 1997-2001

Expanding ..\installableApps\PiggyBank.ear into ..\temp\PiggyBank

D:\WebSphereSSE\AppServer\bin>
```

Figure 16-2 Expanding an EAR file with EARExpander

If the specified directory for expanding or collapsing does not exist, EARExpander creates it.

Figure 16-3 shows an example of collapsing a directory into an EAR file.

```
D:\WebSphereSSE\AppServer\bin>earexpander -ear ..\temp\PiggyBank.ear
       -expandDir ..\temp\PiggyBank -operation collapse
IBM WebSphere Application Server Standard Edition, Release 4.0
J2EE Application Expansion Tool, Version 1.0
Copyright IBM Corp., 1997-2001

Collapsing ..\temp\PiggyBank into ..\temp\PiggyBank.ear

D:\WebSphereSSE\AppServer\bin>
```

Figure 16-3 Collapsing a directory into an EAR file using EARExpander

This tool can be useful when we want to expand an application for viewing and/or updating (although we can view the application by using the Application Assembly Tool, we cannot edit single files, such as JSPs).

For example, when we want to change a class file, we can just expand the module file and change the class (in installed applications we can do this directly on the directory structure under `$WASPATH$\installedApps`, but we have to restart the server afterwards for the changes to take effect).

SEAppInstall command line tool

The SEAppInstall tool is a command line-based tool that allows the user to install enterprise applications in the WebSphere Application Server 4.0 Advanced Edition **Single Server** (for the AE, the enterprise applications are installed using the Admin Console).

Using this tool to install the applications is equivalent to do the installation from the browser-based administrative console, because the console calls the SEAppInstall tool to perform this task. The difference that might make this tool more useful for experienced users is that it allows several options for each command, while the console uses the default options, that might not be suitable in all occasions.

This is the list of features that the SEAppInstall tool provides:

- Install/uninstall enterprise applications

  The uninstall option removes the application data from the configuration files, but by default it does not remove the files from the `installedApps` folder (manual deletion is necessary). To remove the files, first stop the server and then delete them.

To get the files automatically removed using SEAppInstall, select the option `-delete true` when executing the uninstall command. The directory structure is deleted, but not the JAR files included in the modules (for example, utilities JAR file).

- Export enterprise applications installed in the server as an EAR file with binding information
- List applications/modules installed in the server
- Extract DDL for database creation from EJB JAR file configuration data (when using entity beans).
- Validate application and servers configurations

When the option `-ejbDeploy true` is selected, the SEAppInstall tool calls the EJBDeploy tool to perform the deployment of all the EJBs in the module. For usage of the EJBDeploy refer to "EJB deployment tool" on page 418.

The usage of the SEAppInstall tool is shown in Figure 16-4.

When installing an application, precompiling the JSP code (`-precompileJsp true`, the default option) means a faster execution of the pages when we first load them after installation.

In general, when we are installing a module previously assembled with the AAT (or other tool such as Ant, as we showed in "Using Ant to build a WebSphere application" on page 197), we should not have to redeploy the EJBs, and all the necessary binding information should be already included in the module deployment descriptor, but the SEAppInstall tool provides options for performing both of these task in case it is necessary.

The SEAppInstall tool updates the server configuration file with the changes due to the operation performed (includes new data for the newly installed applications or erases data related to the uninstalled ones). The default value for the server configuration file is `server-cfg.xml` (regardless of the configuration file we have started the server with).

```
java com.ibm.websphere.install.se.SEApplicationInstaller
    -install <ear file or directory>
   [-configFile <server configuration file>]
   [-expandDir <directory in which to expand ear>]
   [-nodeName <name of node>]
   [-serverName <name of server>]
   [-ejbdeploy <TRUE | false>]
   [-precompileJsp <TRUE | false>]
   [-validate <app | server | both | NONE>]
   [-denyAll <TRUE | false>]
   [-interactive {TRUE | false}]
       If you selected "-interactive false", you will not be asked an
       questions and you will not be able to specify binding data

java com.ibm.websphere.install.se.SEApplicationInstaller
    -uninstall <application name>
   [-delete <true | false>]
   [-configFile <server configuration file>]
   [-nodeName <name of node>]
   [-serverName <name of server>]

java com.ibm.websphere.install.se.SEApplicationInstaller
    -export <application name>
   [-configFile <server configuration file>]
    -outputFile <name of the ear file to create>

java com.ibm.websphere.install.se.SEApplicationInstaller
    -list <apps | wars | ejbjars | all>
   [-configFile <server configuration file>]

java com.ibm.websphere.install.se.SEApplicationInstaller
    -extractDDL <application name>
   [-DDLPrefix <Prefix to apply to front of all DDL file names>]
   [-configFile <server configuration file>]

java com.ibm.websphere.install.se.SEApplicationInstaller
    -validate <app | server | both | NONE>
   [-ear <ear file>]
   [-configFile <server configuration file>]
      If you specify "-validate app" or "-validate both", you must
      include the "-ear" option. If you specify "-validate server"
      or "-validate both", the "-configFile" option is optional.
```

Figure 16-4 Usage of the SEAppInstall command tool

Figure 16-5 shows an example of the output of the SEAppInstall tool for the installation of an enterprise application (EAR) module.

```
D:\WebSphereSSE\AppServer\bin>seappinstall -install
..\installableapps\SampleApp.ear -configFile ..\config\server-cfg.xml
-expandDir ..\installedapps\SampleApp.ear -nodeName 23bk55y -ejbDeploy false
-precompileJsp false -validate both -interactive false
IBM WebSphere Application Server Release 4, AEs
J2EE Application Installation Tool, Version 1.0
Copyright IBM Corp., 1997-2001

Loading Server Configuration from
D:\WebSphereSSE\AppServer\config\server-cfg.xml
Using Server on Node: 23bk55y
Server Configuration Loaded Successfully
Loading D:\WebSphereSSE\AppServer\installableApps\sampleApp.ear
Validating Application DD and Extensions...
Validating class definition of com.transarc.jmon.examples.Inc.IncBean.
Validating method ejbCreate on com.transarc.jmon.examples.Inc.IncBean.
CHKJ2406E: The method Public void
com.transarc.jmon.examples.Inc.IncBean.ejbCreate(com.transarc.jmon.examples.
Inc.IncKey) declared in bean com.transarc.jmon.examples.Inc.IncBean should
return the primary key type com.transarc.jmon.examples.Inc.IncKey. Read
section 9.2.3 (BMP beans) or 9.4.2 (CMP beans) of the EJB 1.1 specification
for details. The EJB 1.1 specification has changed the method signature
required on CMP beans ejbCreate method.
... (other errors in validation)
... (validation of other methods/classes and possible errors)
Access will be denied to all unprotected methods.
Installed EAR On Server
Validating Application Bindings...
... (errors related to bindings)
Validating Server Configuration...
... (errors related to server configuration)
Finished validating Server Configuration.
Saving EAR File to directory
Saved EAR File to directory Successfully
Saving Server Configuration to
D:\WebSphereSSE\AppServer\config\server-cfg.xml
Backing Up Server Configuration to:
D:\WebSphereSSE\AppServer\config\server-cfg.xml~
Save Server Config Successful
JSP Pre-compile Skipped......
Installation Completed Successfully
```

Figure 16-5 Output of the SEAppInstall tool

In the Single Server Edition (AEs) we have only one node (machine) and one application server. These options can be different for different configuration files (though there can't be more than one node or server per file).

The output when uninstalling an application is shown in Figure 16-6.

```
D:\WebSphereSSE\AppServer\bin>seappinstall -uninstall sampleApp -configFile
..\config\server-cfg.xml

IBM WebSphere Application Server Release 4, AEs
J2EE Application Installation Tool, Version 1.0
Copyright IBM Corp., 1997-2001

Loading Server Configuration from
D:\WebSphereSSE\AppServer\config\server-cfg.xml
Server Configuration Loaded Successfully
Removed Application From Server: sampleApp
Saving Server Configuration to
D:\WebSphereSSE\AppServer\config\server-cfg.xml
Backing Up Server Configuration to:
D:\WebSphereSSE\AppServer\config\server-cfg.xml~
Save Server Config Successful
```

Figure 16-6 Uninstalling applications with SEAppInstall

The option to list applications is useful when we have several configuration files with different installed applications. We can list the enterprise application, the Web modules and the EJB modules (or all of them). Figure 16-7 shows an example of using this option.

```
D:\WebSphereSSE\AppServer\bin>seappinstall -list all -configFile
..\config\server-cfg.xml
IBM WebSphere Application Server Release 4, AEs
J2EE Application Installation Tool, Version 1.0
Copyright IBM Corp., 1997-2001

Loading Server Configuration from
D:\WebSphereSSE\AppServer\config\server-cfg.xml
Server Configuration Loaded Successfully

Installed Applications
--------------------------------------------------------------------
1) another
2) Server Administration Application
3) sampleApp
4) PiggyBank Application

Installed Web Applications
--------------------------------------------------------------------
1) Server Administration Application:admin [WebModuleRef_3]: Admin servlets
2) sampleApp:examples [WebModuleRef_2]: example servlets
3) sampleApp:default_app [WebModuleRef_1]: default application
4) PiggyBank.ear:piggy_bank [WebModuleRef_4]: PiggyBank application

Installed EJB Jars
--------------------------------------------------------------------
1) another:another.ear::deployedtestsejs.jar [EJBModuleRef_3]:
deployedtestsejs jar
2) sampleApp:sampleApp.ear::Increment.jar [EJBModuleRef_1]: null
3) PiggyBank.ear:PiggyBank.ear::PiggyBankEJB.jar [EJBModuleRef_3]: null
```

Figure 16-7 Listing installed applications using SEAppInstall

Single Server Edition: the browser-based console

In this section we discuss the configuration of the Single Server Edition through the Web browser.

Starting the application server

Before launching the administrative console in the browser, it is necessary to start the server. AEs does not install a Windows service to launch the server; this task is performed by executing a command file:

```
D:\WebSphereSSE\AppServer\bin\startstd.bat
```

The usage of the `startstd` command is shown in Figure 16-8.

```
D:\WebSphereSSE\AppServer\bin>startstd ?help
IBM WebSphere Technology For Developers, Release 1.0
Copyright IBM Corp., 1997-2001

Usage:
        java com.ibm.ws.runtime.StandardServer
        [-configFile <server configuration file>]
        [-nodeName <name of node>]
        [-serverName <name of server>]
        [-traceString <package name>]
        [-traceFile <file name>]

If no configuration file is specified and the environment variable
server.root is set to the WebSphere installation root, then
the default configuration file, server-cfg.xml, will be used.
```

Figure 16-8 Usage of the startstd command

In the Single Server Edition, it is possible to have only one application server installed and only one machine, as it is intended for unit testing (each developer can have it installed on his/her machine).

The server configuration files are stored in

```
D:\WebSphereSSE\AppServer\config
```

The default file to use when loading the server is `server-cfg.xml`, though we can have other configuration files with different installed applications, trace level settings, and so forth (this acts as having different servers, though we can start only one at a time, unlike in the Advanced Edition). We will only be able to launch the applications that are installed in the currently loaded configuration file. The administrative application is included in a separate configuration file, `admin-server-cfg.xml`.

To generate new configuration files, we can edit them by hand or modify the configuration on the console and then save it to a new file. The console provides the option of creating a new configuration file using the current one as a template; this option is useful for users not familiar with the syntax of the configuration files.

The messages displayed in the command window when starting the server are essentially in the same format as the ones displayed in the WAS Advanced Edition standalone Admin Console (a sample output is shown in Figure 16-9).

```
D:\WebSphereSSE\AppServer\bin>startstd
IBM WebSphere Technology For Developers, Release 1.0
Copyright IBM Corp., 1997-2001

************ Start Display Current Environment ************
WebSphere AEs 4.0.0 n0117.04 running with process name Default Server and
                             process id515
Host Operating System is Windows NT, version 4.0
Java version = J2RE 1.3.0 IBM build cn130-20010329 (JIT enabled: jitc),
               Java Compiler = jitc
server.root = D:\WebSphereSSE\AppServer
Java Home = D:\WebSphere\AppServer\java\jre
ws.ext.dirs = D:\WebSphere\AppServer\java/lib;D:\WebSphereSSE\AppServer/cla
   sses;D:\WebSphereSSE\AppServer/lib/ext;D:\WebSphereSSE\AppServer/lib;D:\W
   ebSphereSSE\AppServer/web/help
Classpath = D:\WebSphereSSE\AppServer/lib/bootstrap.jar;D:\WebSphereSSE\App
   Server/properties
Java Library path = D:\WebSphere\AppServer\java\bin;.;C:\WINNT\System32;C:\W
   INNT;D:\IBMDebug\bin;C:\Program Files\ibm\gsk5\lib; D:\IBM Connectors\Enc
   ina\bin;D:\IBMCON~1\CICS\BIN;C:\WINNT\system32;C:\WINNT;c:\lotus\notes;C:
   \ProgramFiles\Pcom;D:\IBM\IMNNQ;D:\WebSphere\AppServer\bin;D:\SQLLIB\BIN;
   D:\SQLLIB\FUNCTION;D:\SQLLIB\SAMPLES\REPL;D:\SQLLIB\HELP;D:\WebSphereSSE\
   AppServer\bin;c:\PSM;D:\WebSphereSSE\AppServer\bin
Current trace specification = com.ibm.ws.runtime.*=all=disabled
************* End Display Current Environment *************
[01.06.07 11:23:23:504 PDT] 3605e28d Server       U Version : 4.0.0
[01.06.07 11:23:23:544 PDT] 3605e28d Server       U Edition: AEs
[01.06.07 11:23:23:554 PDT] 3605e28d Server       U Build date: Mon Apr 30
                                                    00:00:00 PDT 2001
[01.06.07 11:23:23:554 PDT] 3605e28d Server       U Build number: n0117.04
[01.06.07 11:23:27:320 PDT] 3605e28d DrAdminServer I WSVR0053I: DrAdmin
                                     available on port 7000
... (Trace messages: launching the Servlet Engine, the HTTP Plugin tool,
     loading the applications, etc.)
[01.06.07 11:23:44:745 PDT] 3605e28d Server       I WSVR0023I: Server
                                     Default Server open for ebusiness
```

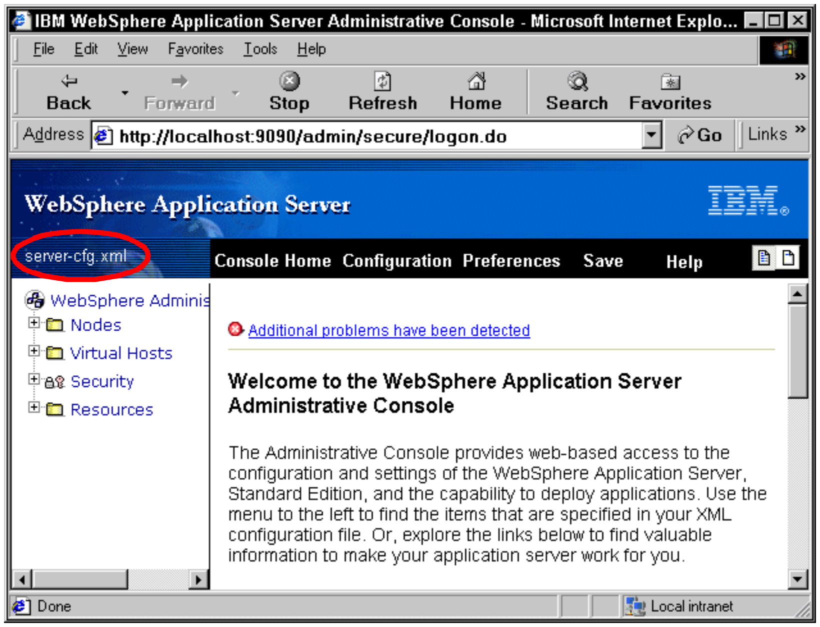
Figure 16-9 Starting the WAS 4.0 AEs server

Launching the administrative console in a browser

To launch the administrative console (after starting the server), enter the following URL in the browser:

```
http://servername:9090/admin
```

Figure 16-10 shows the interface of the browser-based administrative console.


IBM WebSphere Application Server Administrative Console - Microsoft Internet Explo...
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites
Address http://localhost:9090/admin/secure/logon.do Go Links
WebSphere Application Server
IBM
server-cfg.xml
Console Home Configuration Preferences Save Help
WebSphere Adminis
Nodes
Virtual Hosts
Security
Resources
Done
Local intranet

Additional problems have been detected

Welcome to the WebSphere Application Server Administrative Console

The Administrative Console provides web-based access to the configuration and settings of the WebSphere Application Server, Standard Edition, and the capability to deploy applications. Use the menu to the left to find the items that are specified in your XML configuration file. Or, explore the links below to find valuable information to make your application server work for you.

Figure 16-10 WAS Single Server Edition: the browser-based console

The left frame allows us to navigate within the structure of the application server, while the right frame displays the configuration settings and options (similarly to the AE edition).

Note that the current server configuration file is displayed in the screen. It is possible to open other configuration files by clicking on *Configuration* (top of the screen) and selecting the file we want to open. We have to refresh the configuration tree (left pane) manually. This feature lets us switch configurations for editing without having to stop and restart the server to load another configuration file.

Figure 16-11 shows the expanded tree.

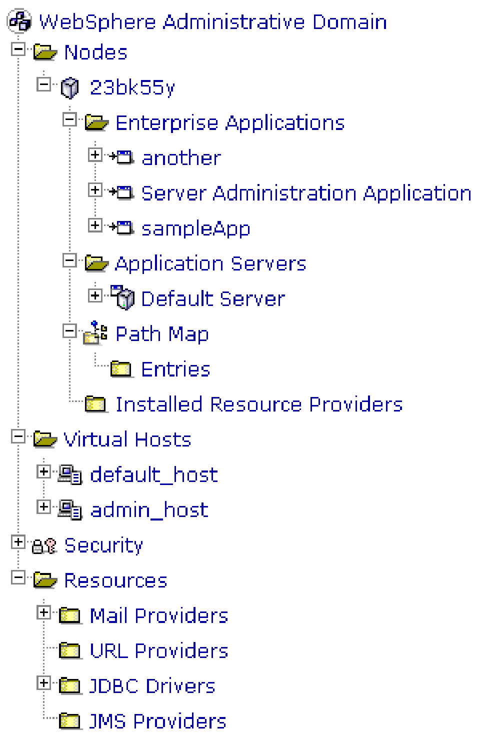


Figure 16-11 The options tree in the browser based console

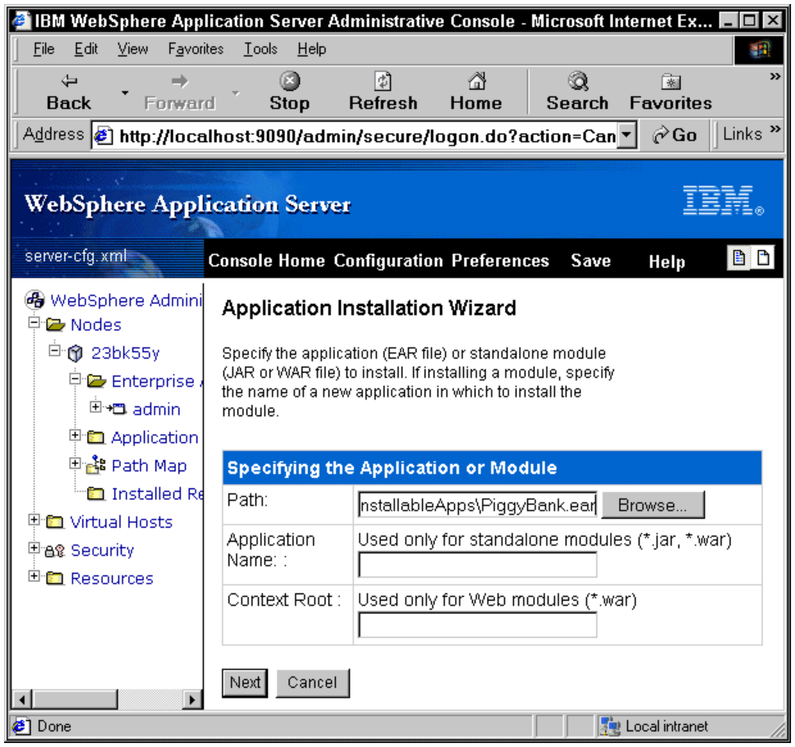
Administering applications though the console

The browser-based console enables us to perform many configuration tasks.

Installing applications

To install a new application, select *Nodes* -> *nodename* -> *Enterprise Applications*. The right frame shows a list of the installed applications. By clicking on *Install*, a new page is displayed where we enter the application file name (Figure 16-12).

The next steps guide us through the Application Installation Wizard, allowing us to select security roles, JNDI bindings for EJBs, virtual hosts and other parameters. This is equivalent to using the SEAppInstall tool with the option `-interactive true` (the default option).


IBM WebSphere Application Server Administrative Console - Microsoft Internet Ex...
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites
Address http://localhost:9090/admin/secure/logon.do?action=Can Go Links
WebSphere Application Server
IBM
server-cfg.xml
Console Home Configuration Preferences Save Help
WebSphere Admini
Nodes
23bk55y
Enterprise
admin
Application
Path Map
Installed Re
Virtual Hosts
Security
Resources
Done
Local intranet

Application Installation Wizard

Specify the application (EAR file) or standalone module (JAR or WAR file) to install. If installing a module, specify the name of a new application in which to install the module.

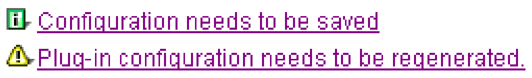
Specifying the Application or Module	
Path:	nstallableApps\PiggyBank.ear Browse...
Application Name: :	Used only for standalone modules (*.jar, *.war)
Context Root :	Used only for Web modules (*.war)

Next Cancel

Figure 16-12 Installing an application in the browser-based console

Interactive mode means that we are able to change the JNDI names for EJBs, EJB resource references or other resources, the virtual host, and so forth. The application installer extracts the information from the deployment descriptors and display it in the screen so that we can make changes.

After installing an application, the console displays a warning in the main frame indicating that the Web server plugin has to be regenerated and the configuration has to be saved (Figure 16-13).


Configuration needs to be saved

Plug-in configuration needs to be regenerated.

Figure 16-13 Warnings after installing an application

When installing applications through the SEAppInstall tool, it is necessary to regenerate the plugin configuration manually, by executing `GenPluginCfg.bat` as described in “Web server plugin” on page 454, though the configuration is saved automatically.

The new application is installed under `\installedApps`, in a folder with the same name as the EAR file: `..\installedApps\application-name.ear`.

Changing configuration settings

Settings related to the internal components of the enterprise application cannot be changed through the administrative console (use the Application Assembly Tool or edit the deployment descriptors manually to do this). Using the console we can modify the settings of the Default Server and install needed resources as JDBC drivers or DataSources.

The options for the server configuration are listed under *Default Server* (Figure 16-14).

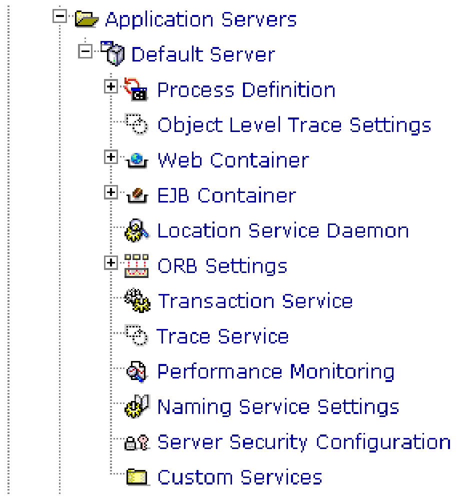


Figure 16-14 AEs server configuration settings

When setting up the environment for unit testing, we are interested in configuring these features:

- *OLT* settings—activate OLT to debug the application using the Distributed Debugger (see more about this topic in Chapter 17, “Debugging the application” on page 467).
- *Web container* settings—customize the properties for the Web container, such as allowing persistent sessions or cookies.

- *EJB container* settings—specify the default DataSource or the passivation directory for the EJBs.
- *Transaction service* settings—activate the transaction service to enable transactions within the server.
- *Trace service* settings—enable tracing to receive information about the events in the server (loading applications, invoking components or custom tracing).

Setting up resources

The resources are structured all in the same fashion: first it is necessary to create a *resource provider* (for example, a JavaMail provider or a JDBC driver). Later, we can create a *resource factory* relating to the provider, for example, a JavaMail session or a JDBC DataSource. These *resource factories* are available to the enterprise applications installed in the server.

The resources used by the applications are listed under the *Resources* folder (Figure 16-15).

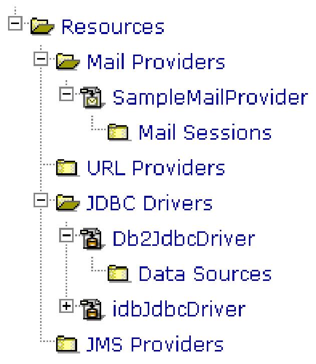


Figure 16-15 Resource settings

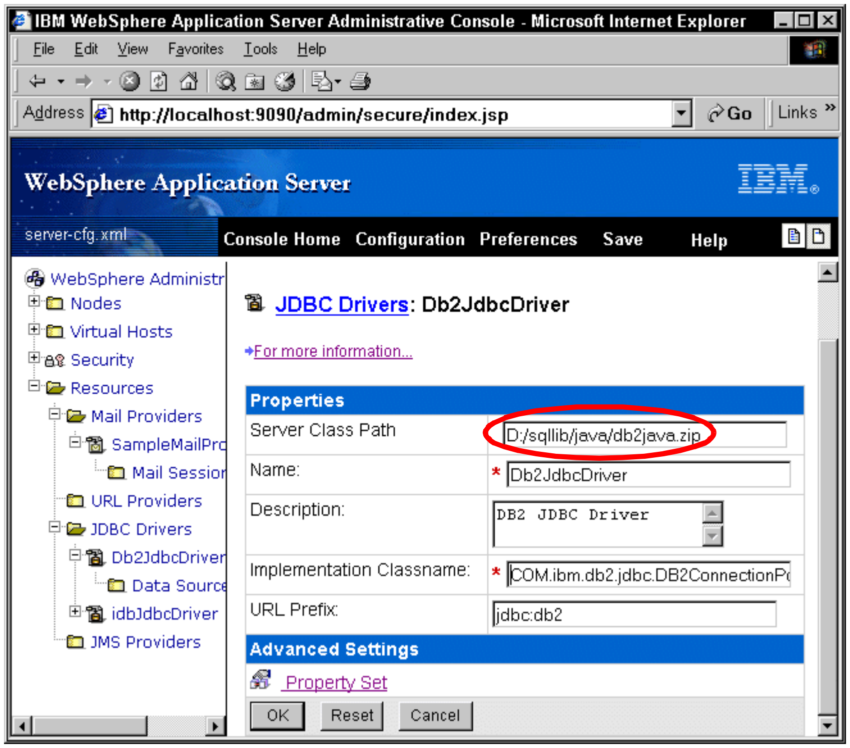
Here is a brief description on how to set up a DataSource, based on the PiggyBank application:

1. First install the JDBC driver (or use one of the existing). In our case, because we are using DB2 as the database for the PiggyBank application, we use one of the drivers already installed: `Db2JdbcDriver`, as it suites our needs.

   Specify the *implementation class* as well as the *class path* where it is located.

   Check the product documentation for the list of available drivers.

Setup the driver properties as shown in Figure 16-16.


IBM WebSphere Application Server Administrative Console - Microsoft Internet Explorer

Address: http://localhost:9090/admin/secure/index.jsp

JDBC Drivers: Db2JdbcDriver

For more information...

Properties

Server Class Path	D:/sqllib/java/db2java.zip
Name:	* Db2JdbcDriver
Description:	DB2 JDBC Driver
Implementation Classname:	* COM.ibm.db2.jdbc.DB2ConnectionP(
URL Prefix:	jdbc:db2

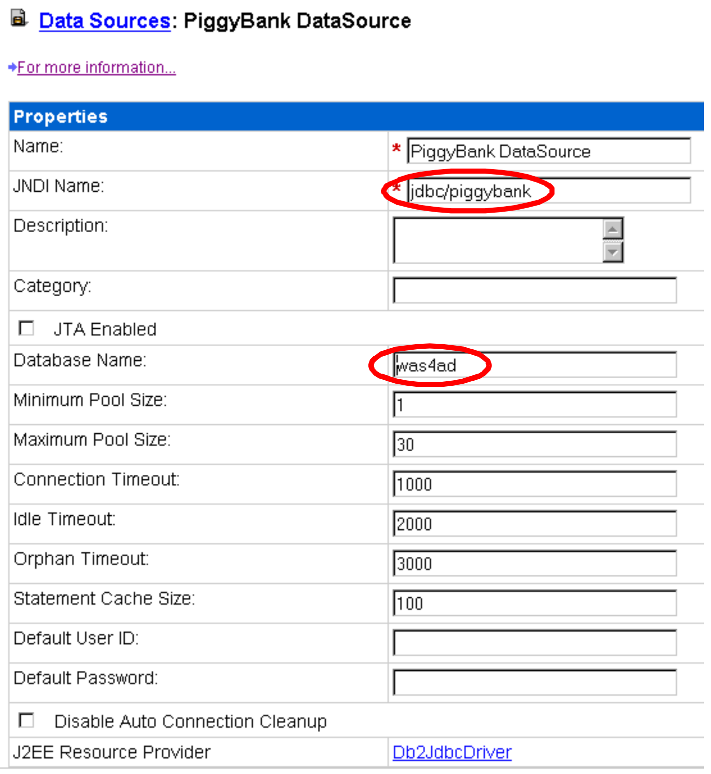
Advanced Settings

Property Set

Figure 16-16 Setting up the JDBC driver

2. Then, configure the DataSource using this driver.

   Setup the JNDI name and the database name, as well as the pooling properties (maximum/minimum number of connections, timeouts) as shown in Figure 16-17.


Data Sources: PiggyBank DataSource

For more information...

Properties	
Name:	* PiggyBank DataSource
JNDI Name:	* jdbc/piggybank
Description:	
Category:	
☐ JTA Enabled	
Database Name:	was4ad
Minimum Pool Size:	1
Maximum Pool Size:	30
Connection Timeout:	1000
Idle Timeout:	2000
Orphan Timeout:	3000
Statement Cache Size:	100
Default User ID:	
Default Password:	
☐ Disable Auto Connection Cleanup	
J2EE Resource Provider	Db2JdbcDriver

Figure 16-17 Setting up the DataSource properties

Uninstalling applications

Applications are uninstalled from the *Enterprise Applications* panel. After uninstallation, it is necessary to regenerate the plugin configuration (the appropriate warning appears in the console screen).

Uninstallation through the console does not delete the files from the uninstalled application (like executing SEAppInstall with the default option `-delete false`), so we have to delete them manually.

If we have just uninstalled an application and we want to install another one with the same name (for example, a newer version of the same application), we have to stop the server, delete the files, and start the server again before we are able to make the new installation. If we attempt to delete the files without stopping the server, the console does not allow us to do it and it displays a screen with instructions.

Stopping the AEs application server

To stop the server, execute:

```
D:\WebSphereSSE\AppServer\bin\stopserver.bat
```

The usage of the command is shown in Figure 16-18.

```
D:\WebSphereSSE\AppServer\bin>stopserver ?help
stopServer
Syntax: stopServer [-configFile "<server configuration file path>"]
```

Figure 16-18 Usage of the stopserver command

As usual, the default configuration file name used will be `server-cfg.xml`, if no other is specified.

Advanced Edition: the stand-alone console

In this section we discuss the configuration of the Advanced Edition through the stand-alone console.

Start and stop

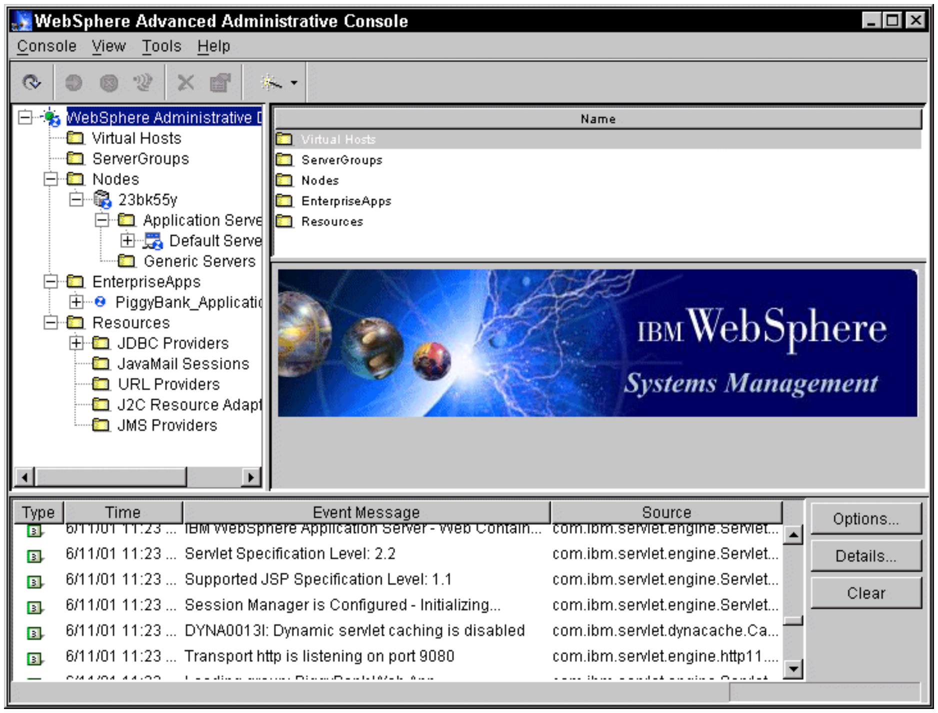
WebSphere Application Server Advanced Edition is normally started and stopped through the Windows Services panel. The service is named IBM WS AdminServer.

You can also use the Start menu and select *Start -> Programs -> IBM WebSphere -> Application Server v4.0 -> Start Admin Server*.

Starting the console

To start the Administrative Console select *Start -> Programs -> IBM WebSphere -> Application Server v4.0 -> Administrative Console*.

The console in Version 4.0 is quite similar to previous versions, though several options have changed to adapt the server to the J2EE model of packaging. Figure 16-19 shows the user interface.



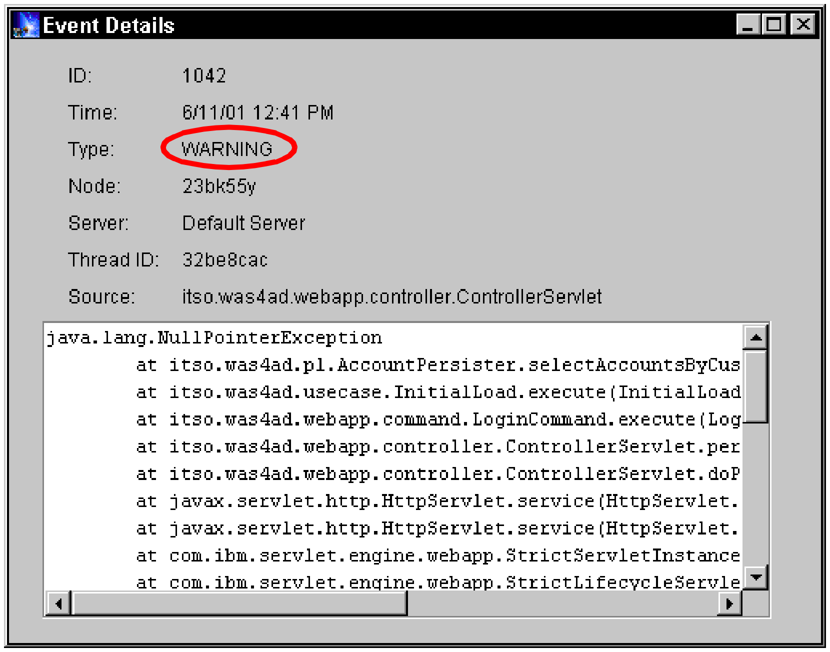
```
Type  Time             Event Message                                       Source
      6/11/01 11:23 ... IBM WebSphere Application Server - Web Contain...  com.ibm.servlet.engine.Servlet...
      6/11/01 11:23 ... Servlet Specification Level: 2.2                   com.ibm.servlet.engine.Servlet...
      6/11/01 11:23 ... Supported JSP Specification Level: 1.1             com.ibm.servlet.engine.Servlet...
      6/11/01 11:23 ... Session Manager is Configured - Initializing...    com.ibm.servlet.engine.Servlet...
      6/11/01 11:23 ... DYNA0013I: Dynamic servlet caching is disabled     com.ibm.servlet.dynacache.Ca...
      6/11/01 11:23 ... Transport http is listening on port 9080           com.ibm.servlet.engine.http11....
```

Figure 16-19 Admin Console in WAS 4.0 Advanced Edition

The structure is similar to the Web-based console in the Single Server Edition, but the stand alone console includes some more options as the J2C Resources Adapters (for a review on the differences between AE and AEs, see “Differences between the AE and AEs versions” on page 66).

The Console Messages panel displays the tracing messages of the server and of the applications that use the WebSphere Tracing facility (see details about this feature in “Using the WebSphere JRas facility” on page 341).

The *Options* button lets us select the event level (Fatal, Warning or Audit) as well as properties for the log file. If we want to examine a trace, we use the *Details* button, which displays a window with the complete information for the event. An example is shown in Figure 16-20.



```
ID:         1042
Time:       6/11/01 12:41 PM
Type:       WARNING
Node:       23bk55y
Server:     Default Server
Thread ID:  32be8cac
Source:     itso.was4ad.webapp.controller.ControllerServlet
```

```
java.lang.NullPointerException
        at itso.was4ad.pl.AccountPersister.selectAccountsByCus
        at itso.was4ad.usecase.InitialLoad.execute(InitialLoad
        at itso.was4ad.webapp.command.LoginCommand.execute(Log
        at itso.was4ad.webapp.controller.ControllerServlet.per
        at itso.was4ad.webapp.controller.ControllerServlet.doP
        at javax.servlet.http.HttpServlet.service(HttpServlet.
        at javax.servlet.http.HttpServlet.service(HttpServlet.
        at com.ibm.servlet.engine.webapp.StrictServletInstance
        at com.ibm.servlet.engine.webapp.StrictLifecycleServle
```

Figure 16-20 Examining trace details in the AE Admin Console

Installing new applications

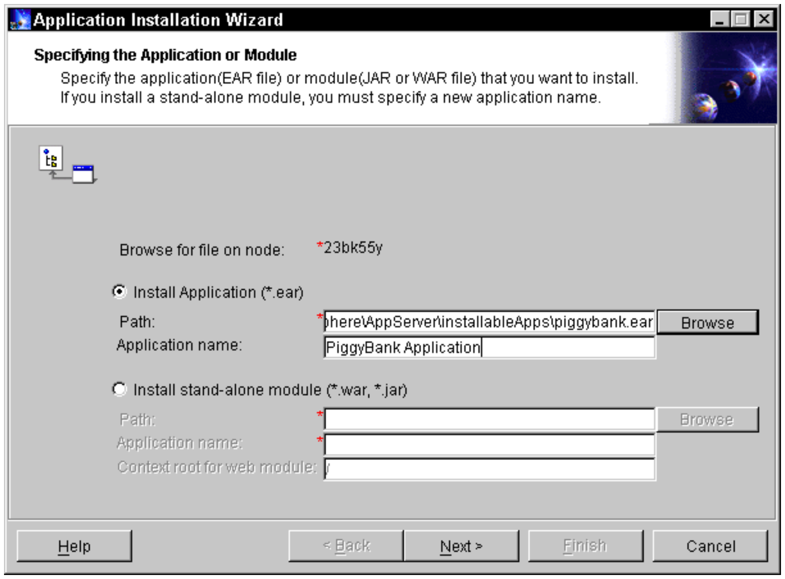
As in the case of the Single Server Edition, we can choose to install a pre-assembled enterprise application (EAR file), or a standalone module, so that the console itself will assemble this module in a new EAR file and deploy its contents if we require it.

Let's consider the following scenario: we have a standard client for EJBs deployed in the application server and we want to unit test the EJBs. In this case we can deploy the EJB module into the server and use the client to test it, and it is not necessary to assemble both modules in an EAR file.

Applications are installed using the *Install Enterprise Application* wizard. This wizard lets us specify binding information for EJBs, security roles and other properties such as the virtual host and the server to which the application will be installed. The last option is the deployment of the module. In general, modules should have been deployed previously, in the assembly phase (see Chapter 15, "Assembling the application" on page 389), but we can redo the process if necessary.

Using the application installation wizard

Figure 16-21 shows the first screen of the wizard.



Specifying the Application or Module

Specify the application(EAR file) or module(JAR or WAR file) that you want to install.
If you install a stand-alone module, you must specify a new application name.

Browse for file on node: *23bk55y

(•) Install Application (*.ear)

Path: *\here\AppServer\installableApps\piggybank.ear [Browse]

Application name: PiggyBank Application

( ) Install stand-alone module (*.war, *.jar)

Path: *

Application name: *

Context root for web module: /

Figure 16-21 Using the application installation wizard

This is a description of the options available when installing an application:

- *Specify role mappings for bean methods*

  This relates to security roles defined for the module. If any mappings have been defined at module level, they appear in the list so that we can modify them.

  When the EJBs added to the enterprise application have role mappings defined for some of their methods but not to all, these other methods appear in the list in case we want to specify security roles for them.

- *Mapping users to roles*

  Roles defined at the assembly phase for the application have to be mapped to users in the system where the application server is running.

- *Mapping EJB RunAs roles to users*

  EJBs have RunAs roles defined to relate with other EJBs, and in this step we have to map these roles to the users of the system that have been mapped to that specific role (done in the previous step).

- *Mapping resource references to resources*

  Resource references used by the application have to be mapped to external resources.

- *Bind EJB to JNDI name*

  This information is read from the EJB deployment descriptor (and, again, we can change it if necessary).

- *Mapping EJB references to EJB*

  EJB references in other modules: for example, in the Web module servlets looking up EJBs directly have to be mapped to the specific EJB they reference (this is not the case in the PiggyBank application).

- *Specifying the default DataSource for an EJB module*

  If we have not created the specific DataSource we want to use with our enterprise application, we can skip this section and create it after. Otherwise, we select it in *Select a Datasource.*

- *Specifying a DataSource for individual CMP beans*

  This is used when different CMP beans use different DataSources, and it overrides the default DataSource specification of the previous screen. If no specific DataSources are specified for CMP beans, they use the default specified for the module.

- *Selecting the virtual hosts for WAR modules*

  We should create the virtual host we want to use with our application before (or, if we want to change it afterwards, we have to edit the configuration using XMLConfig, for example).

  Once the enterprise application has been installed using a virtual host, that information is not visible in the console.

- *Selecting the application server*

  We select the application server for each of the modules integrating our enterprise application. We can select to install the modules on the same server, or each of them in a different server (but we must have the servers created previously).

- *Finish and summary information*

  At this moment, the wizard prompts if the module should be deployed again. As we have said before, in general we would perform the deployment before the installation, so it will not be necessary to redeploy the code.

Uninstalling applications

To uninstall an application from the console, it is required that this application is stopped (though the server where it is running does not have to be stopped to perform this task).

The console offers the possibility to export the configuration of the installed application (as an EAR file with updated binding information), so that, if later we decide to reinstall it again, we can choose this exported file that contains the most recent binding configuration.

Note: In the Single Server Edition, the equivalent task is performed through the SEAppInstall tool, with the `-export` option.

After uninstalling an application, the files are removed from `..\installedApps`, unlike the case of the AEs.

Setting up resources

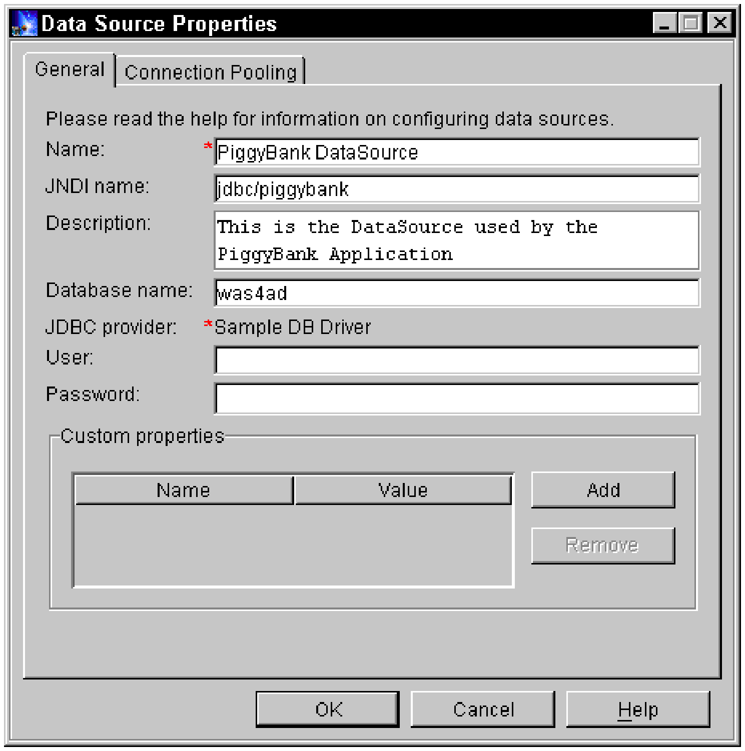
Resource configuration can be done either before or after the application installation. As in the case of the Single Server Edition, the resources are structured as *providers* and *factories*.

We now go through the same example of PiggyBank's DataSource creation to show how to configure resources in the application server's Advanced Edition.

The Admin Console provides the *Create DataSource Wizard* to perform this task. It allows us to create a DataSource based on an existing JDBC driver, or either install a new driver for the new DataSource.

In case we do not want to use the wizard, we can create the DataSource and/or JDBC driver directly under the *Resources* folder.

Figure 16-22 shows the properties of a new DataSource.



Please read the help for information on configuring data sources.

Name:	*PiggyBank DataSource
JNDI name:	jdbc/piggybank
Description:	This is the DataSource used by the PiggyBank Application
Database name:	was4ad
JDBC provider:	*Sample DB Driver
User:	
Password:	

Figure 16-22 Creating a new DataSource

- If we create a new driver, we have to install it in the appropriate node(s).

  To do this, we select the driver under the *JDBC Providers* folder, go to *Nodes* and click on *Install New*. In the next panel, we select the node in which the driver will be installed, and the JAR file containing the driver's class.

- Create a new virtual host (if necessary); this will be done prior to the application installation.

  Select the *Virtual Host* folder in the navigation pane, right-click on it and select *New*. Then, enter the name of the new virtual host, as well as the needed aliases, and click on *OK* to complete the process.

Web server plugin

Web server plugins allow separation between the Web server machine and the application server machine, and this configuration provides more flexibility in the architecture of the global solution.

Prior to WebSphere 4.0, the Web server plugin communicated with the application server using an IBM proprietary protocol, OSE. To solve several problems due to this configuration, two transport plugins have been implemented, HTTP and OSE. They communicate with the backend application server over HTTP and can communicate securely (HTTPS) if needed.

When installing WebSphere Application Server Version 4.0, you are prompted to specify which plugin to use, so that it will be configured in the Web server configuration files. Figure 16-23 shows the entries added to the `httpd.conf` file for the IBM HTTP Server (IHS).

```
LoadModule ibm_app_server_http_module full/path/to/module
AddModule mod_app_server_http.c (Optional)
WebSpherePluginConfig full/path/to/config
```

Figure 16-23 Configuration of the HTTP server plugin for IHS

The HTTP transport plugin

The configuration of the HTTP plugin is stored in a single file (Figure 16-24):

```
D:\WebSphere\AppServer\config\plugin-cfg.xml
```

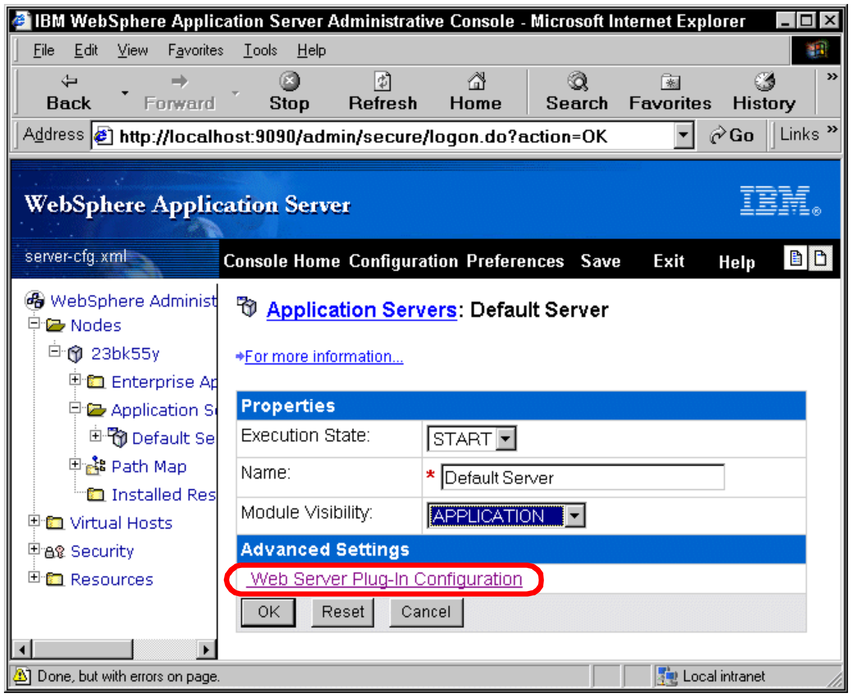
```
<?xml version="1.0"?>
<Config>
   <Log LogLevel="Inform" Name="D:\Websphere\Appserver\logs\native.log"/>
   <VirtualHostGroup Name="default_host">
      <VirtualHost Name="*:80"/>
      <VirtualHost Name="*:9080"/>
   </VirtualHostGroup>
   <ServerGroup Name="Default Server">
      <Server Name="Default Server" SessionID="991347369197">
         <Transport Hostname="*" Port="9080" Protocol="http"/>
      </Server>
   </ServerGroup>
      <UriGroup Name="PiggyBank webapp_URIs">
         <Uri Name="/"/>
      </UriGroup>
      <Route ServerGroup="Default Server" UriGroup="PiggyBank webapp_URIs"
       VirtualHostGroup="default_host"/>
</Config>
```

Figure 16-24 HTTP plugin configuration file: an example

The application server regenerates the configuration after any changes that affect communication with the Web server, for example when adding a new application server, changing components in the Web applications, defining new virtual hosts, and so forth.

Single Server Edition

For the Single Server Edition, manual regeneration is necessary. When using the Web console, it displays a warning message advising to regenerate the plugin configuration. This can be done through the application server's property panel (Figure 16-25).


IBM WebSphere Application Server Administrative Console - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites History
Address http://localhost:9090/admin/secure/logon.do?action=OK Go Links
WebSphere Application Server
server-cfg.xml
Console Home Configuration Preferences Save Exit Help
WebSphere Administ
Nodes
23bk55y
Enterprise Ap
Application S
Default Se
Path Map
Installed Res
Virtual Hosts
Security
Resources
Done, but with errors on page.
Local intranet

Application Servers: Default Server

For more information...

Properties	
Execution State:	START
Name:	* Default Server
Module Visibility:	APPLICATION

Advanced Settings

Web Server Plug-In Configuration

OK Reset Cancel

Figure 16-25 Regenerating the plugin configuration in the AEs console

This link leads us to the Web Server Plug-in configuration page, where we can regenerate the configuration file by simply clicking on the button.

Advanced Edition

In the Advanced Edition, manual regeneration is also possible through the administrative console.

Command line plugin generation

For both editions, there is a command line program that regenerates the configuration of the plugin:

```
D:\WebSphere\AppServer\bin\GenPluginCfg.bat
```

However, its usage is different for both versions.

Single Server Edition

Figure 16-26 shows the plugin generator for the Single Server Edition.

```
D:\WebSphereSSE\AppServer\bin>genplugincfg -help
IBM WebSphere Application Server Standard Edition, Release 4.0
Plugin Configuration Generator, Version 1.0
Copyright IBM Corp., 1997-2001

Usage: Use one of the following commands

        java com.ibm.websphere.plugincfg.tool.SEGeneratePluginCfg
            -configFile <server configuration file>
           [-outputFile  <directory to write the config file to>]
           [-nodeName <name of node>]
           [-serverName <name of server>]
```

Figure 16-26 WebSphere plugin generator, AEs

Because only one node can exist in the Single Server Edition, it has to match the administration node (so this option is redundant).

The key option in the AEs plugin regenerator is the configuration file of the server. Because no administrative database is used, the configuration is stored in XML files in the `..\config` directory.

Advanced Edition

Figure 16-27 shows the plugin generator for the Advanced Edition.

```
D:\WebSphere\AppServer\bin>genplugincfg -help
Usage: java com.ibm.websphere.plugincfg.tool.AEGeneratePluginCfg
        -serverRoot <Product Install Directory>
        -adminNodeName <Administration Server Node Name>
        -nodeName <Local Node Name>
        [-nameServicePort <Administration Server Name Service Port>]
        [[-traceString <trace spec>]
        [-traceFile <file name>]
        [-inMemoryTrace <number of entries>]] |
        [-help | /help | -? | /? ]
```

Figure 16-27 WebSphere plugin generator, AE

Manual plugin regeneration is necessary, for example, after installing/uninstalling an application using the command line tools (see Chapter 16, "Deploying to the test environment" on page 431).

Application client resource configuration tool

This is a stand alone tool that allows us to setup resources associated to enterprise applications' standalone clients. To launch the tool, execute

```
D:\WebSphereSSE\AppServer\bin\clientConfig.bat
```

The configuration data is stored in the EAR client application file, and is used by WebSphere to resolve resources bindings at runtime.

The tool interface is shown in Figure 16-28.

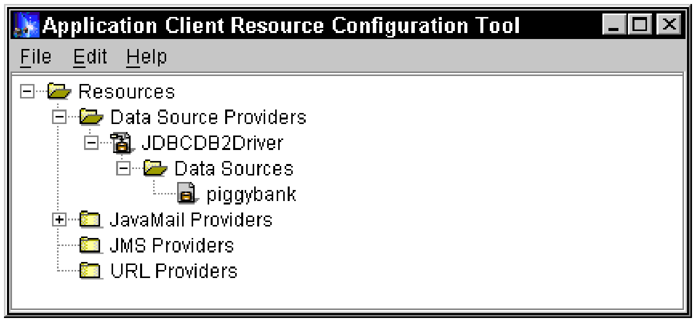


Figure 16-28 Application client resource configuration tool

The resource providers (for example, JDBC drivers) have to be installed in the client's application prior to running the code.

The data for this resource configuration is stored in the EAR file under `\META-INF\client-resource.xmi`.

Figure 16-29 shows an example of the client configuration file for the PiggyBank application, setting up the DataSource associated to a DB2 JDBC driver.

```
<xmi:XMI xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
xmlns:resources="resources.xmi">
   <resources:MailProvider xmi:id="MailProvider_1" name="Default Mail
   Provider" description="IBM JavaMail Implementation">
      <propertySet xmi:id="null_ps"/>
   </resources:MailProvider>
   <resources:ResourceProviderRef xmi:id="ResourceProviderRef_1"
    resourceProvider="MailProvider_1"/>
   <resources:ResourceProviderRef xmi:id="ResourceProviderRef_2"
    classpath="D:\SQLLIB\java\db2java.zip" resourceProvider="JDBCDriver_1"/>
   <resources:JDBCDriver xmi:id="JDBCDriver_1" name="DB2JdbcDriver"
    description="DB2 JDBC Driver"
    implementationClassName="COM.ibm.db2.jdbc.DB2ConnectionPoolDataSource"
    urlPrefix="jdbc:db2">
      <factories xsi:type="resources:DataSource" xmi:id="DataSource_1"
       name="PiggyBankDataSource" jndiName="jdbc/piggybank"
       description="PiggyBank DataSource" databaseName="was4ad">
         <propertySet xmi:id="J2EEResourcePropertySet_3">
            <resourceProperties xmi:id="J2EEResourceProperty_3" name="user"
             value=""/>
            <resourceProperties xmi:id="J2EEResourceProperty_4"
             name="password" value="{xor}"/>
         </propertySet>
      </factories>
      <propertySet xmi:id="J2EEResourcePropertySet_2"/>
   </resources:JDBCDriver>
</xmi:XMI>
```

Figure 16-29 Application client resource configuration file

Note: Information related to client resource configuration does not appear in the standard deployment descriptor files (application.xml, etc.), and will not be visible through the Application Assembly Tool. This information is only used by the WebSphere Application Server runtime.

Other tools in the Advanced Edition

In this section we provide a brief introduction to other configuration tools available for WebSphere Application Server 4.0 Advanced Edition. We do not discuss the tools in depth, as they are broad topics in themselves, but we give a basic idea of their purpose and how they fit in the unit testing environment.

XMLConfig

The XMLConfig tool allows to export/import configuration files from and to the WAS repository database in the Advanced Edition. It can be used to setup the configuration manually, so that a single configuration file can be imported into WebSphere, and we avoid having to perform a long list of tasks in the console.

The bat file to execute XMLConfig can be found in:

```
D:\WebSphere\AppServer\bin\xmlconfig.bat
```

The usage of XMLConfig is shown in Figure 16-30.

```
D:\WebSphere\AppServer\bin>xmlconfig -?
Illegal command line:Odd number of arguments specified.

java com.ibm.websphere.xmlconfig.XMLConfig
   { [( -import <xml data file> ) ||
   ( -export <xml output file> [-partial <xml data file>] )]
   -adminNodeName <primary node name>
   [ -nameServiceHost <host name> [ -nameServicePort <port number> ]]
   [-traceString <trace spec> [-traceFile <file name>]]
   [-substitute <"key1=value1[;key2=value2[...]]">]}
   In input xml file, the key(s)  should appear as $key$ for substitution.
```

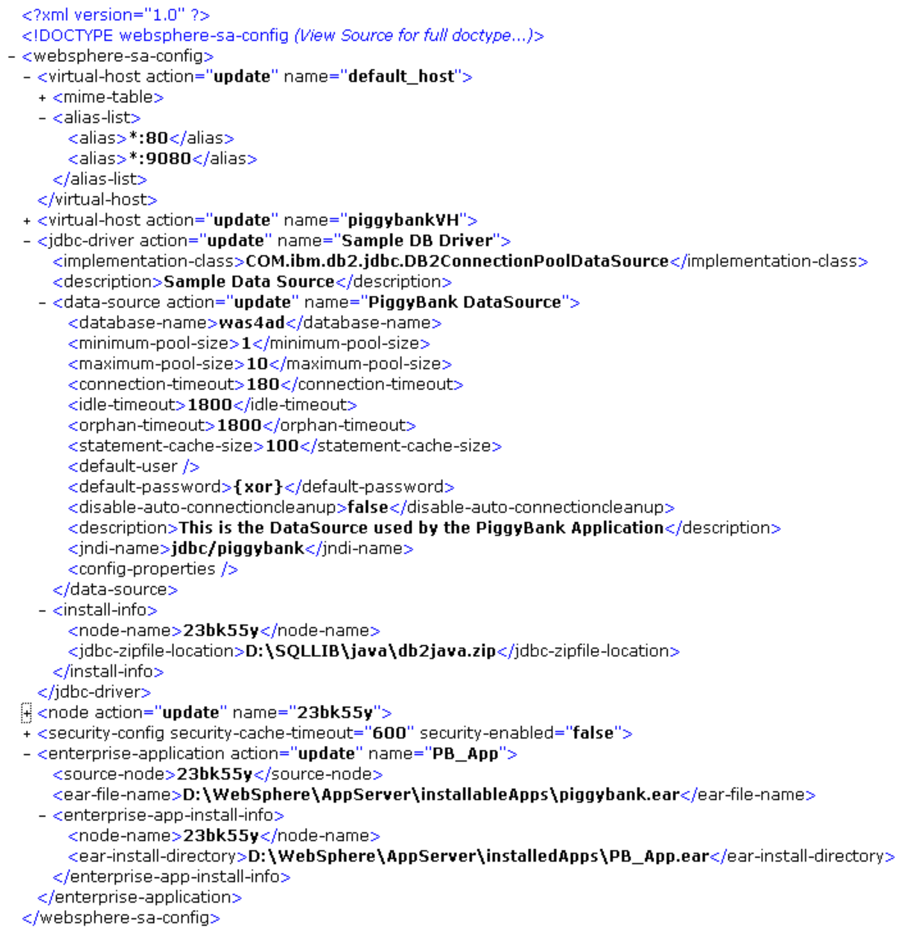
Figure 16-30 Usage of XMLConfig

XMLConfig is also accessible as an option from the Admin Console, in *Files -> Import from XML/Export to XML*.

The XMLConfig tool provides three main features:

- *Full export*—the configuration file generated describes the configuration of the entire domain.
- *Partial export*—the configuration file includes only data from the resources that we specify. The list of this resources to export must be detailed in another XML file, the XML data file.
- *Import*—a new or modified configuration file is imported to the administrative repository.

An example of an exported configuration file is shown in Figure 16-31.



```
  <?xml version="1.0" ?>
  <!DOCTYPE websphere-sa-config (View Source for full doctype...)>
- <websphere-sa-config>
  - <virtual-host action="update" name="default_host">
    + <mime-table>
    - <alias-list>
        <alias>*:80</alias>
        <alias>*:9080</alias>
      </alias-list>
    </virtual-host>
  + <virtual-host action="update" name="piggybankVH">
  - <jdbc-driver action="update" name="Sample DB Driver">
      <implementation-class>COM.ibm.db2.jdbc.DB2ConnectionPoolDataSource</implementation-class>
      <description>Sample Data Source</description>
    - <data-source action="update" name="PiggyBank DataSource">
        <database-name>was4ad</database-name>
        <minimum-pool-size>1</minimum-pool-size>
        <maximum-pool-size>10</maximum-pool-size>
        <connection-timeout>180</connection-timeout>
        <idle-timeout>1800</idle-timeout>
        <orphan-timeout>1800</orphan-timeout>
        <statement-cache-size>100</statement-cache-size>
        <default-user />
        <default-password>{xor}</default-password>
        <disable-auto-connectioncleanup>false</disable-auto-connectioncleanup>
        <description>This is the DataSource used by the PiggyBank Application</description>
        <jndi-name>jdbc/piggybank</jndi-name>
        <config-properties />
      </data-source>
    - <install-info>
        <node-name>23bk55y</node-name>
        <jdbc-zipfile-location>D:\SQLLIB\java\db2java.zip</jdbc-zipfile-location>
      </install-info>
    </jdbc-driver>
  + <node action="update" name="23bk55y">
  + <security-config security-cache-timeout="600" security-enabled="false">
  - <enterprise-application action="update" name="PB_App">
      <source-node>23bk55y</source-node>
      <ear-file-name>D:\WebSphere\AppServer\installableApps\piggybank.ear</ear-file-name>
    - <enterprise-app-install-info>
        <node-name>23bk55y</node-name>
        <ear-install-directory>D:\WebSphere\AppServer\installedApps\PB_App.ear</ear-install-directory>
      </enterprise-app-install-info>
    </enterprise-application>
  </websphere-sa-config>
```

Figure 16-31 Exported XML configuration file

You can edit such a configuration file to:

- Add new virtual hosts using `<virtual-host>` tags
- Add new JDBC drivers and DataSources using `<jdbc-driver>` and `<data-source>`
- Configure the application server properties under the tag `<application-server>`
- Add new enterprise applications by adding `<enterprise-application>` tags.

XMLConfig syntax

In this section we provide a brief introduction to the XMLConfig syntax. Check the product documentation for further details.

In an XMLConfig document, each tag, representing an object type (application-server, virtual-host, etc.), has two attributes:

- *Name*—identifies the specific resource on which the action is taken
- *Action*—controls the behavior of the import/partial export operation.

The list of available actions follows:

  - create
  - update
  - delete
  - locate
  - export
  - start
  - stop
  - stopforrestart
  - restart
  - enable
  - disable
  - createclone
  - associateclone
  - disassociateclone

Some of the actions (*start, stop*) only apply to some resources (for example, you can start an application server, but not a virtual host), and others apply to specific operations (for example, *export* applies only to the *partial export* operation).

WSCP

WSCP stands for WebSphere Control Program, and it is a command-line and scripting interface for administering resources in WebSphere Application Server Advanced Edition.

All console tasks can be performed through WSCP, using commands or scripts. For example, to emulate a wizard, there is no specific command, but a script can be written to perform the same task.

WSCP is based on the tool command language (TCL), an scripting language with a simple and programmable syntax. WSCP extends TCL by providing a set of commands suitable to manipulating WebSphere objects.

Using WSCP is possible to automate administrative tasks or create custom procedures. However, the actions performed via WSCP are not immediately reflected in the console (explicit refresh is required). It is recommended not to perform concurrent tasks both in the console and WSCP, to avoid inconsistencies in the configuration data.

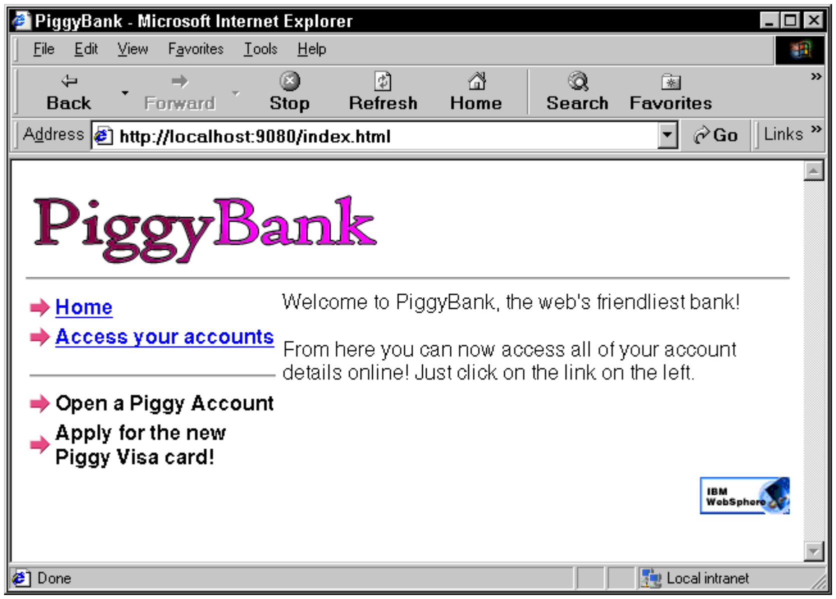
In an unit testing environment, we can consider that the administrative tasks are not going to be many, so we could use only the console, but WSCP provides ways to automate these administrative tasks. For example, if we have to install and uninstall our application often to incorporate fixes or new features, we might consider appropriate to write a WSCP script that automates the task, so that we do not need to run through the application installation wizard every time.

Performing a unit test: executing the application

Once the application has been deployed and installed on the server, it is time to begin with the actual unit test process.

Launching the Web application

Start the application server that contains the Web application to test and launch it in the browser. Figure 16-32 shows the welcome page for the PiggyBank.


PiggyBank - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Forward Stop Refresh Home Search Favorites
Address http://localhost:9080/index.html
Go Links
PiggyBank
Home
Access your accounts
Open a Piggy Account
Apply for the new Piggy Visa card!
IBM WebSphere
Done
Local intranet

Welcome to PiggyBank, the web's friendliest bank!

From here you can now access all of your account details online! Just click on the link on the left.

Figure 16-32 PiggyBank Web application welcome page

Launching the client application with the launchClient tool

WebSphere 4.0 provides a command line tool for launching a client application installed as an EAR file (it can be an independent EAR file with the external references appropriately configured, or it can be the global EAR file, as we have done with PiggyBank):

```
D:\WebSphere\AppServer\bin\launchClient.bat
```

The usage of the command is shown in Figure 16-33.

```
D:\WebSphereSSE\AppServer\bin>launchClient
IBM WebSphere Application Server, Release 4.0
J2EE Application Client Tool, Version 1.0
Copyright IBM Corp., 1997-2001

WSCL0012I: Processing command line arguments.
Usage: launchClient [<userapp.ear> | -help | -?] [-CC<name>=<value>] [app args]
where:
    <userapp.ear>       = The path/name of the .ear file containing the
                             client application.
    -help, -?           = print this help message

where the -CC properties are for use by the Client Container:
    -CCverbose          = <true|false> Use this option to display additional
                             informational messages.
    -CCjar              = The path/name of the jar file within the ear
                             file that contains the application you wish to
                             launch. This argument is only necessary when
                             you have multiple client application jar files
                             in your ear file.
    -CCBootstrapHost    = The name of the host server you wish to connect to
                             initially. Format: your.server.ofchoice.com
    -CCBootstrapPort    = The server port number to use.
    -CCtrace            = <true|false> Use this option to have WebSphere
                             write debug trace information to a file. You may
                             need this information when reporting a problem to
                             IBM Service.
    -CCtracefile        = The name of the file to write trace information
                             to.
    -CCpropfile         = Name of a Properties file containing launchClient
                             specific properties.
    -CCinitonly         = <true|false> This option is intended for ActiveX
                             applications to initialize the Application Client
                             runtime without launching the client application.

where "app args" are for use by the client application and are ignored by
WebSphere.
```

Figure 16-33 Usage of the launchClient tool

Figure 16-34 shows the output of the launchClient tool for the PiggyBank command line application client.

```
D:\WebSphereSSE\AppServer\bin>launchClient ..\installableapps\piggybank.ear
IBM WebSphere Application Server, Release 4.0
J2EE Application Client Tool, Version 1.0
Copyright IBM Corp., 1997-2001

WSCL0012I: Processing command line arguments.
WSCL0013I: Initializing the J2EE Application Client Environment.
WSCL0035I: Initialization of the J2EE Application Client Environment has
completed.
WSCL0014I: Invoking the Application Client class
itso.was4ad.client.StandaloneClient

Standalone Client Menu

Select an option:

        (1)     Create a new customer
        (2)     Create a new account
        (3)     Display Customer
        (4)     Display Account
        (5)     Transfer Money
        (6)     Cash a check
        (0)     Exit

Choice (1-6,0)
```

Figure 16-34 Launching the PiggyBank application client

17

Debugging the application

In this chapter we discuss how to debug applications using the following tools:

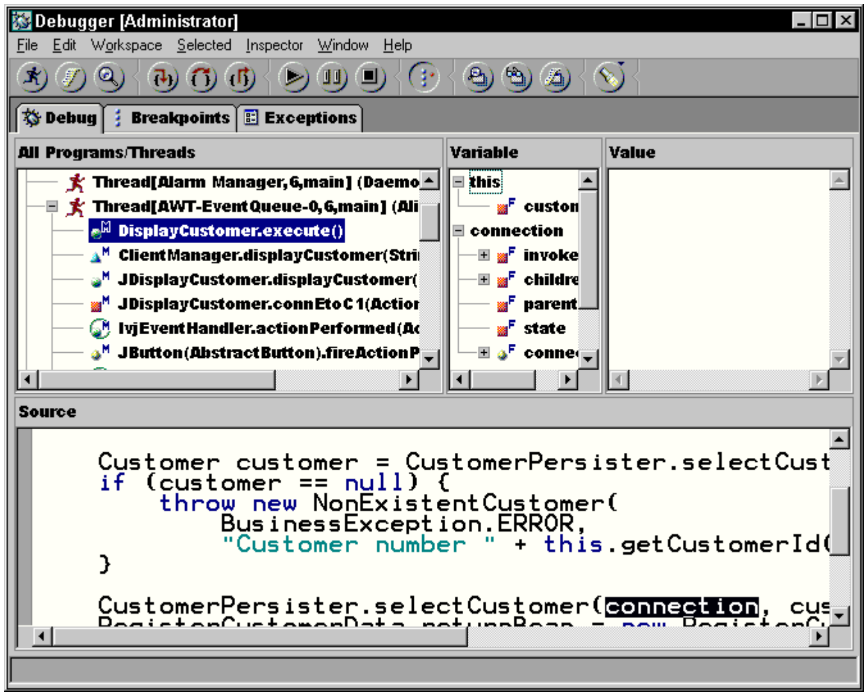
- VisualAge for Java Version 4.0
- WebSphere Application Server Version 4.0 and the Distributed Debugger and Object Level Trace (OLT) Version 9.1

Version 9.1 of the Distributed Debugger and OLT software is required to work with WebSphere Version 4.0—this is the first version that supports the Version 1.3 Java virtual machine (JVM) used by this release of WebSphere.

Debugging with VisualAge for Java Version 4.0

The VisualAge for Java debugger is a powerful feature that allows the developer to debug code within the IDE (debug servlets, JSPs, general Java applications and even classes outside VisualAge for Java). It is possible to debug either complete classes or code snippets (in the scrapbook). In this section we describe the main features of the debugger and how to make the most of the tool when developing WebSphere applications.

First, let's take a look at the user interface of the debugger (Figure 17-1).



Debugger [Administrator]
File Edit Workspace Selected Inspector Window Help
Debug
Breakpoints
Exceptions
All Programs/Threads
Thread[Alarm Manager,6,main] (Daemo
Thread[AWT-EventQueue-0,6,main] (Ali
DisplayCustomer.execute()
ClientManager.displayCustomer(Stri
JDisplayCustomer.displayCustomer(
JDisplayCustomer.connEtoC1(Actior
IvjEventHandler.actionPerformed(Ac
JButton(AbstractButton).fireActionP
Variable
this
custon
connection
invoke
childre
parent
state
conne
Value
Source

```
    Customer customer = CustomerPersister.selectCust
    if (customer == null) {
        throw new NonExistentCustomer(
            BusinessException.ERROR,
            "Customer number " + this.getCustomerId(
    }

    CustomerPersister.selectCustomer(connection, cus
```

Figure 17-1 VisualAge for Java debugger interface

The main pane, *Debug*, shows several frames with different information:

- Threads—a list of all the currently running threads is shown. Threads are grouped by application.
- Variable—this panel shows a list of the variables visible at the current debugging stage.
- Value—displays the values of the variables. From this panel, we can change these values to alter the flow of execution.

- Source—shows the source code for the class and method being currently debugged.

Variables and their values be detached into a separate window by selecting *Window -> Visible Variables*. This action provides more space to inspect variables with a large amount of data or applications with many variables.

The *Breakpoints* pane shows a list of the breakpoints and displays the source code where the breakpoint has been set.

The *Exceptions* pane shows a list of exceptions recognized by the VisualAge for Java runtime. The debugger stops when the selected exception is thrown, both if it is an uncaught exception or if it is managed later in a `catch` or `finally` block.

With the debugger it is possible to fix code errors while debugging, without having to restart the application. Changes to a method mean that only that method is recompiled (incremental compilation).

Working with breakpoints

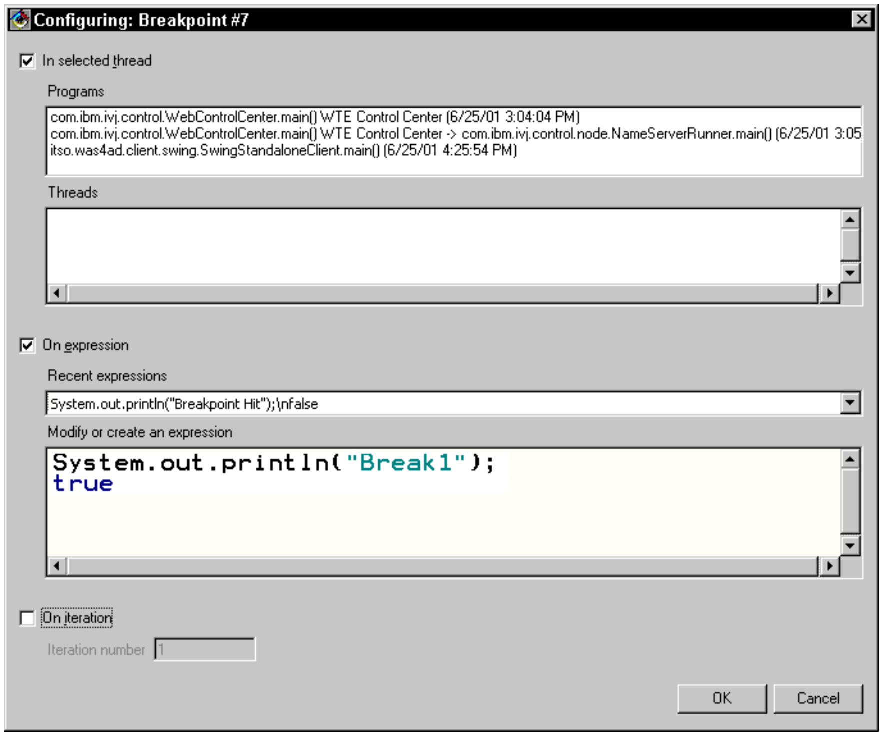
Setting a breakpoint in VisualAge for Java is as easy as making a double click at the left side of a code line in the Workbench or in any source code view (not in the scrapbook, we describe how to debug code fragments in “Debugging code snippets” on page 474), or hitting `Ctrl+B` when selecting a statement.

The debugger provides several options to manage the breakpoints globally: it is possible to enable/disable or clear all of them at a time.

Breakpoints can also be configured to have a segment of code attached to them, so that, every time the breakpoint is reached, this code is executed.

This and other settings can be configured through the context menu *Modify* (focusing on the breakpoint) for an existing breakpoint, or by selecting *Breakpoint* from the context menu of any source line for a new breakpoint.

The configuration window is shown in Figure 17-2.


Configuring: Breakpoint #7

[✓] In selected thread

Programs

```
com.ibm.ivj.control.WebControlCenter.main() WTE Control Center (6/25/01 3:04:04 PM)
com.ibm.ivj.control.WebControlCenter.main() WTE Control Center -> com.ibm.ivj.control.node.NameServerRunner.main() (6/25/01 3:05
itso.was4ad.client.swing.SwingStandaloneClient.main() (6/25/01 4:25:54 PM)
```

Threads

[✓] On expression

Recent expressions: System.out.println("Breakpoint Hit");\nfalse

Modify or create an expression

```
System.out.println("Break1");
true
```

[ ] On iteration

Iteration number: 1

Figure 17-2 Configuring breakpoints in VisualAge for Java

Here we can configure a conditional breakpoint (that is, one that only opens the debugger if a certain condition is evaluated to the boolean value `true`), using the *On expression* option. For conditions involving looping parameters, we can use the *On iteration* option, and set up the iteration number where we want the debugger to be triggered.

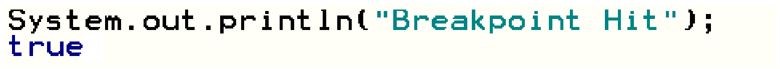
To configure a conditional breakpoint, we can either select one of the expressions from the list (that contains the 10 last expressions used by breakpoints), or write a code segment in the window, so that we can make the breakpoint print a message in the console when it is triggered, for example.

The package `com.ibm.uvm.tools`, that is part of the IBM Java implementation, contains the `DebugSupport` class used by the debugger, that we may also use for our conditional breakpoints. In particular, the `bell` method can be useful in certain circumstances; it causes a "beep" sound to be played by the computer when the breakpoint is hit.

To print messages on a conditional breakpoint, we can use simple `System.out.println` statements, though the expression window allows us to write any code.

To make the debugger open conditionally when hitting a breakpoint, we use the variable `true`. If we use the variable `false`, any message we include as output are displayed in the console, but the debugger does not halt at the breakpoint. Instead of using directly these variables, we can use conditional expressions that, when evaluated to a true or false boolean value, causes the debugger to halt at the breakpoint or not.

An example of the usage is shown in Figure 17-3.



```
System.out.println("Breakpoint Hit");
true
```

Figure 17-3 Working with conditional breakpoints

The *Modify* window also allows us to set breakpoints in specific threads. If we are running several threads of the same program, we might not want to have the same breakpoint set in all of them. This window lets us select the thread for which the breakpoint will be enabled.

When the debugger hits a breakpoint, we have several options to continue with the program execution (toolbar buttons or *Selected* menu):

Step into—the debugger steps into the next executable statement and halts at its first line.

Step over—the debugger steps to the next statement.

Run to return—skip to the end of the method and go back to the point where it was called (or stepped into).

Resume—the program continues to be executed until it hits the next breakpoint (or until the end if there are none).

Suspend—halt a running thread (this option is not available when a thread is stopped at a breakpoint).

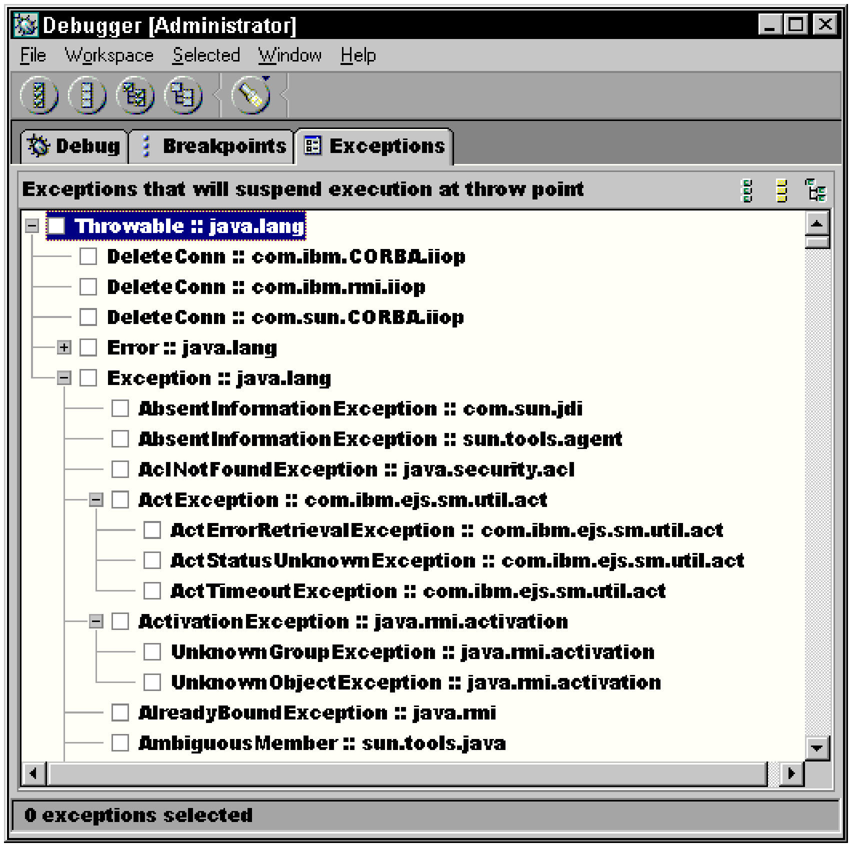
Terminate—the execution of the thread is terminated (there is the option to stop all the threads for the current program).

Run to cursor—the debugger runs the program up to the point where the cursor has been placed. After setting the cursor, select *Selected -> Run to Cursor*.

Exceptions

The debugger stops at uncaught exceptions, but sometimes we might want to stop at caught exceptions to inspect them and find out about their origin. The *Exceptions* pane of the debugger lets us select exceptions by class, package or hierarchy, so that if any of the selected is thrown, the debugger stops at it regardless of whether our program catches it or not.

Figure 17-4 shows the *Exceptions* pane when sorting the exceptions by hierarchy.



Exceptions that will suspend execution at throw point

```
Throwable :: java.lang
    DeleteConn :: com.ibm.CORBA.iiop
    DeleteConn :: com.ibm.rmi.iiop
    DeleteConn :: com.sun.CORBA.iiop
    Error :: java.lang
    Exception :: java.lang
        AbsentInformationException :: com.sun.jdi
        AbsentInformationException :: sun.tools.agent
        AclNotFoundException :: java.security.acl
        ActException :: com.ibm.ejs.sm.util.act
            ActErrorRetrievalException :: com.ibm.ejs.sm.util.act
            ActStatusUnknownException :: com.ibm.ejs.sm.util.act
            ActTimeoutException :: com.ibm.ejs.sm.util.act
        ActivationException :: java.rmi.activation
            UnknownGroupException :: java.rmi.activation
            UnknownObjectException :: java.rmi.activation
        AlreadyBoundException :: java.rmi
        AmbiguousMember :: sun.tools.java
```

0 exceptions selected

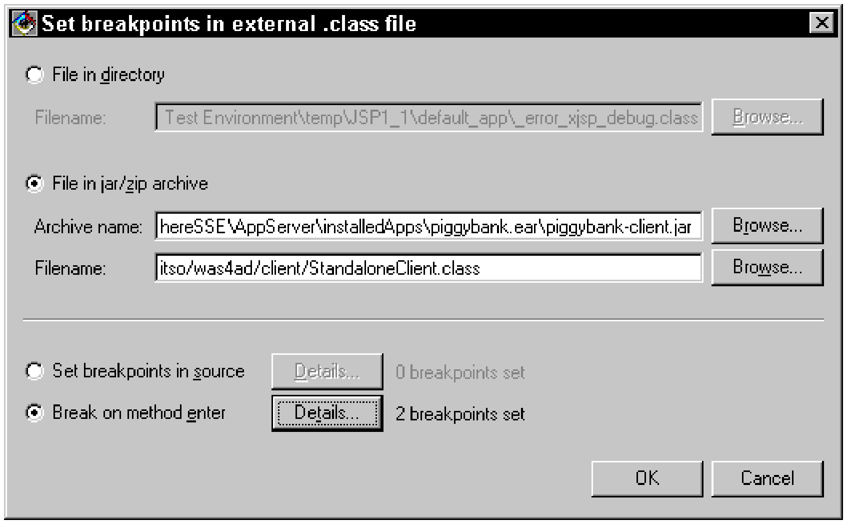
Figure 17-4 Listing exceptions by hierarchy

Debugging external classes

The debugger lets the developer set breakpoints in external classes outside of the repository (but not in classes running in other JVM; to debug a external Java program or file, it has to run inside VisualAge for Java).

We set breakpoints in external class files through the *Breakpoints* pane of the debugger, selecting *Methods -> External .class file breakpoints*.

We can select the file from a directory or from a JAR/ZIP file. We can set breakpoints when entering the methods of the selected class, but if we want to set breakpoints anywhere in the code, we place the corresponding source file in the same location as the class file (in the same directory or JAR/ZIP file) or we specify the location of the source in the debug class path (Figure 17-5).


Set breakpoints in external .class file

File in directory

Filename: Test Environment\temp\JSP1_1\default_app_error_xjsp_debug.class

[✓ File in jar/zip archive]

Archive name: hereSSE\AppServer\installedApps\piggybank.ear\piggybank-client.jar

Filename: itso/was4ad/client/StandaloneClient.class

Set breakpoints in source — 0 breakpoints set

[✓ Break on method enter] — 2 breakpoints set

Figure 17-5 Setting breakpoints in external class files

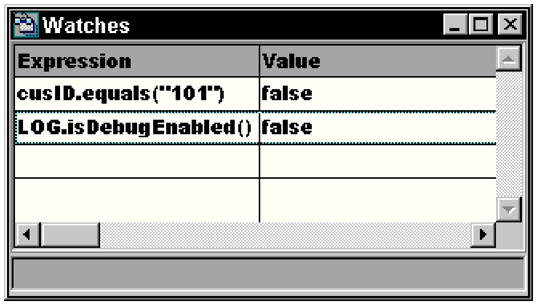
External breakpoints cannot be conditional.

Inspecting data

We can inspect the values of variables through the *Variables* and *Values* panes of the debugger. The values shown are at the current execution point, so if we step through the code, we can see the values change. We can also change the values through the *Value* pane.

The option *Inspect* acts in the same way, though it opens a new window containing all the variable's data.

A *watch* is an Java expression that the debugger evaluates each time a stack trace is shown (when hitting a breakpoint or encountering an exception as well as when advancing the stack frame with the step functions). By using the *Watches* panel, we can have a number of expressions defined so that every time we execute the program, we do not have to inspect the results manually (we get them automatically in the *Watches* panel). An example is shown in Figure 17-6.



Expression	Value
cusID.equals("101")	false
LOG.isDebugEnabled()	false

Figure 17-6 Watches panel

Debugging code snippets

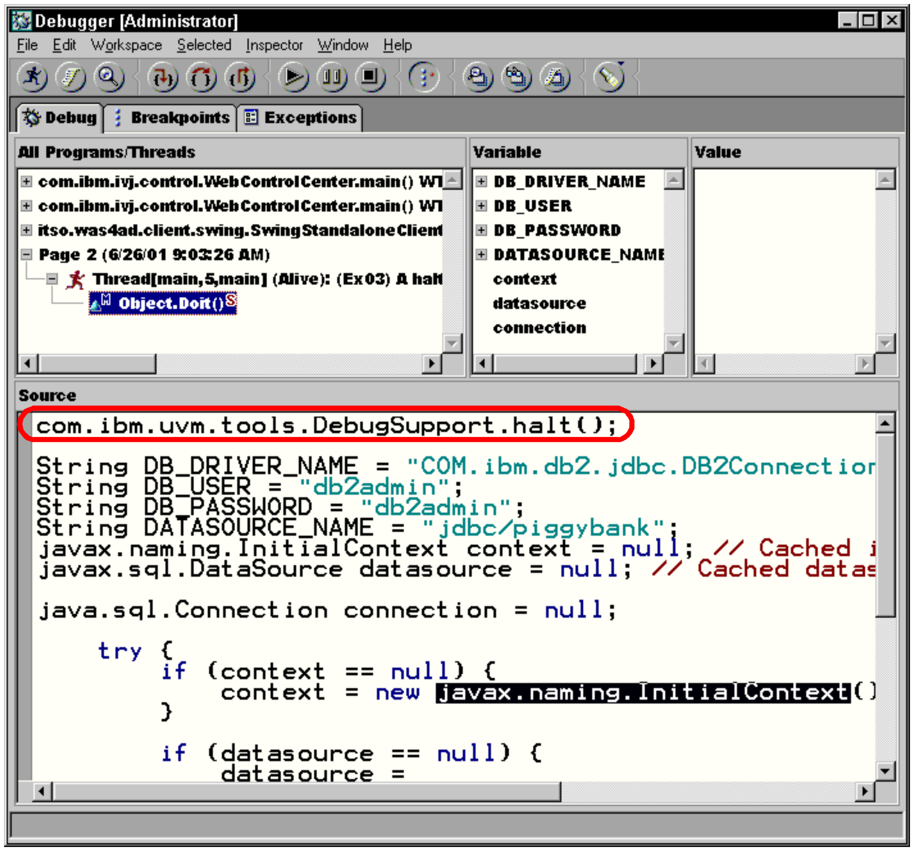
Sometimes, we might have pieces of code we want to debug, and we do not want to run through the whole application. The *scrapbook* in combination with the debugger provides us with the tool we have to perform this task.

To run a piece of code in the scrapbook, we simply copy it in a new page, select the fragment of code we want to run and hit the *Run* button. If we also want to debug it, we click on the *Debug* button.

It is not possible to set breakpoints in the scrapbook, but we can debug the code fragment step by step adding the following sentence at the beginning of the page:

```
com.ibm.uvm.tools.DebugSupport.halt();
```

The execution of the `halt` method causes the debugger to be opened at the first sentence of the page. Then we can perform the debugging of the code fragment as any other Java code. The thread associated to the snippet we are executing is shown in the debugger with the name of the scrapbook page that contains it (Figure 17-7).



```
com.ibm.ivj.control.WebControlCenter.main() WT
com.ibm.ivj.control.WebControlCenter.main() WT
itso.was4ad.client.swing.SwingStandaloneClient
Page 2 (6/26/01 9:03:26 AM)
    Thread[main,5,main] (Alive): (Ex03) A halt
        Object.DoIt()
```

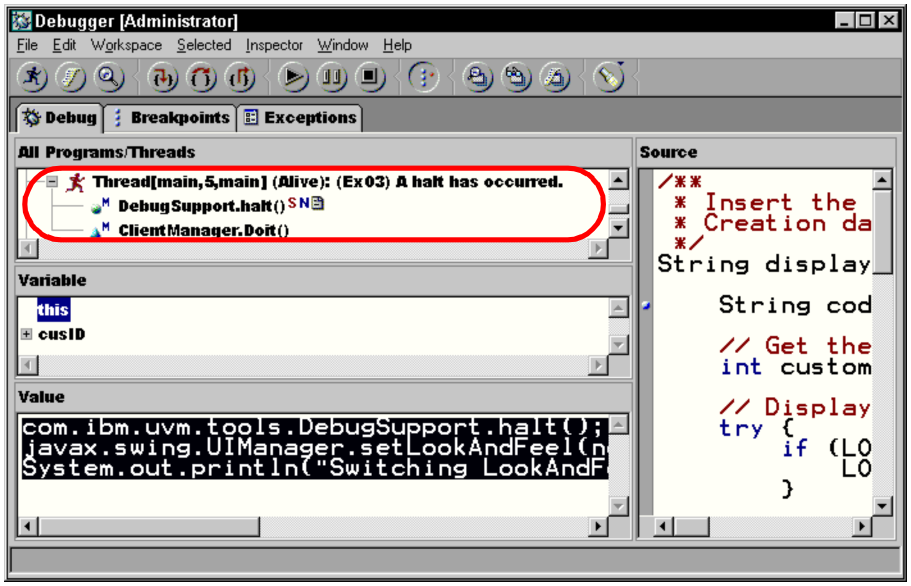
```
DB_DRIVER_NAME
DB_USER
DB_PASSWORD
DATASOURCE_NAME
context
datasource
connection
```

```
com.ibm.uvm.tools.DebugSupport.halt();

String DB_DRIVER_NAME = "COM.ibm.db2.jdbc.DB2Connectior
String DB_USER = "db2admin";
String DB_PASSWORD = "db2admin";
String DATASOURCE_NAME = "jdbc/piggybank";
javax.naming.InitialContext context = null; // Cached i
javax.sql.DataSource datasource = null; // Cached datas

java.sql.Connection connection = null;

    try {
        if (context == null) {
            context = new javax.naming.InitialContext()
        }

        if (datasource == null) {
            datasource =
```

Figure 17-7 Debugging code snippets in the scrapbook

The Debugger has another feature that allows us debugging code fragments: the *Evaluation Area*. It lets us run any Java code against the currently selected object (as if we wrote the code in the *Source* pane, or in the *Value* pane if we have selected the variable `this`). We can use this feature for example to set or change parameters related to the current object (for example, the look and feel for a Swing object, or others as the class path, environment variables).

The way to execute code in the *Evaluation Area* is similar to the scrapbook: we select the piece of code and run it (but this time it is related to an object in the debugger window, for example, `this`). If we want to trigger the debugger before executing the code snippet, we include a call to `DebugSupport.halt()`. Again, the thread corresponding to the snippet is displayed in the debugger's *Threads* window (Figure 17-8).



```
Thread[main,5,main] (Alive): (Ex03) A halt has occurred.
    DebugSupport.halt()
    ClientManager.Doit()
```

```
this
cusID
```

```
com.ibm.uvm.tools.DebugSupport.halt();
javax.swing.UIManager.setLookAndFeel(n
System.out.println("Switching LookAndF
```

```
/**
 * Insert the
 * Creation da
 */
String display

    String cod

    // Get the
    int custom

    // Display
    try {
        if (LO
            LO
        }
```

Figure 17-8 Executing code in the evaluation area

Tip: Select *Window -> Flip Orientation* to get this look of the debugger.

Debugging with the Distributed Debugger and OLT

The Distributed Debugger and Object Level Trace (OLT) are two parts of a single product we can use to trace and debug code running in WebSphere. The major benefits in using the Distributed Debugger and OLT are:

- We can debug code running on platforms other than Windows.
- We can debug code running in WebSphere Application Server processes on both local and remote systems.
- We can collate and display debugging information from multiple processes in the same window.

The Distributed Debugger and OLT installation code is included on the product CD with WebSphere Application Server, VisualAge for Java and WebSphere Studio. In the examples in this chapter we installed the Distributed Debugger and OLT code in `D:\IBMDebug`.

In the following sections we describe how to configure WebSphere to enable debugging support, and how to use the OLT and Distributed Debugger tools.

Enabling debugging support in WebSphere Application Server

To debug a WebSphere application, it is necessary to attach the debugger to the application server Java Virtual Machine. To debug a standalone application, we can either launch the program containing the main method, or attach the debugger to the Java Virtual Machine of the running application.

To use OLT in combination with the Distributed Debugger, we select either the *Debug only* or the *Trace and Debug* modes.

Both OLT and the Distributed Debugger allow us to trace and debug local or remote applications—the remote machine does not have to be running the same operating system as the client. We now describe how to set up the debugger to connect to a running Java Virtual Machine.

To use the Distributed Debugger in a WebSphere environment, it is necessary to set up the application server for this purpose.

- The Java classes to be debugged must be compiled with the `-g` option (to include debugging information).
- The Java Virtual Machine must be started in debug mode.
- OLT and the Distributed Debugger have to be enabled in WebSphere.

The first task has to be performed prior to the code deployment and installation in the server, but the other two are specified as administration options. Because JSPs are compiled in the application server, specifying the debug option for the JVM assures that the compiled files contain the debug information needed.

Virtual machine debug options

There are a number of options we can specify to control the debugging behavior of a virtual machine. We use these options in the following examples to configure the WebSphere JVM to allow us to debug our applications.

Attention: These extended JVM options are specific to the virtual machine implementation—in this case we describe the IBM virtual machine that runs on Windows. You may find that options available on other platforms differ. WebSphere on Windows sets these options when you enable the debugger.

Let’s describe the JVM options we use in more detail:

-Xdebug	Enables debugging
-Xnoagent	Disables the old debug agent (`sun.tools.debug`, see more details below)
-Djava.compiler	The *NONE* option disables the just in time (JIT) compiler

-Xrunjdwp	Loads the implementation of the Java Debug Wire Protocol (JDWP)

The Java 2 Platform incorporate a new debugging support with the Java Platform Debug Architecture (JPDA). JDWP is the interface that communicates between the debugger's JVM and the application's JVM.

The options for **-Xrunjdwp** that we use in this example are the following:

transport	Name of the transport used to connect to the debugger's JVM (`dt_socket` in our case).
server	The default value *n* indicates that the server will attach to the debug engine at the specified address (or at the automatically generated address if this parameter is not specified). Specifying a value of *y* means that the server will listen for a debug engine to attach at the address specified.
suspend	The value of *y* (default) indicates that the JVM is suspended before the main class is loaded. The user then sets a deferred breakpoint where to stop, and runs the application to that breakpoint. The value of *n* indicates that the JVM can proceed with the execution of the program before the debug engine is attached.
address	The port number where the debug engine will attach to the server. More information about JDPA and its associated interfaces is available at Sun's Web site: `http://java.sun.com/products/jpda/readme.html`

Enabling support in Advanced Edition

We use the WebSphere AE administrative console to enable OLT and the Distributed Debugger and set JVM debug options. These settings are all controlled at the application server level using the application server's properties. To change these settings, open the console and use the tree view in the panel on the left side to navigate to and select the application server whose properties you want to change (Figure 17-9).

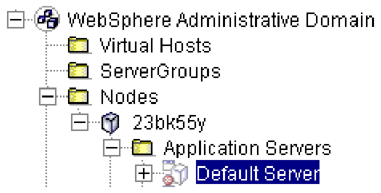
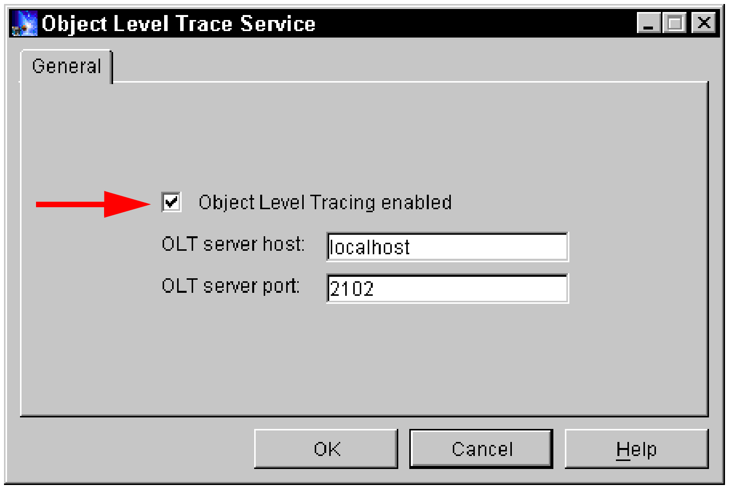


Figure 17-9 Navigating to the Default Server application server in the console

Enabling OLT

Select the application server in the console and click the *Services* tab. Highlight *Object Level Trace* in the list of services, and click *Edit Properties* (Figure 17-10).



☑ Object Level Tracing enabled

OLT server host: localhost

OLT server port: 2102

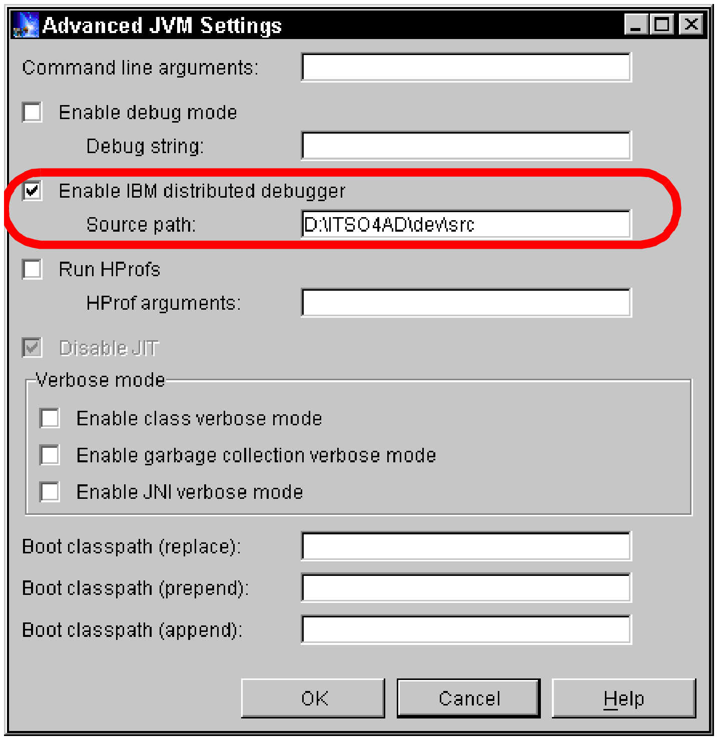
Figure 17-10 Enabling OLT in AE

In the *Object Level Trace Service* dialog check the box to enable OLT. Enter the host name of the machine where the OLT server is running in the *OLT server host* field—typically this is the machine where you collect the trace and run the OLT and debugger GUI. The default port number of 2102 should normally not be changed, unless the port is already being used for another purpose.

Enabling the Distributed Debugger

To enable the Distributed Debugger to work in conjunction with OLT we must enable it from the administrative console. Select the *JVM Settings* tab and click *Advanced Settings* to open the dialog shown in Figure 17-11.

- Check the *Enable IBM distributed debugger* box highlighted in the figure. You may also specify a location on the server for the debugger to search for the Java source files for the code to debug. If the source files are not found on the server, the debugger GUI prompts you for a local location when you debug the code—this is described in “Source pane” on page 491.
- If you have not already enabled OLT, a dialog pop-up asks if you want to enable OLT—OLT must be enabled if you check the *Enable IBM distributed debugger* box.


Advanced JVM Settings

Command line arguments:

☐ Enable debug mode

Debug string:

☑ Enable IBM distributed debugger

Source path: D:\ITSO4AD\dev\src

☐ Run HProfs

HProf arguments:

☑ Disable JIT

Verbose mode

☐ Enable class verbose mode

☐ Enable garbage collection verbose mode

☐ Enable JNI verbose mode

Boot classpath (replace):

Boot classpath (prepend):

Boot classpath (append):

Figure 17-11 Enabling the Distributed Debugger to work with OLT in AE

Setting virtual machine options

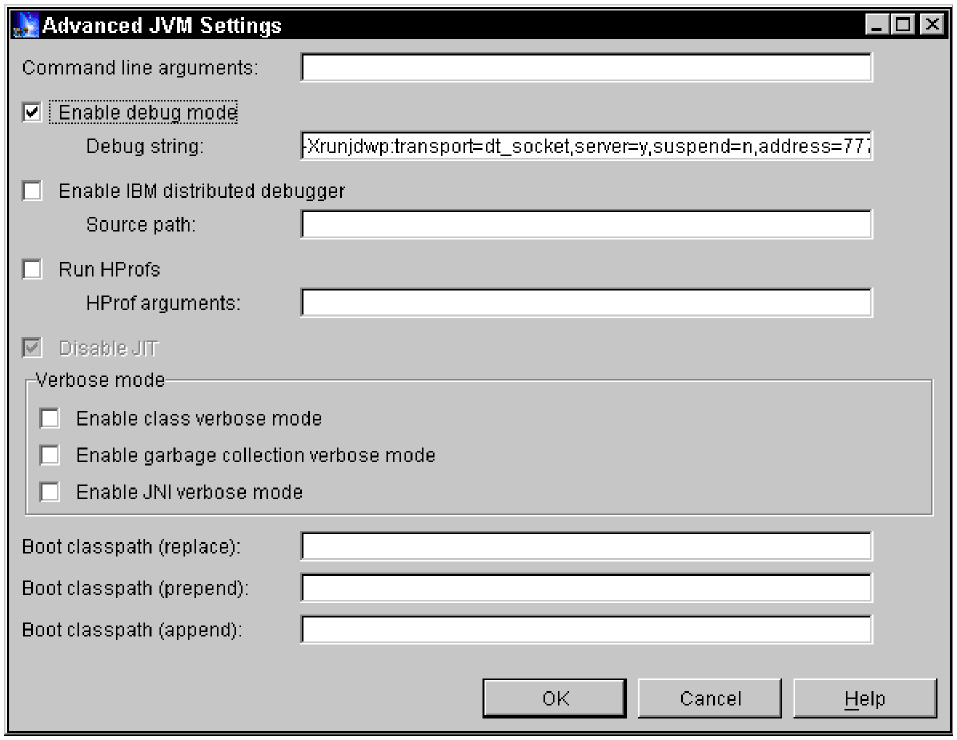
If you want to use specific options with the Distributed Debugger without OLT, or if you want to use another JDPA debugger against the applications running in WebSphere, you can explicitly specify the debugging options for the virtual machine.

- Select the application server in the console and click the *JVM Settings* tab. Click *Advanced Settings* to open a new dialog (Figure 17-12).
- Check the *Enable debug mode* box—this causes the console to add these arguments to the JVM options:

```
-Xdebug -Xnoagent
```

- Add to these options to tell the JVM to listen for debugging requests on port 7777, by entering the following in the *Debug string* field:

```
-Xrunjdwp:transport=dt_socket,server=y,suspend=n,address=7777
```



Command line arguments:

[✓] Enable debug mode

Debug string: -Xrunjdwp:transport=dt_socket,server=y,suspend=n,address=77

[ ] Enable IBM distributed debugger

Source path:

[ ] Run HProfs

HProf arguments:

[✓] Disable JIT

Verbose mode

[ ] Enable class verbose mode

[ ] Enable garbage collection verbose mode

[ ] Enable JNI verbose mode

Boot classpath (replace):

Boot classpath (prepend):

Boot classpath (append):

Figure 17-12 Entering JVM debugging options in AE

- After starting the application server with these options, you have to start the debugger and attach to the JVM (see "Attaching the debugger to the JVM" on page 496).

For the users of the IBM distributed debugger we recommend to follow the steps in "Enabling the Distributed Debugger" on page 479. We did not test debugging with another JDPA debugger.

Enabling support in Advanced Edition, Single Server

The AEs Web administration tool provides fields that mirror the settings already described for the AE console. AEs also allows us to specify debug and OLT parameters on the command line when we start the application server, and we found this to be the most convenient way to work.

Figure 17-13 shows the command line options available when issuing the AEs `startServer` command. The options related to OLT and debugging are highlighted.

```
D:\work\src>startServer -usage

IBM WebSphere Application Server
Application Server Launcher
Copyright (C) IBM Corporation, 2001

Usage:
  startServer
    [ -configFile <config file name> ]
    [ -nodeName <node name> ]
    [ -serverName <server name> ]
    [ ( -oltEnable | -oltenable ) ]
    [ ( -oltPort | -port ) <OLT port number> ]
    [ ( -oltHost | -host ) <OLT hostname> ]
    [ ( -debugEnable | -debug ) ]
    [ -jdwpPort <JDWP port number> ]
    [ ( -debugSource | -debug_cp ) <JDWP souce path> ]
    [ ( -serverTrace | -traceString ) <server trace string> ]
    [ ( -serverTraceFile | -traceFile ) <server trace file name> ]
    [ -script [<script file name>]
      [ -targetOS <operating system name> ] ]
    [ -noExecute ]
    [ -usage ]
    [ -help ]
    [ -verbose ]
```

Figure 17-13 AEs startServer command line options

- We enable OLT in AEs by specifying either the `-oltEnable` or `-oltenable` option (the two are equivalent). We can also choose to specify options to change the OLT server host, which defaults to the local host if not specified, and the OLT server port, which defaults to 2102 if not supplied.
- We enable debugging by specifying the `-debugEnable` or `-debug` options—again, the two are equivalent. We can also specify a port for the JVM to listen on; the default port number is 7777.
- The command we issue to start our AEs application server, enabling both OLT and debug is:

  ```
  startServer -oltEnable -debugEnable   [ -host hostnameUserInterface ]
  ```

This would be the typical usage of this command, accepting the defaults for both OLT and debugging, with the OLT server code running on the same machine as AEs—likely to be the case if you are a developer debugging code on your own machine. Specify the `-host` option if you run the OLT and debugger user interface on another machine.

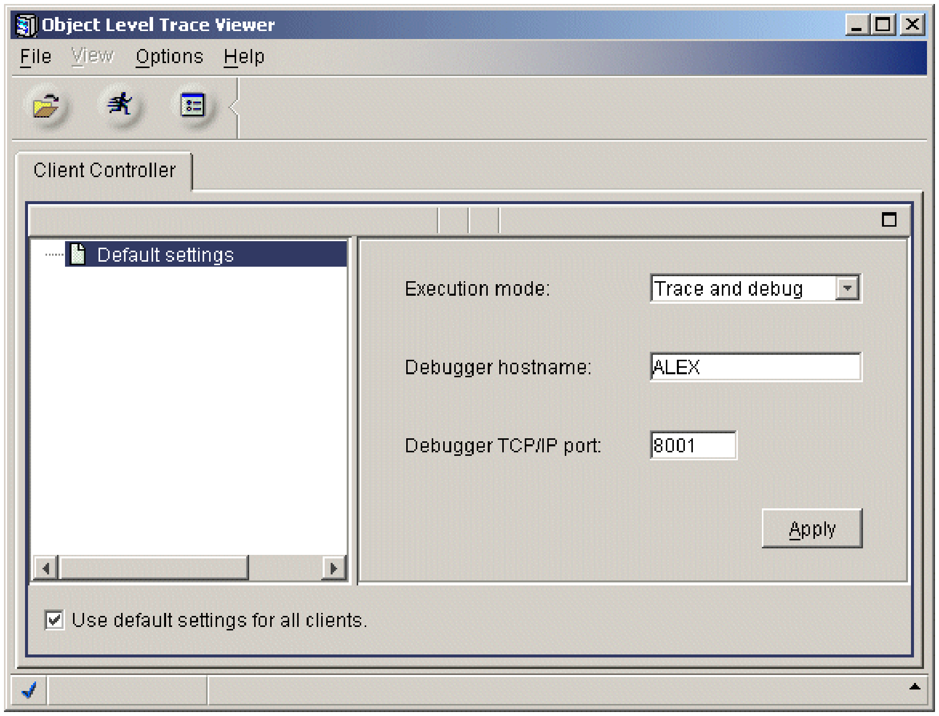
Using Object Level Trace

Object Level Trace (OLT) is a distributed object tracing facility that allows us to see the relations between servlets and JSPs involved in a Web application request process, and in combination with the Distributed Debugger, allows to debug distributed code.

The OLT consists of a GUI client and a trace server. When we configure WebSphere to use OLT, we specify the hostname and port for the OLT server. WebSphere connects to the OLT server and sends it application trace events. More than one application server can connect to the same OLT server at the same time, enabling the OLT server to collate trace from multiple servers.

The OLT GUI also connects to the server—it takes the collected trace events and presents them in a graphical display for analysis. We start the OLT GUI (Figure 17-14) and the local OLT server using this command:

```
D:\IBMDebug\bin\olt.exe
```



Execution mode: Trace and debug

Debugger hostname: ALEX

Debugger TCP/IP port: 8001

[✓] Use default settings for all clients.

Figure 17-14 Initial OLT GUI

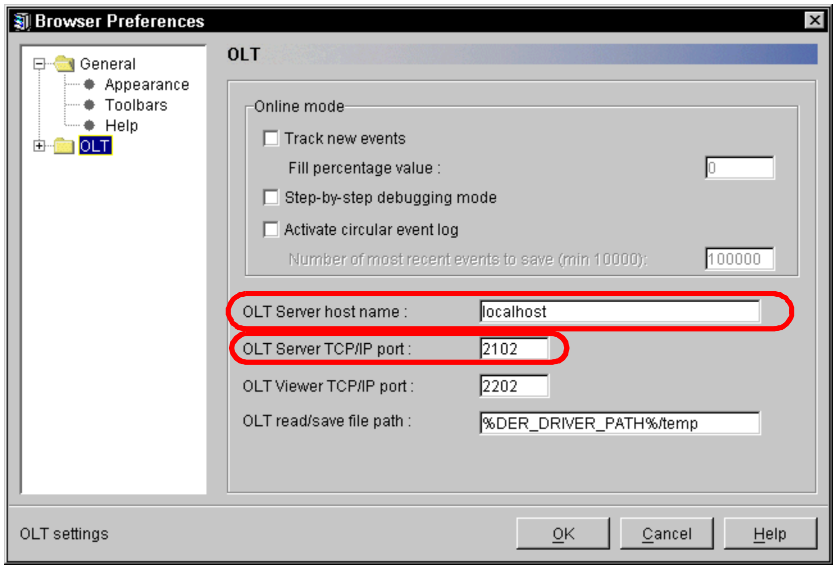
Configuring OLT

From the initial screen we can select one of the following trace and debugging modes:

- Trace only
- Debug only
- Trace and debug
- No trace and debug

The debug modes operate OLT in conjunction with the Distributed Debugger, so that breakpoints set in OLT trigger the debugger, and we can see at the same time the flow of the application and the possible code errors.

Figure 17-15 shows the OLT client settings (*File -> Preferences*) when the client is installed in the same machine as the OLT server. It is possible to have distributed configurations where the application server, the application client, the OLT server and the OLT client are running in different machines, though it is more common to have the OLT server and client in the same machine.



Online mode

- [ ] Track new events

Fill percentage value : 0

- [ ] Step-by-step debugging mode
- [ ] Activate circular event log

Number of most recent events to save (min 10000): 100000

OLT Server host name : localhost

OLT Server TCP/IP port : 2102

OLT Viewer TCP/IP port : 2202

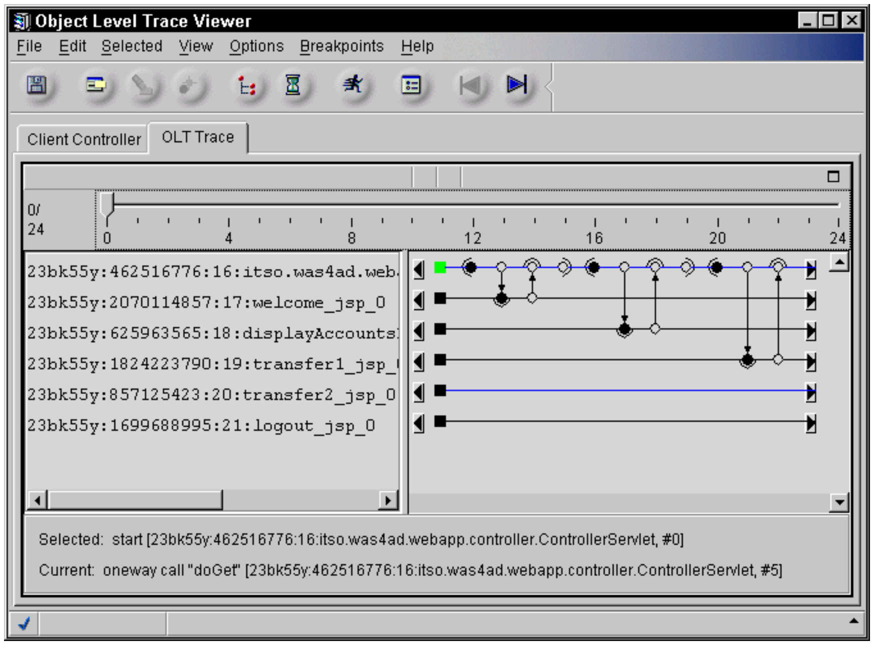
OLT read/save file path : %DER_DRIVER_PATH%/temp

OLT settings

Figure 17-15 OLT settings

Reading tracing information

The graphical interface of the OLT client is shown in Figure 17-16.


Object Level Trace Viewer
File Edit Selected View Options Breakpoints Help
Client Controller
OLT Trace
0/
24
0 4 8 12 16 20 24

23bk55y:462516776:16:itso.was4ad.web
23bk55y:2070114857:17:welcome_jsp_0
23bk55y:625963565:18:displayAccounts
23bk55y:1824223790:19:transfer1_jsp_
23bk55y:857125423:20:transfer2_jsp_0
23bk55y:1699688995:21:logout_jsp_0

Selected: start [23bk55y:462516776:16:itso.was4ad.webapp.controller.ControllerServlet, #0]

Current: oneway call "doGet" [23bk55y:462516776:16:itso.was4ad.webapp.controller.ControllerServlet, #5]

Figure 17-16 OLT graphical interface

Each node represents a method call, and the arrows indicate the flow of the program.

A trace line is an horizontal line connecting events running under the same execution thread. Each trace line represents one component (servlet, JSP, client application) running in the application server. Events can be defined as method calls, return from method calls or start or end of a process.

The main elements of the trace window are shown in Figure 17-17. For more information about the graphical representation, check the product documentation.

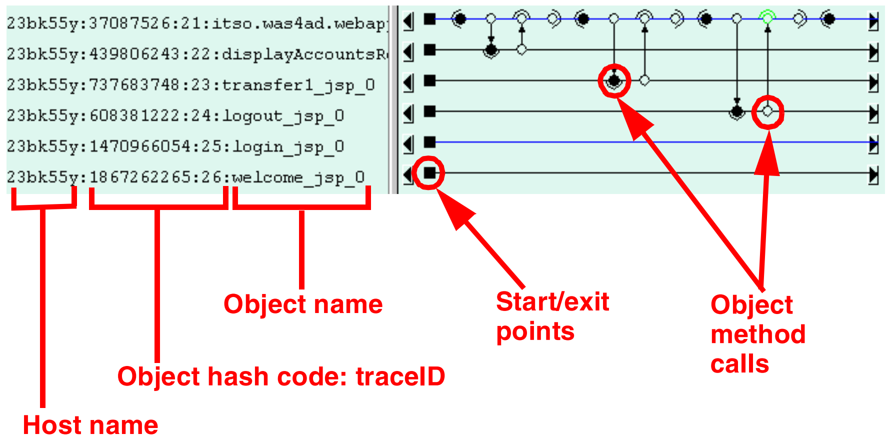

23bk55y:37087526:21:itso.was4ad.webap
23bk55y:439806243:22:displayAccountsR
23bk55y:737683748:23:transfer1_jsp_0
23bk55y:608381222:24:logout_jsp_0
23bk55y:1470966054:25:login_jsp_0
23bk55y:1867262265:26:welcome_jsp_0
Object name
Start/exit points
Object method calls
Object hash code: traceID
Host name

Figure 17-17 Reading OLT traces

The status lines at the bottom of the window provide information identifying the location of the *selected* event and the *current* event:

- The selected event (highlighted green by default) is the last event clicked with the left mouse button.
- The current event is the event that the mouse pointer is currently positioned over. As you move your pointer, the current event changes.

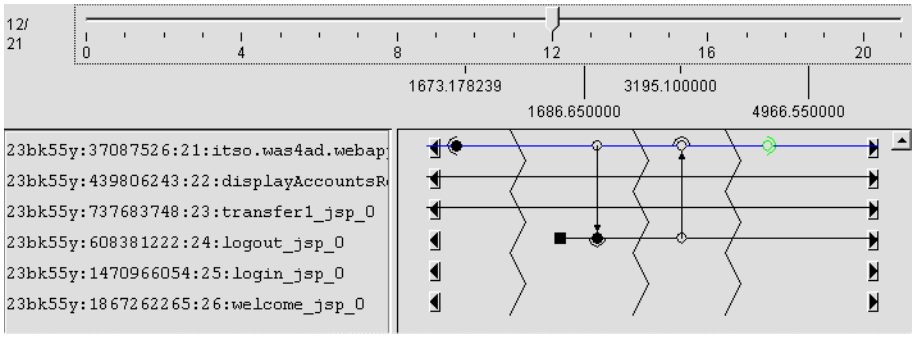
An example of the status line display is shown in Figure 17-18.

```
Selected: receive reply "_jspService" [23bk55y:37087526:21:itso.was4ad.webapp.
                                       controller.ControllerServlet,#11]
Current: start [23bk55y:73768374:23:transfer1_jsp_0,#0]
```

Figure 17-18 Status line information

We can choose two displays for the tracing:

- *Partial order display* (default, shown in Figure 17-17)—events are not always drawn in the sequence in which they happened. The goal of this display is to represent as many events as possible in the trace screen. However, the casual relationships between elements are preserved.
- *Real time display*—shows the real order for the sequence of events and timing information (Figure 17-19).

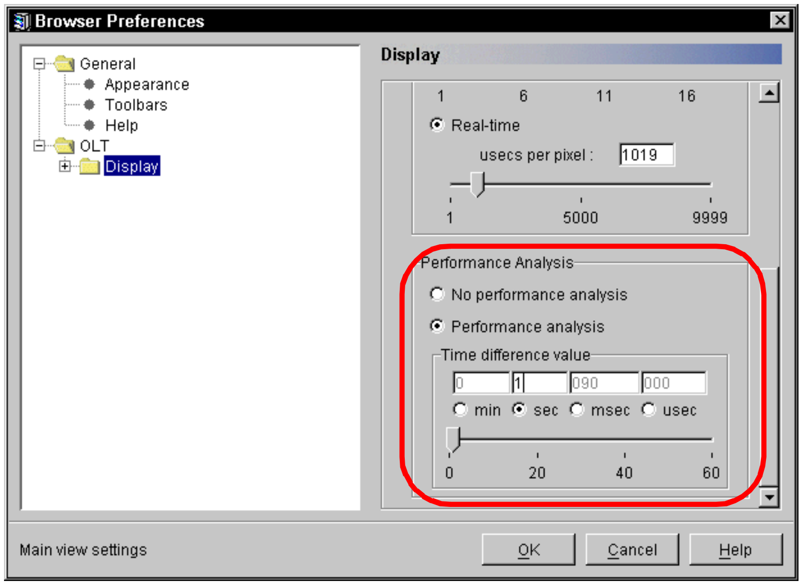


```
23bk55y:37087526:21:itso.was4ad.webap
23bk55y:439806243:22:displayAccountsR
23bk55y:737683748:23:transfer1_jsp_0
23bk55y:608381222:24:logout_jsp_0
23bk55y:1470966054:25:login_jsp_0
23bk55y:1867262265:26:welcome_jsp_0
```

Figure 17-19 Traces in real time display

The display properties can be changed through *File -> Preferences -> OLT -> Display*. This window also allow us to enable *Performance Analysis*.

Performance Analysis lets us monitor the time between any two calls. By setting up a maximum time, we can control if the call takes more time than the established maximum, so that we can detect possible bottlenecks or slowness in the functions. Figure 17-20 shows how to set the time intervals.



(●) Real-time

usecs per pixel : 1019

Performance Analysis

( ) No performance analysis

(●) Performance analysis

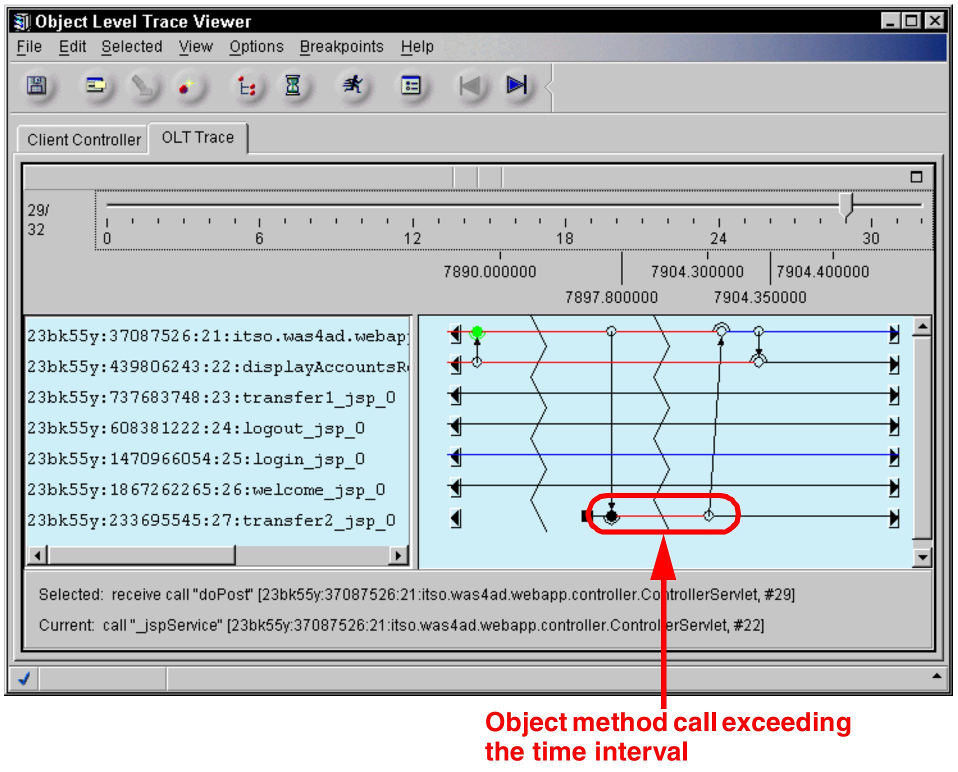
Time difference value

0 | 1 | 090 | 000

( ) min (●) sec ( ) msec ( ) usec

Figure 17-20 Configuring the performance analysis parameters

Calls exceeding the defined time interval are shown in red (default color) in the traces window (Figure 17-21).


Object Level Trace Viewer
File Edit Selected View Options Breakpoints Help
Client Controller
OLT Trace
29/
32
0
6
12
18
24
30
7890.000000
7897.800000
7904.300000
7904.350000
7904.400000
Object method call exceeding the time interval

```
23bk55y:37087526:21:itso.was4ad.webap
23bk55y:439806243:22:displayAccountsR
23bk55y:737683748:23:transfer1_jsp_0
23bk55y:608381222:24:logout_jsp_0
23bk55y:1470966054:25:login_jsp_0
23bk55y:1867262265:26:welcome_jsp_0
23bk55y:233695545:27:transfer2_jsp_0
```

```
Selected: receive call "doPost" [23bk55y:37087526:21:itso.was4ad.webapp.controller.ControllerServlet, #29]
Current: call "_jspService" [23bk55y:37087526:21:itso.was4ad.webapp.controller.ControllerServlet, #22]
```

Figure 17-21 Viewing the performance analysis data in the traces window

It is possible to save the trace to a file if we want to analyze it afterwards, or transport it to another machine.

Tracing local and remote applications

Tracing a local application means that the OLT server and client are installed on the same machine as the application server. Then, the host name specified in the application server when enabling OLT can be simply `localhost`.

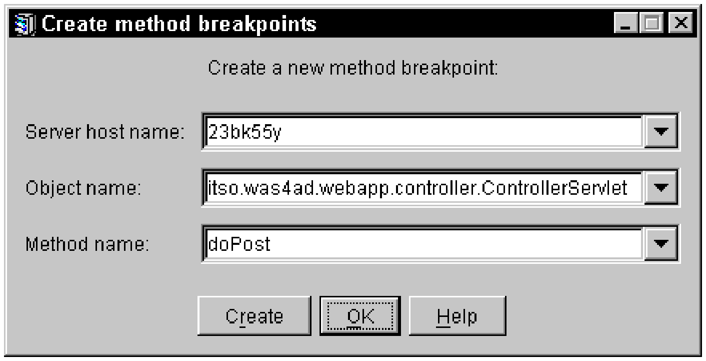
For remote applications, we specify the fully qualified network name of the machine where the OLT server is installed.

See details about setting up the application server for OLT and debugging in section “Enabling debugging support in WebSphere Application Server” on page 477.

Setting breakpoints

Breakpoints can be set for any debuggable event. Debuggable events in the OLT graphical interface are represented by filled circles (see Figure 17-17 on page 486).

The *Breakpoints* menu also provides the option to set breakpoints in any object method that is part of the trace. Figure 17-22 shows an example of how to set a method breakpoint using this feature.



Create a new method breakpoint:

Server host name: 23bk55y

Object name: itso.was4ad.webapp.controller.ControllerServlet

Method name: doPost

Figure 17-22 Creating method breakpoints in OLT

The option *List Method Breakpoints* allows us to enable, disable or delete any of the breakpoints we have set.

With OLT we cannot use more types of breakpoints than the method type, but more options are available when using the Distributed Debugger (which is more appropriate to perform in-depth debugging).

Setting breakpoints in OLT would be done before launching the Distributed Debugger, where we perform the “hard” debugging work.

If we select the *Step-by-step Debugging Mode*, OLT stops at every debuggable method of the trace, prompting us to enter the method, and then the debugger is automatically launched to connect to the running JVM or process.

Using the Distributed Debugger

The Distributed Debugger is a client/server application that allows to debug programs both locally or through a network connection. It is composed of a server, or *Debug engine*, and a *client* GUI where we can set up breakpoints and control the execution of the debugged applications. Both can be installed in the same or in different machines (we talk more about configurations in the next sections).

It is possible to debug programs in several programming languages (compiled, as C or C++, or interpreted, as Java), though we only focus on Java applications in this book. For more information about other possibilities, check the product documentation.

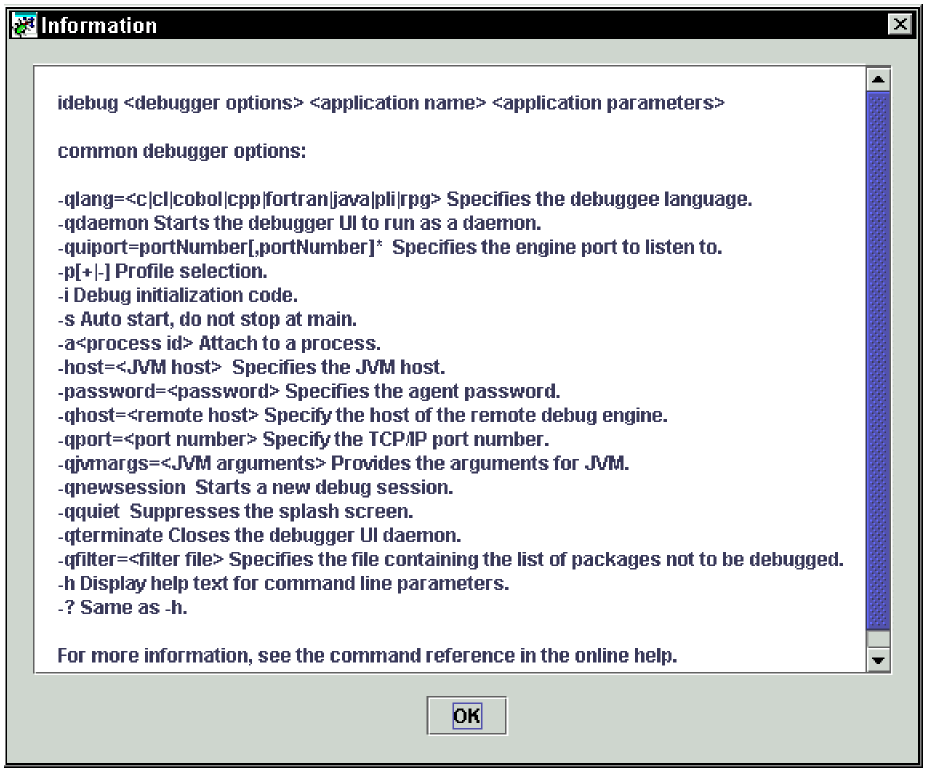
With the Distributed Debugger it is possible to debug different applications simultaneously, with applications located in different systems (locally or over the network). The information displayed for each application depends on the debug engine the client is connected to.

Starting the debugger

The Distributed Debugger program can be launched from:

```
D:\IBMDebug\bin\idebug.exe
```

Several options are available through a command line startup (Figure 17-23) and we provide more details on some of the options in the following sections.

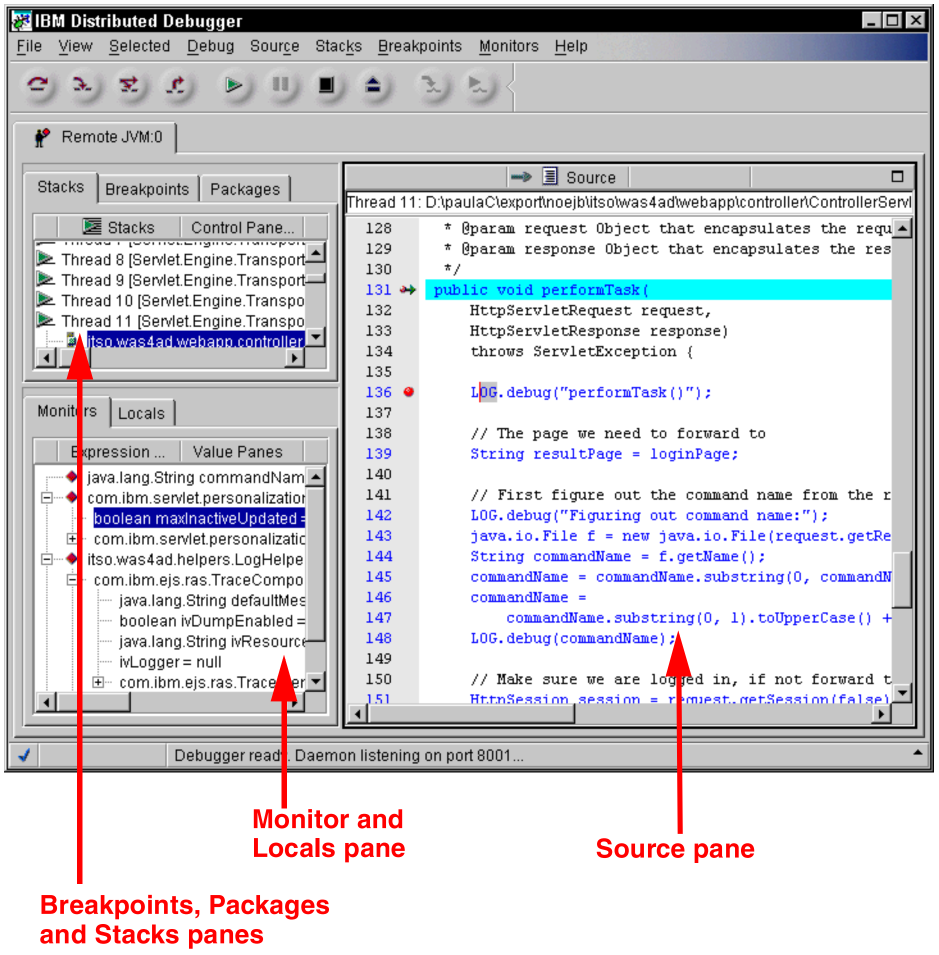


```
idebug <debugger options> <application name> <application parameters>

common debugger options:

-qlang=<c|cl|cobol|cpp|fortran|java|pli|rpg> Specifies the debuggee language.
-qdaemon Starts the debugger UI to run as a daemon.
-quiport=portNumber[,portNumber]*  Specifies the engine port to listen to.
-p[+|-] Profile selection.
-i Debug initialization code.
-s Auto start, do not stop at main.
-a<process id> Attach to a process.
-host=<JVM host>  Specifies the JVM host.
-password=<password> Specifies the agent password.
-qhost=<remote host> Specify the host of the remote debug engine.
-qport=<port number> Specify the TCP/IP port number.
-qjvmargs=<JVM arguments> Provides the arguments for JVM.
-qnewsession  Starts a new debug session.
-qquiet  Suppresses the splash screen.
-qterminate Closes the debugger UI daemon.
-qfilter=<filter file> Specifies the file containing the list of packages not to be debugged.
-h Display help text for command line parameters.
-? Same as -h.

For more information, see the command reference in the online help.
```

Figure 17-23 Idebug command line parameters

The graphical interface

Figure 17-24 shows the graphical user interface of the Distributed Debugger.



IBM Distributed Debugger
File View Selected Debug Source Stacks Breakpoints Monitors Help
Remote JVM:0
Stacks Breakpoints Packages
Stacks Control Pane...
Monitors Locals
Expression ... Value Panes
Source
Monitor and Locals pane
Source pane
Breakpoints, Packages and Stacks panes

```
Thread 8 [Servlet.Engine.Transport
Thread 9 [Servlet.Engine.Transport
Thread 10 [Servlet.Engine.Transpo
Thread 11 [Servlet.Engine.Transpo
  itso.was4ad.webapp.controller
```

```
java.lang.String commandNam
com.ibm.servlet.personalizatior
  boolean maxInactiveUpdated =
  com.ibm.servlet.personalizatic
itso.was4ad.helpers.LogHelpe
  com.ibm.ejs.ras.TraceCompo
    java.lang.String defaultMes
    boolean ivDumpEnabled =
    java.lang.String ivResource
    ivLogger = null
    com.ibm.ejs.ras.TraceLer
```

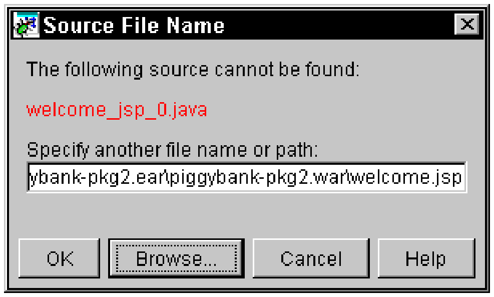
```
Thread 11: D:\paulaC\export\noejb\itso\was4ad\webapp\controller\ControllerServl
     * @param request Object that encapsulates the requ
     * @param response Object that encapsulates the res
     */
    public void performTask(
        HttpServletRequest request,
        HttpServletResponse response)
        throws ServletException {

        LOG.debug("performTask()");

        // The page we need to forward to
        String resultPage = loginPage;

        // First figure out the command name from the r
        LOG.debug("Figuring out command name:");
        java.io.File f = new java.io.File(request.getRe
        String commandName = f.getName();
        commandName = commandName.substring(0, commandN
        commandName =
            commandName.substring(0, 1).toUpperCase() +
        LOG.debug(commandName);

        // Make sure we are logged in, if not forward t
        HttpSession session = request.getSession(false)
```

```
Debugger ready. Daemon listening on port 8001...
```

Figure 17-24 Distributed Debugger user interface

Source pane

The *Source* pane displays the source code for the program being debugged. When we are debugging an application, the debugger searches for the source code and prompts us to specify the location if it cannot find the source code (Figure 17-25).



The following source cannot be found:

welcome_jsp_0.java

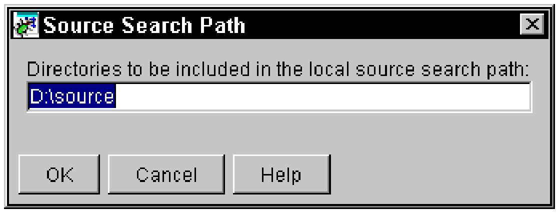
Specify another file name or path:

ybank-pkg2.ear\piggybank-pkg2.war\welcome.jsp

Figure 17-25 Locating source code files

No source code is displayed if we have not compiled our classes including the debugging information.

It is possible to specify to the Distributed Debugger where to look for the source code (Figure 17-26).



Directories to be included in the local source search path:

D:\source

Figure 17-26 Specifying the source search path

You can set the source search path either before you launch the program from the Load Program dialog by clicking on the *Advanced* button, or while you are debugging the program from the *Source* menu.

Tip: You can also specify the debugger source path using the `DER_DBG_PATH` environment variable. If you include this variable in your environment you can avoid having to set the source path each time you restart the debugger. You must set the variable before starting the debugger.

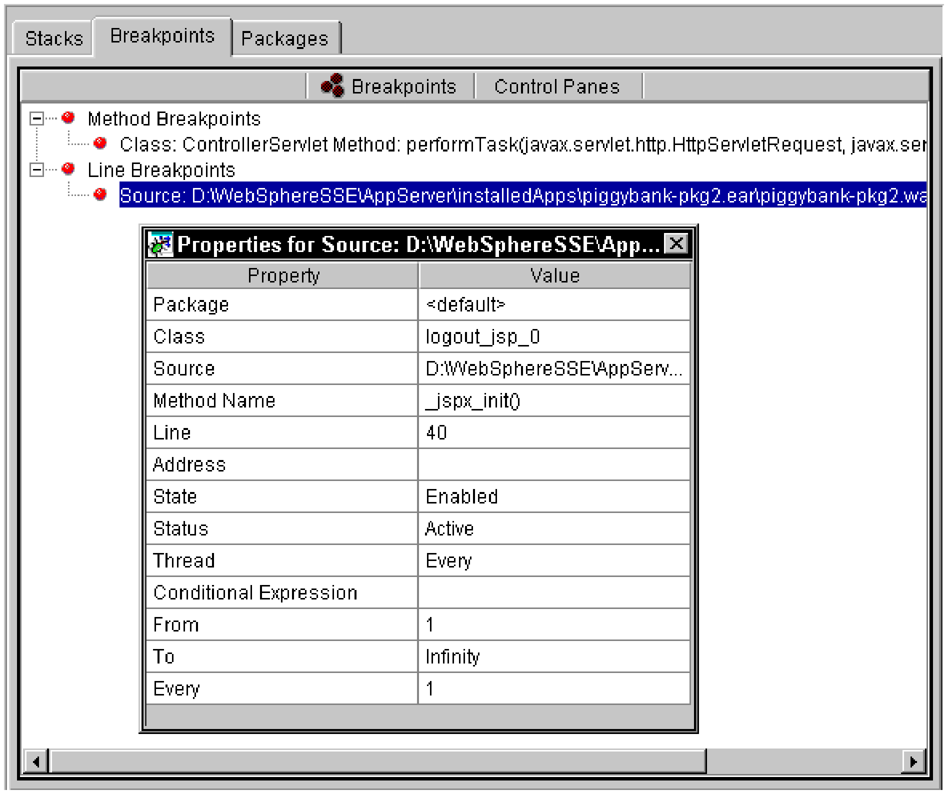
Breakpoints pane

The *Breakpoints* pane displays a view of the breakpoints we have set in the application we are debugging. In this pane we can add new breakpoints or modify the properties of the existing ones.

We can configure three types of breakpoints:

- *Line breakpoints*—triggered before the code in the specified line is executed
- *Method breakpoints*—triggered when the specified method is called
- *Watchpoints*—triggered when class field being monitored is modified (available only for Java if the JVM supports it)

We can also see the properties of each breakpoint: if it is enabled, or for which thread it is enabled (Figure 17-27).



Method Breakpoints
Class: ControllerServlet Method: performTask(javax.servlet.http.HttpServletRequest, javax.ser
Line Breakpoints
Source: D:\WebSphereSSE\AppServer\installedApps\piggybank-pkg2.ear\piggybank-pkg2.wa

Properties for Source: D:\WebSphereSSE\App...

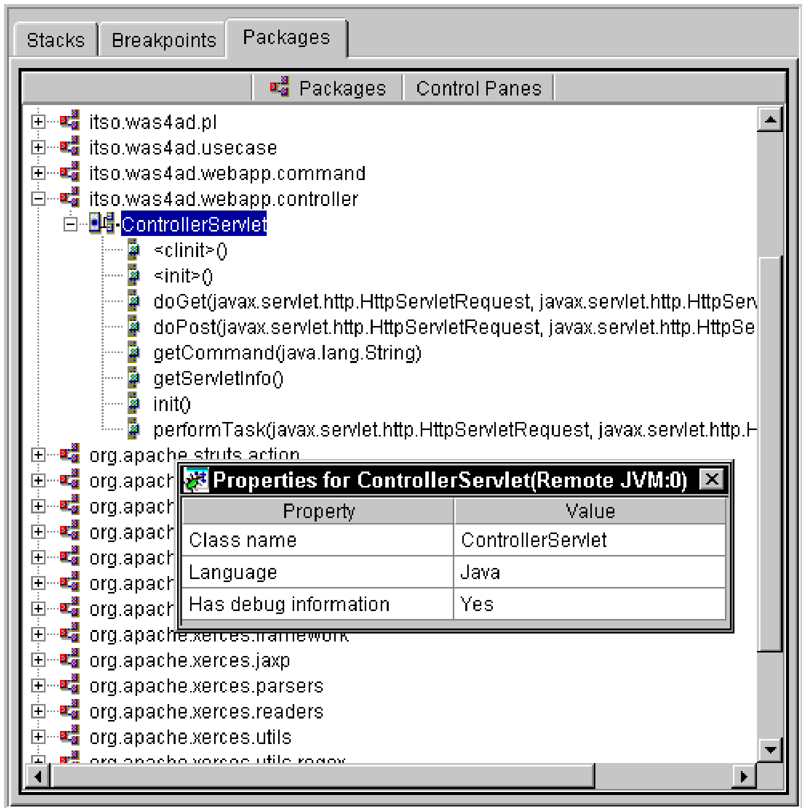
Property	Value
Package	<default>
Class	logout_jsp_0
Source	D:\WebSphereSSE\AppServ...
Method Name	_jspx_init()
Line	40
Address	
State	Enabled
Status	Active
Thread	Every
Conditional Expression	
From	1
To	Infinity
Every	1

Figure 17-27 Breakpoints pane

Packages pane

The *Packages* pane displays a list of the packages used by the application. The `<default>` package includes the JSPs.

The context menu allows us to set breakpoints for the methods, as well as to see the *Properties* information for the elements (for example, if the classes include debug information). An example for the PiggyBank Controller Servlet is shown in Figure 17-28.



```
itso.was4ad.pl
itso.was4ad.usecase
itso.was4ad.webapp.command
itso.was4ad.webapp.controller
  ControllerServlet
    <clinit>()
    <init>()
    doGet(javax.servlet.http.HttpServletRequest, javax.servlet.http.HttpServ
    doPost(javax.servlet.http.HttpServletRequest, javax.servlet.http.HttpSe
    getCommand(java.lang.String)
    getServletInfo()
    init()
    performTask(javax.servlet.http.HttpServletRequest, javax.servlet.http.H
org.apache.struts.action
org.apach
org.apach
org.apach
org.apach
org.apach
org.apach
org.apache.xerces.framework
org.apache.xerces.jaxp
org.apache.xerces.parsers
org.apache.xerces.readers
org.apache.xerces.utils
org.apache.xerces.utils.regex
```

Properties for ControllerServlet(Remote JVM:0)

Property	Value
Class name	ControllerServlet
Language	Java
Has debug information	Yes

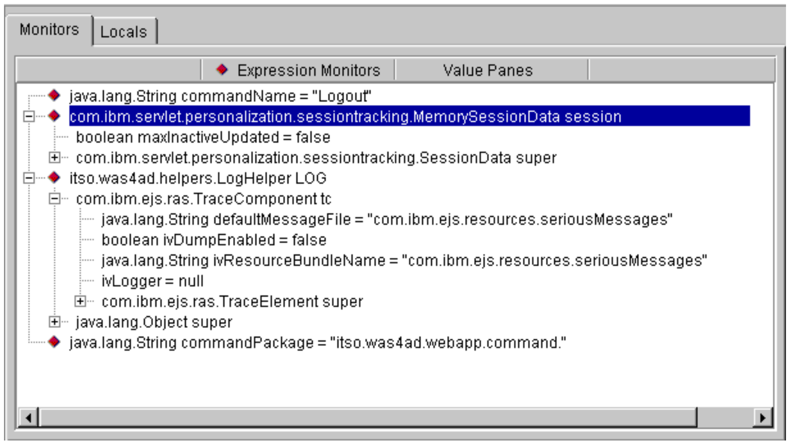
Figure 17-28 Packages pane

Stacks pane

The *Stacks* pane provides a view of the stack in each thread of the program we are debugging.

Monitors pane

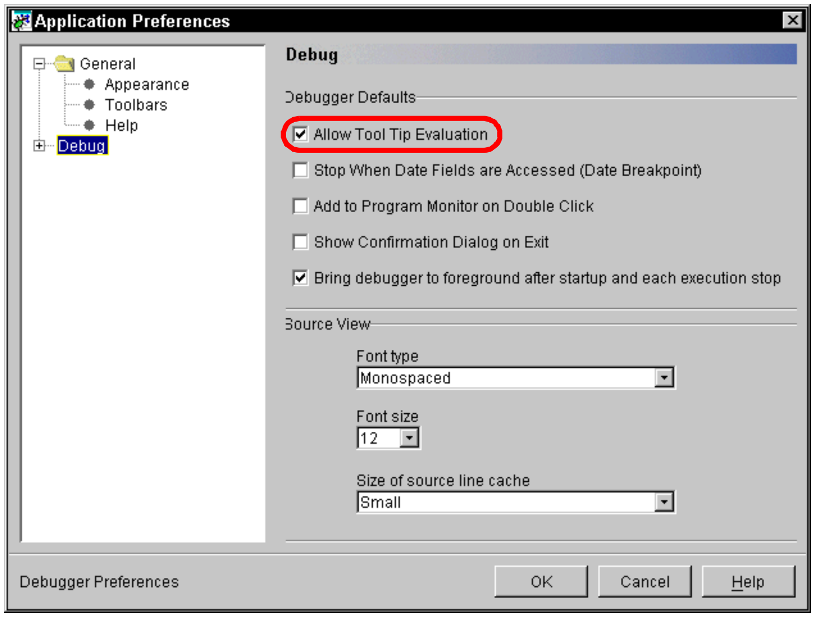
The *Monitors* pane (Figure 17-29) shows a list of the variables and expressions that we have selected to monitor. We can enable, disable or delete the monitored elements through this pane (right button menu). Options concerning all the monitored elements are available through the *Monitors* menu.



```
java.lang.String commandName = "Logout"
com.ibm.servlet.personalization.sessiontracking.MemorySessionData session
    boolean maxInactiveUpdated = false
    com.ibm.servlet.personalization.sessiontracking.SessionData super
itso.was4ad.helpers.LogHelper LOG
    com.ibm.ejs.ras.TraceComponent tc
        java.lang.String defaultMessageFile = "com.ibm.ejs.resources.seriousMessages"
        boolean ivDumpEnabled = false
        java.lang.String ivResourceBundleName = "com.ibm.ejs.resources.seriousMessages"
        ivLogger = null
        com.ibm.ejs.ras.TraceElement super
    java.lang.Object super
java.lang.String commandPackage = "itso.was4ad.webapp.command."
```

Figure 17-29 Monitors pane

This pane is useful when we want to monitor global variables throughout the debugging process. To see the current value of a variable, we suggest to activate the *Tool Tip Evaluation* for variables, either in the *Source* menu or through *File -> Preferences -> Debug* (Figure 17-30).



Debug

Debugger Defaults

[✓] Allow Tool Tip Evaluation
[ ] Stop When Date Fields are Accessed (Date Breakpoint)
[ ] Add to Program Monitor on Double Click
[ ] Show Confirmation Dialog on Exit
[✓] Bring debugger to foreground after startup and each execution stop

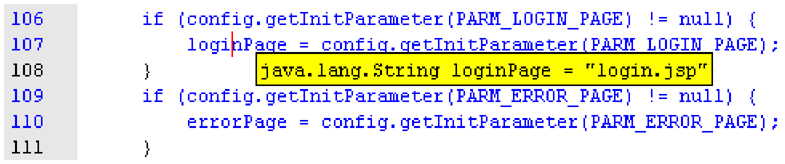
Source View

Font type: Monospaced
Font size: 12
Size of source line cache: Small

Debugger Preferences

Figure 17-30 Enabling the tool tip evaluation for variables

This option provides a "hover help" on variable values in the source pane, so that at any time we can view the value of a variable simply by pointing at it (Figure 17-31).

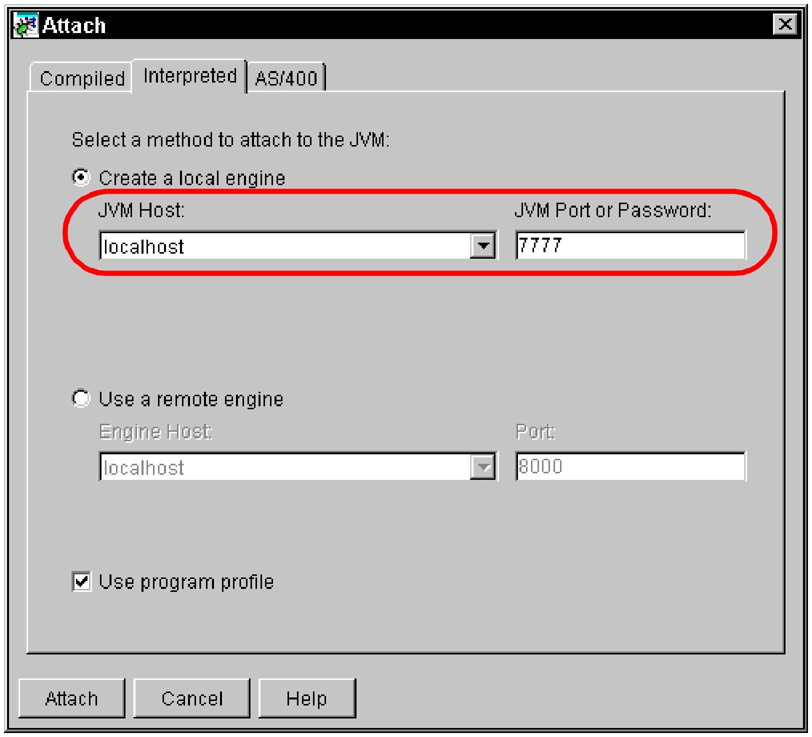


```
        if (config.getInitParameter(PARM_LOGIN_PAGE) != null) {
            loginPage = config.getInitParameter(PARM_LOGIN_PAGE);
        }       java.lang.String loginPage = "login.jsp"
        if (config.getInitParameter(PARM_ERROR_PAGE) != null) {
            errorPage = config.getInitParameter(PARM_ERROR_PAGE);
        }
```

Figure 17-31 Using the tool tip evaluation on the source pane

Attaching the debugger to the JVM

After setting up the environment, we are ready to attach to the application server's JVM. It is possible to use the command line arguments of `idebug.exe` to provide the data, but if no arguments are used, we select *File -> Attach* and specify the appropriate data (Figure 17-32).



Select a method to attach to the JVM:

● Create a local engine

JVM Host: localhost

JVM Port or Password: 7777

○ Use a remote engine

Engine Host: localhost

Port: 8000

☑ Use program profile

Figure 17-32 Attaching to a local JVM with the Distributed Debugger

To attach to a running JVM from the command line, we use the command shown in Figure 17-33.

```
idebug -a0 -host=hostname -password=7777
```

Figure 17-33 Attaching to a local JVM through the command line

- The `-a` option indicates the attachment, and the `0` indicates that it is made to a JVM.
- If we are attaching to a local JVM (the application server and the debugger's daemon and client are in the same machine), we use `localhost` as the host name.
- If we are attaching to a remote JVM but the debugger's daemon and client are in the same machine, we use the same command, but specifying the name of the machine where the application is running as the host name.

At this point you can run the application. See "Working with breakpoints" on page 499 on how to set breakpoints to stop the application. You can also set deferred breakpoints in classes that have not been loaded yet.

Attaching to a different machine

If we have the debugger's daemon installed in a different machine than the interface, we first start the daemon connecting to the running JVM (Figure 17-34).

```
irmtdbgj -qhost=debuggerhostname -quiport=8001 -host=hostname
        -password=agent_password_or_port_number
```

Figure 17-34 Starting the debugger's daemon remotely

- The `debuggerhostname` is the TCP/IP name or address of the machine where the debugger's interface will be running.
- `The hostname` is the TCP/IP name or address of the machine where the JVM we want to attach to is running (they can be both the same). If the daemon and the JVM are in the same machine, we use `localhost` instead of the `hostname`.

Other `irmtdbgj` options are shown in Figure 17-35.

```
Usage: irmtdbgj [debugger options] [jvm attach options]
                [ui daemon options [class [parameters]]]

debugger options:
        [-help] [-multi] [-qquiet] [-qfilter=<filter file>] [-lang=<lang>]
        [-qport=<service port>] [-jvmargs=<args>]
where:
   -help          help for command
   -multi         allows connections from multiple front ends
   -qquiet        suppresses irmtdbgj output
   <filter file>  file containing the list of packages not to be debugged
   <lang>         is the console locale (eg. en_US, jp_JP)
   <service port> is the port on which the engine will listen (default 8000)
   <args>         are the arguments passed to the JVM which will run the
                  application to be debugged

jvm attach options:
        [-host=<hostname> -password=<password>]
where:
   <hostname>     is the host name of the JVM to attach to
                  (name or IP address)
   <password>     is the agent password of the JVM to attach to

ui daemon options (auto UI invocation):
        [-qhost=<uidhost>] [-quiport=<uidport>] [-qtitle=<uidtitle>] [-s]
where:
   <uidhost>      is a UI Daemon's host name
   <uidport>      is a UI Daemon's port (default 8001)
   <uidtitle>     is the title for the debug session in the UI
   -s             causes the debugger to run the application after attach
                  (default is to stop the application after attach)
```

Figure 17-35 Irmtdbgj options

After starting the daemon we start the interface connecting to the daemon (Figure 17-36). The default port for connecting to the daemon is `8000`.

```
idebug -a0 -qhost=daemonhost -qport=8000
```

Figure 17-36 Starting the debugger's interface

You would also use this command to attach to the WebSphere Application Server when using the Distributed Debugger without OLT. The `qport` would be the address specified in the `-Xrunjdwp` options (Figure 17-12 on page 481).

When using OLT in combination with the Distributed Debugger, it is not necessary to launch the debugger separately: OLT launches the program when it encounters the first method breakpoint set (or, if we are operating under the step-by-step mode, when it encounters the first debuggable method), and the debugger will attach automatically to the remote JVM.

Attention: When using the Distributed Debugger and you click the *Terminate* button (or select *Debug -> Terminate*), the program does not stop the current thread, but the whole JVM—in the case of a WebSphere application this means the WebSphere Application Server itself is stopped.

To stop debugging and continue with the normal flow of the application, we use the option *Detach Program*. We can always reattach to the JVM at any time by activating a breakpoint or the *Step-by-step* debug mode in OLT.

Debugging Web applications with the debugger

In this section we explain the techniques to debug Web applications with the Distributed Debugger.

Working with breakpoints

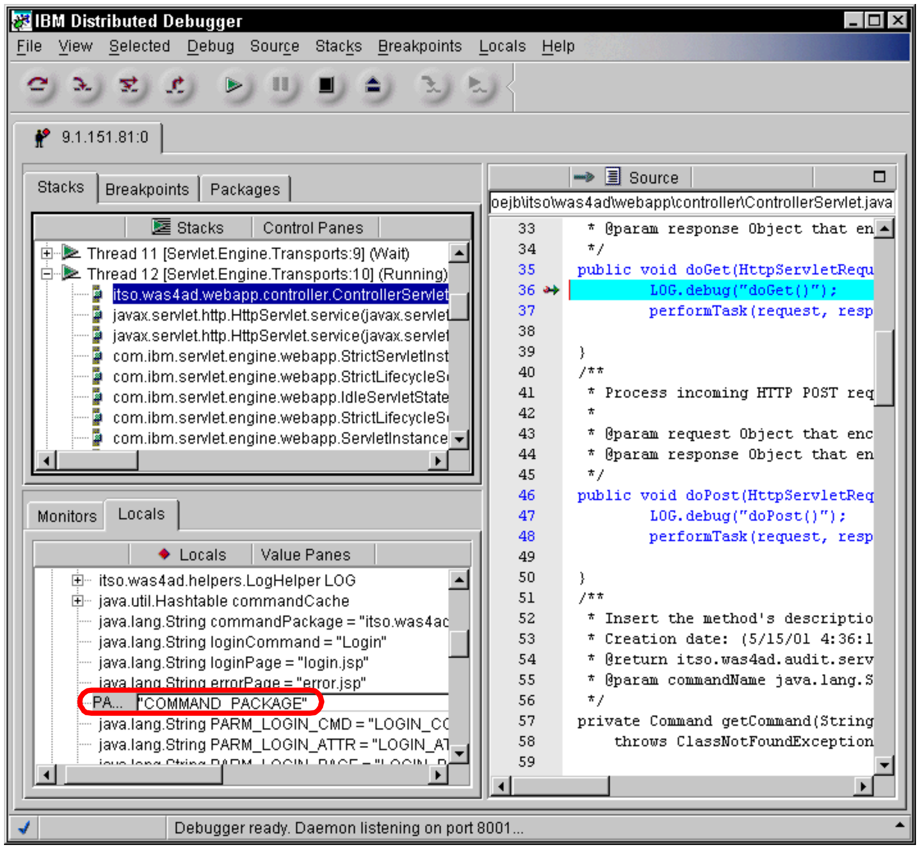
We have already listed the three types of breakpoints available for the Distributed Debugger:

- Line breakpoints
- Method breakpoints
- Watchpoints

It is possible to set line breakpoints by double clicking on the line number in the Source pane. The breakpoint information is automatically added to the *Breakpoints* pane.

When creating a breakpoint in the Line Breakpoint dialog (*Breakpoints -> Set Line* or *Set Method*), you can create a breakpoint in a class that is not yet loaded by selecting the *Defer Breakpoint* check box. The breakpoint is enabled when that DLL or package is loaded, and then it behaves as a normal breakpoint.

Whenever the debugger stops on a breakpoint, the suspended thread is shown in the *Stacks* pane. In the *Locals* pane, we can see the subelements of the trace (it is possible to specify the number of subelements to be displayed), including the variables. At this point we can change the variables values to vary the execution path of the program (Figure 17-37).



```
Thread 11 [Servlet.Engine.Transports:9] (Wait)
Thread 12 [Servlet.Engine.Transports:10] (Running)
    itso.was4ad.webapp.controller.ControllerServlet
    javax.servlet.http.HttpServlet.service(javax.servlet
    javax.servlet.http.HttpServlet.service(javax.servlet
    com.ibm.servlet.engine.webapp.StrictServletInst
    com.ibm.servlet.engine.webapp.StrictLifecycleS
    com.ibm.servlet.engine.webapp.IdleServletState
    com.ibm.servlet.engine.webapp.StrictLifecycleS
    com.ibm.servlet.engine.webapp.ServletInstance
```

```
itso.was4ad.helpers.LogHelper LOG
java.util.Hashtable commandCache
java.lang.String commandPackage = "itso.was4ac
java.lang.String loginCommand = "Login"
java.lang.String loginPage = "login.jsp"
java.lang.String errorPage = "error.jsp"
PA...  "COMMAND_PACKAGE"
java.lang.String PARM_LOGIN_CMD = "LOGIN_CC
java.lang.String PARM_LOGIN_ATTR = "LOGIN_AT
```

```
oejb\itso\was4ad\webapp\controller\ControllerServlet.java
     * @param response Object that en
     */
    public void doGet(HttpServletRequ
            LOG.debug("doGet()");
            performTask(request, resp

    }
    /**
     * Process incoming HTTP POST req
     *
     * @param request Object that enc
     * @param response Object that en
     */
    public void doPost(HttpServletReq
            LOG.debug("doPost()");
            performTask(request, resp

    }
    /**
     * Insert the method's descriptio
     * Creation date: (5/15/01 4:36:1
     * @return itso.was4ad.audit.serv
     * @param commandName java.lang.S
     */
    private Command getCommand(String
        throws ClassNotFoundException
```

Debugger ready. Daemon listening on port 8001...

Figure 17-37 Working with breakpoints in the Distributed Debugger

When stopping in a breakpoint, the options available for continuing with a step by step execution are the following:

Step over—to skip over the current statement

Step into—to step into the current message

Step debug—to step into the next debuggable statement

Step return or s*tep out*—to exit the current method and go back to the calling method

Only the *Step debug* option is different from the debug options available with VisualAge for Java.

If we select *Step debug*, the debugger does not step into the base class code; this is very useful, because we want to debug the code we wrote, not the JDK classes. The debugger looks in the following packages to check if a method is in a base class:

- `java.*`
- `javax.*`
- `sun.*`
- `com.sun.*`
- `com.ibm.*`
- `org.omg.*`
- `org.xml.*`
- `org.w3c.*`

If we want to add other classes to the base class list (so that the debugger doesn't step into them during the execution), we can add an option to the `DEBUG_OPTIONS` when starting the application server (that would be in the `startServer.bat` file for AEs or in the JVM settings for the corresponding application server in AE's console):

```
-qfilter=%WAS_HOME%\bin\debug.lst
```

Where `debug.lst` is a plain text file containing the base class list. An example is shown in Figure 17-38.

```
java.*
javax.*
sun.*
com.sun.*
org.omg.*
org.xml.*
org.w3c.*
com.ibm.som.*
com.ibm.CORBA.*
com.ibm.debug.*
com.ibm.IExtendedNaming.*
com.ibm.IExtendedLifeCycle.*
com.ibm.CBCUtil.*
com.ibm.IManagedClient.*
com.ibm.IManagedCollections.*
com.ibm.ISessions.*
com.ibm.IQueryManagedClient.*
com.ibm.IExtendedQuery.*
com.ibm.ICollectionBase.*
```

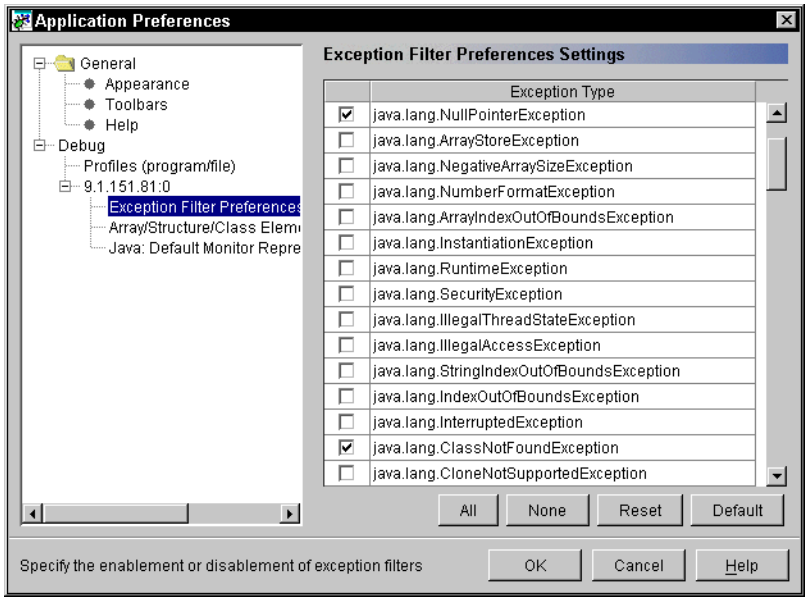
Figure 17-38 An example of a base classes list

Another function provided by the Distributed Debugger is the *Run to location* function. To use it, we select a statement in the *Source* pane and execute this option (right button menu or *Debug -> Run to Location*). The debugger runs the program up to the specified location (if no active breakpoints are hit). This is useful when we want to skip sections of code not interesting for debugging purposes.

Exceptions

The Distributed Debugger allows us to select from a list of recognized exceptions that stop the execution of the program if thrown (they can be uncaught exceptions or exceptions handled in a `catch` or `finally` block).

We select the exceptions to be monitored in *File -> Preferences -> Debug -> ProcessName -> Exception Filter Preferences Settings* (Figure 17-39).

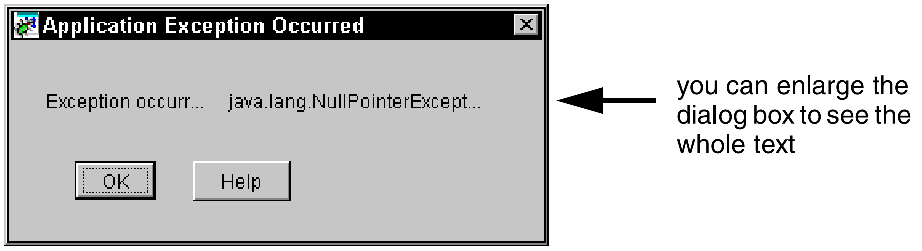

Application Preferences
General
Appearance
Toolbars
Help
Debug
Profiles (program/file)
9.1.151.81:0
Exception Filter Preferences
Array/Structure/Class Elem
Java: Default Monitor Repre
Exception Filter Preferences Settings
All
None
Reset
Default
Specify the enablement or disablement of exception filters
OK
Cancel
Help

	Exception Type
✓	java.lang.NullPointerException
☐	java.lang.ArrayStoreException
☐	java.lang.NegativeArraySizeException
☐	java.lang.NumberFormatException
☐	java.lang.ArrayIndexOutOfBoundsException
☐	java.lang.InstantiationException
☐	java.lang.RuntimeException
☐	java.lang.SecurityException
☐	java.lang.IllegalThreadStateException
☐	java.lang.IllegalAccessException
☐	java.lang.StringIndexOutOfBoundsException
☐	java.lang.IndexOutOfBoundsException
☐	java.lang.InterruptedException
✓	java.lang.ClassNotFoundException
☐	java.lang.CloneNotSupportedException

Figure 17-39 Selecting exception filtering in the Distributed Debugger

When the debugger encounters an exception, it displays the exception name and it points to the code line where it was thrown in the *Source* pane, if the source code is available.

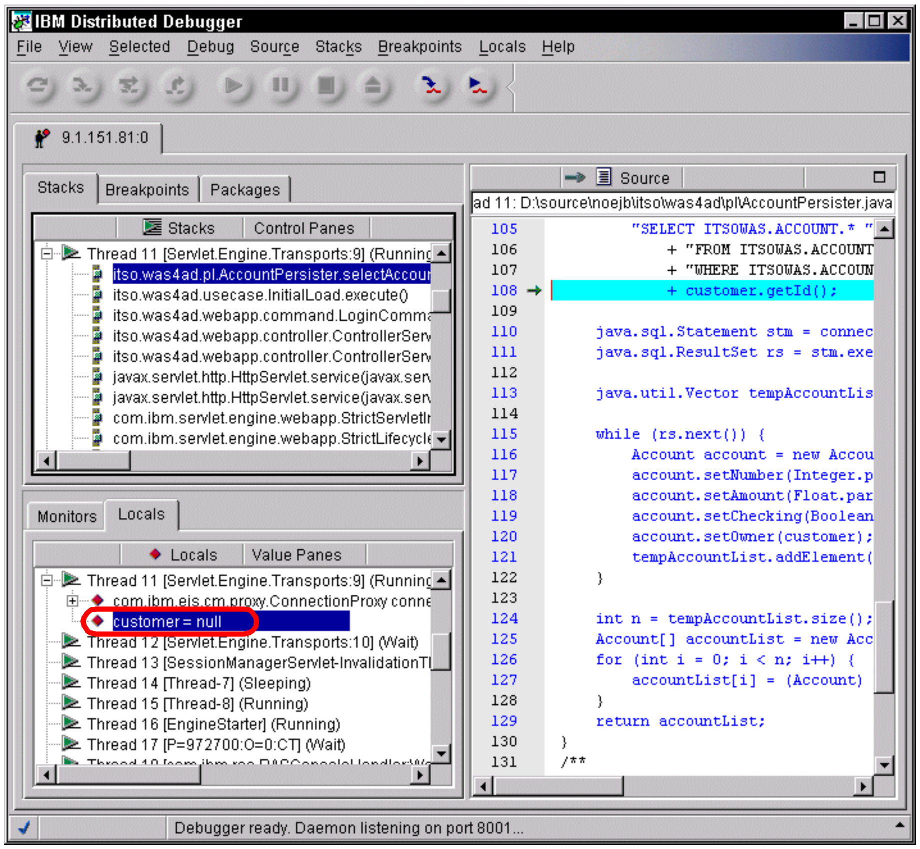
For example, if we query a customer number that is not in the database, the program throws a `NullPointerException` when trying to access retrieved data (Figure 17-40).


Application Exception Occurred
OK
Help
you can enlarge the dialog box to see the whole text

```
Exception occurr...    java.lang.NullPointerExcept...
```

Figure 17-40 Exception detected by the Distributed Debugger

Whenever an exception is detected by the debugger, we are able to inspect the data in the *Locals* pane. In this example, we see that the `customer` variable is null, which causes the exception to be thrown (Figure 17-41).



```
Thread 11 [Servlet.Engine.Transports:9] (Running
    itso.was4ad.pl.AccountPersister.selectAccour
    itso.was4ad.usecase.InitialLoad.execute()
    itso.was4ad.webapp.command.LoginComma
    itso.was4ad.webapp.controller.ControllerServ
    itso.was4ad.webapp.controller.ControllerServ
    javax.servlet.http.HttpServlet.service(javax.serv
    javax.servlet.http.HttpServlet.service(javax.serv
    com.ibm.servlet.engine.webapp.StrictServletIn
    com.ibm.servlet.engine.webapp.StrictLifecycle
```

```
Thread 11 [Servlet.Engine.Transports:9] (Running
    com.ibm.ejs.cm.proxy.ConnectionProxy conne
    customer = null
Thread 12 [Servlet.Engine.Transports:10] (Wait)
Thread 13 [SessionManagerServlet-InvalidationTI
Thread 14 [Thread-7] (Sleeping)
Thread 15 [Thread-8] (Running)
Thread 16 [EngineStarter] (Running)
Thread 17 [P=972700:O=0:CT] (Wait)
```

```
ad 11: D:\source\noejb\itso\was4ad\pl\AccountPersister.java
            "SELECT ITSOWAS.ACCOUNT.* "
                + "FROM ITSOWAS.ACCOUNT
                + "WHERE ITSOWAS.ACCOUN
                + customer.getId();

        java.sql.Statement stm = connec
        java.sql.ResultSet rs = stm.exe

        java.util.Vector tempAccountLis

        while (rs.next()) {
            Account account = new Accou
            account.setNumber(Integer.p
            account.setAmount(Float.par
            account.setChecking(Boolean
            account.setOwner(customer);
            tempAccountList.addElement(
        }

        int n = tempAccountList.size();
        Account[] accountList = new Acc
        for (int i = 0; i < n; i++) {
            accountList[i] = (Account)
        }
        return accountList;
    }
    /**
```

```
Debugger ready. Daemon listening on port 8001...
```

Figure 17-41 Inspecting data after an exception detection

Two options are available to continue the execution:

- *Step exception*
- *Run exception*

With *Step exception*, the debugger stops at the `catch` block handling the exception (if any), or at the method that started the JVM thread if it is an uncaught exception.

Run exception continues the execution of the program, stopping at any `catch` or `finally` blocks (as *Step exception*), if any. For an uncaught exception, the debugger itself handles it (writing the data to a command window).

Inspecting data

If we want to keep track of a variable value, the Distributed Debugger gives us the possibility of adding the variable to the *Monitors* pane (see Figure 17-29 on page 495), so that we can keep control of its value during the execution process.

We can add variables to the *Monitors* pane using the *Monitors* menu, or by double clicking on the selected variable declaration (we have to enable this option in the *Preferences* window: *Debug -> Add to program monitor on double click*). We can change the values of these monitored applications in the *Monitors* pane (in a similar way to other non-monitored variables in the *Locals* pane).

Debugging standalone applications with the debugger

To attach the Distributed Debugger to a running standalone application, Java 1.3.x applications have to be executed with the options shown in Figure 17-42.

```
java -Xdebug -Xnoagent
     -Djava.compiler=NONE
     -Xbootclasspath/a:<WAS_HOME>/java/lib/tools.jar
     -Xrunjdwp:transport=dt_socket,server=y,suspend=n <classname and args>
```

Figure 17-42 Executing Java 1.3.x applications for debugging

The `tools.jar` file contains the JPDA classes and is passed using the `-Xbootclasspath` option.

Check the Distributed Debugger documentation for details on other options.

Other JVM options for the program, such as class path entries, can be included with the former. When executing the program, we get this output in the command line:

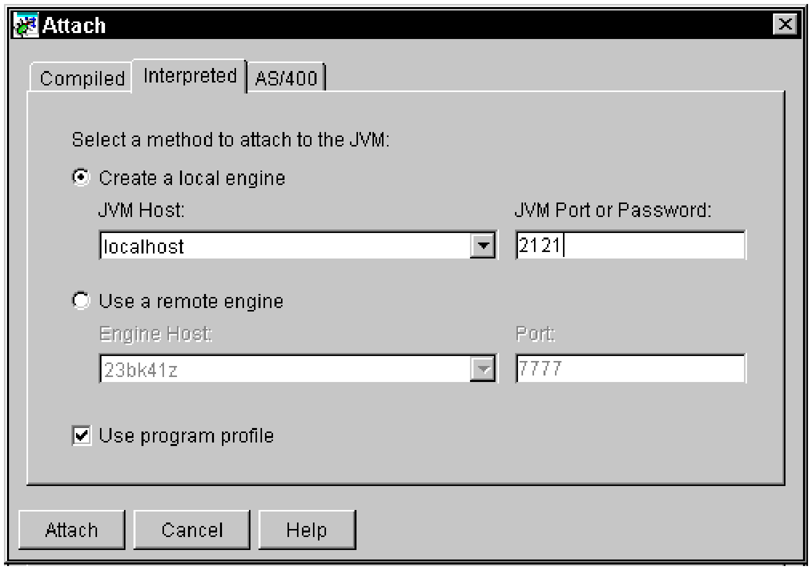
```
Listening for transport dt_socket at address: 2121
```

We can force the JVM to listen in a specific port number by adding `address=<portnumber>` to the `-Xrunjdwp` option (Figure 17-43).

```
-Xrunjdwp:transport=dt_socket,server=y,suspend=n,address=portnumber
```

Figure 17-43 Specifying a JVM port

Once we have this port number, we can connect to the running JVM from the debugger's interface or from the command line. An example of connecting through the interface is shown in Figure 17-44.



Select a method to attach to the JVM:

[✓ Create a local engine]

JVM Host: localhost

JVM Port or Password: 2121

Use a remote engine

Engine Host: 23bk41z

Port: 7777

[✓] Use program profile

Figure 17-44 Attaching to a local JVM running a Java client

To perform the attachment through the command line we use the command shown in Figure 17-45.

```
idebug -a0 -password=2121 -host=machineid
```

Figure 17-45 Attaching to a JVM machine running an application client

We can also use `localhost` as machine ID for local attachment.

For remote attachment, we start the debugger daemon listening on the JVM port, as we have described previously (Figure 17-46).

```
irmtdbgj -qhost=workstation_id -quiport=8001 -host=hostname -password=2121
```

Figure 17-46 Launching the debugger daemon

Then we launch the debugger client program (Figure 17-47).

```
idebug -a0 -qhost=hostname -qport=8000
```

Figure 17-47 Launching the debugger client

The `hostname` is the TCP/IP name or address of the machine where the debugger daemon is running (if all the three hosts are the same, then we perform local debugging, so we use the commands described before for this purpose).

If we are using WAS 4.0, we must use the `launchClient` command line program to launch our J2EE client application, instead of doing it directly by the Java runtime. We can find this utility program in

```
D:\WebSphere\AppServer\bin\launchclient.bat
```

Its usage is described in “Performing a unit test: executing the application” on page 463. `launchClient` does not include the `-Xrunjdwp` option that allows us to get or specify the JVM port number, but if we are going to use this tool to launch our client application, it is easy to edit the file and add the option (Figure 17-48).

```
@echo off
REM Usage: launchClient [<ear-file> | -help | -?]

setlocal
call "%~dp0setupCmdLine.bat"

set NAMING_FACTORY=com.ibm.websphere.naming.WsnInitialContextFactory

%JAVA_HOME%\bin\java %CLIENTSAS% -Xdebug -Xnoagent -Djava.compiler=NONE
         -Xrunjdwp:transport=dt_socket,server=y,suspend=n
-Dws.ext.dirs=%WAS_HOME%/classes;%WAS_HOME%/lib/ext;%WAS_HOME%/lib;%WAS_HOME
%/web/help;%WAS_HOME%/properties;%DBDRIVER_PATH% -Dcom.ibm.CORBA.Bootstrap
Host=%COMPUTERNAME% -Djava.naming.factory.initial=%NAMING_FACTORY%
-Dserver.root=%WAS_HOME% -jar %WAS_HOME%/lib/bootstrap.jar
com.ibm.websphere.client.applicationclient.launchClient %*

endlocal
```

Figure 17-48 Editing the launchClient.bat file to include JVM port information

Note: Our example illustrates modifying the Windows version of the `launchClient` script—the same principle also works on UNIX platforms, but the scripting languages differ.

Then, when we launch the application, we get the same message about the port number in the command line (Figure 17-49).

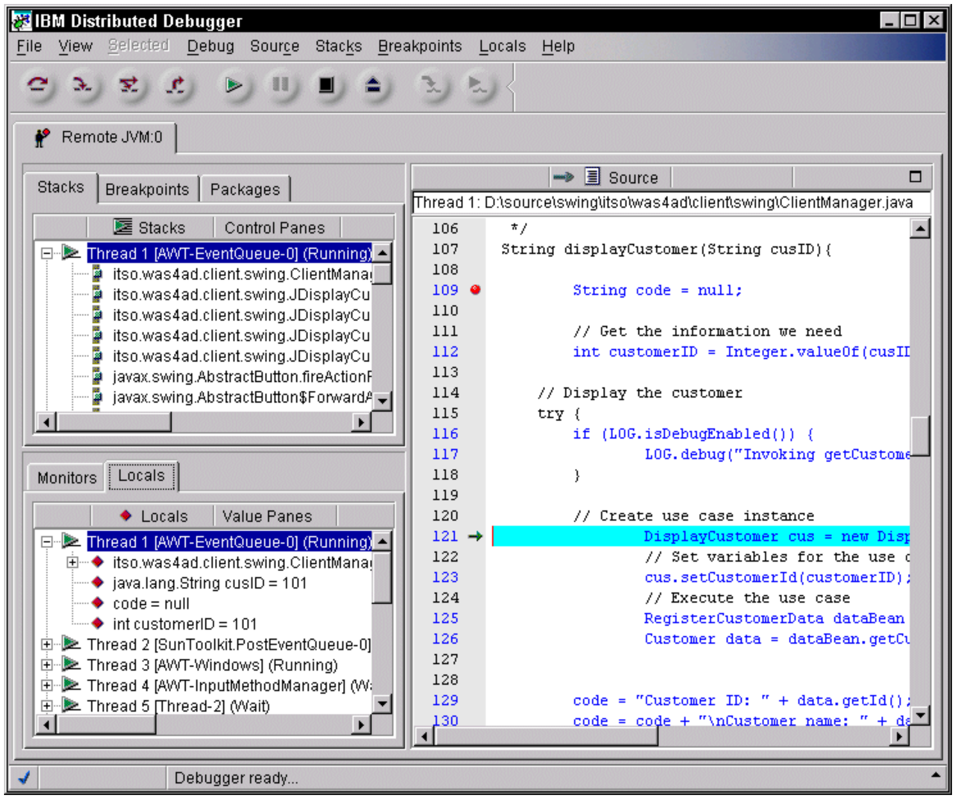
```
D:\WebSphereSSE\AppServer\bin>launchClient
   ..\installedapps\piggybank-swing.ear -CCjar=pb-swingclient.jar
Listening for transport dt_socket at address: 2363
IBM WebSphere Application Server, Release 4.0
J2EE Application Client Tool, Version 1.0
Copyright IBM Corp., 1997-2001

WSCL0012I: Processing command line arguments.
WSCL0013I: Initializing the J2EE Application Client Environment.
WSCL0035I: Initialization of the J2EE Application Client Environment has
completed.
WSCL0014I: Invoking the Application Client class
itso.was4ad.client.swing.SwingStandaloneClient
```

Figure 17-49 Using the edited launchClient command line tool

We take the port information and attach to the JVM locally or remotely in the same way as before. Then we are ready for debugging.

Debugging application clients is not different from debugging Web applications. We are prompted to locate the source files if necessary, and we can work with breakpoints, as in any other case.

An example where we are debugging the PiggyBank Swing application client is shown in Figure 17-50.



```
Thread 1 [AWT-EventQueue-0] (Running)
    itso.was4ad.client.swing.ClientMana
    itso.was4ad.client.swing.JDisplayCu
    itso.was4ad.client.swing.JDisplayCu
    itso.was4ad.client.swing.JDisplayCu
    itso.was4ad.client.swing.JDisplayCu
    javax.swing.AbstractButton.fireActionF
    javax.swing.AbstractButton$ForwardA
```

```
Thread 1 [AWT-EventQueue-0] (Running)
    itso.was4ad.client.swing.ClientMana
    java.lang.String cusID = 101
    code = null
    int customerID = 101
Thread 2 [SunToolkit.PostEventQueue-0]
Thread 3 [AWT-Windows] (Running)
Thread 4 [AWT-InputMethodManager] (W
Thread 5 [Thread-2] (Wait)
```

Thread 1: D:\source\swing\itso\was4ad\client\swing\ClientManager.java

```
   */
String displayCustomer(String cusID){

        String code = null;

        // Get the information we need
        int customerID = Integer.valueOf(cusII

    // Display the customer
    try {
        if (LOG.isDebugEnabled()) {
                LOG.debug("Invoking getCustome
        }

        // Create use case instance
                DisplayCustomer cus = new Disp
                // Set variables for the use c
                cus.setCustomerId(customerID);
                // Execute the use case
                RegisterCustomerData dataBean
                Customer data = dataBean.getCu


        code = "Customer ID: " + data.getId();
        code = code + "\nCustomer name: " + da
```

Figure 17-50 Debugging a client application using the Distributed Debugger

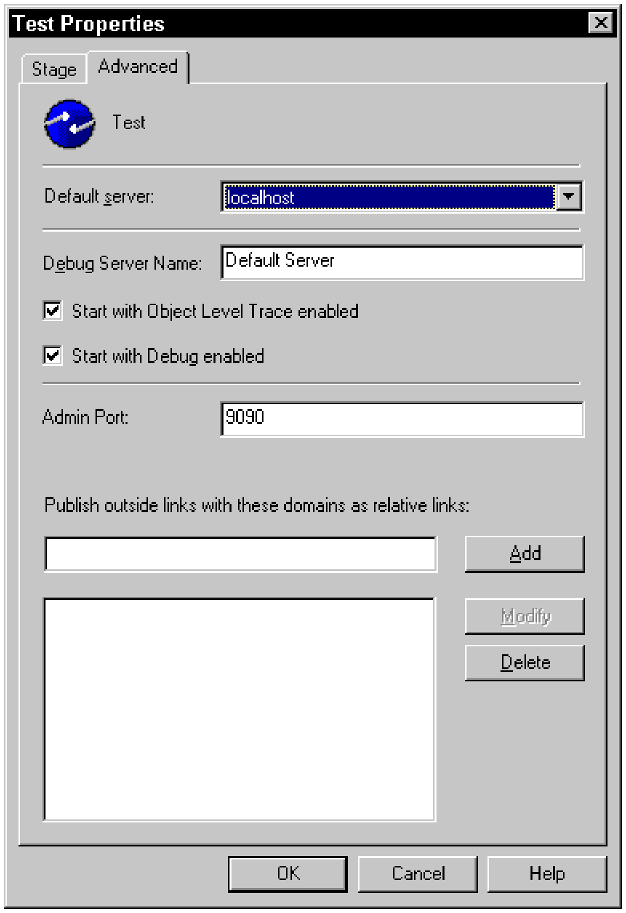
Debugging WebSphere Studio code

To debug the project files in WebSphere Studio, they must be compiled with the debug flag on and published to an application server that has the debug and OLT option enabled.

In the case of WAS 4.0, it is a requirement that we assemble the Web module before publishing to the server (see Chapter 10, “Development using WebSphere Studio” on page 237 for details about publishing Web archive files, and Chapter 15, “Assembling the application” on page 389 for general details about assembling the application modules).

Studio provides the *Debug publish* option, that compiles and publishes the java files and JSPs that we select. We can use this option to publish the class files to a temporary location so that afterwards we can assemble them into the Web archive (WAR file) for deploying in the server. This way, the classes contain the debug information necessary for later debugging.

The first thing we have to do in Studio prior to compiling and publishing the Java files is setting the debug server for each publishing stage (or for the stage where the debuggable code is). Figure 17-51 shows an example of setting up a debug server for the *Test* stage.


Test Properties

Test

Default server: localhost

Debug Server Name: Default Server

☑ Start with Object Level Trace enabled

☑ Start with Debug enabled

Admin Port: 9090

Publish outside links with these domains as relative links:

Figure 17-51 Setting up a debug server in Studio

To compile the java files and JSPs and publish them to the Debug Server, we use the *Debug Publish* option, accessed through the *Project* menu (Figure 17-52).

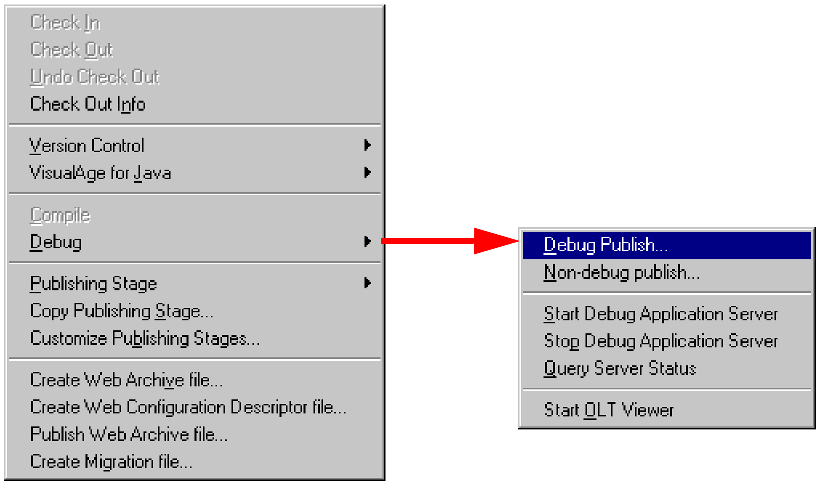


Figure 17-52 Using the Debug Publish option in Studio

The *Non-debug Publish* option compiles and publishes the selected files without debug information. *Query Server Status* is used to verify that the server is running. We can start the debug server directly enabling OLT and the debug mode, if we have selected this options when setting up the debug server for the current stage.

When using WAS 4.0 as the application server, we can test the just-published files only if we assemble them in a Web archive file, or if we are using this Studio feature to replace old files already deployed in the server.

For example, consider the case where we have already assembled and deployed the PiggyBank application in WAS. Then, by performing some unit tests, we discover that several JSPs have to be modified. We can do this in Studio and then publish them directly on the debug server (though in this case it is not necessary to compile the JSPs, because the application server compiles a JSP the next time the JSP is invoked).

We can set up different debug servers depending on the stage the application is in (using WAS 4.0 Advanced Edition, which allows to have several servers, but not for Single Server Edition, where we can only start one server per machine).

A special case: how to debug a JSP

JSPs are “harder” to debug than conventional Java classes because of their combination of HTML and Java code: we cannot import them in VisualAge as standard code, but we can take other approaches:

- In VisualAge for Java:
  - Import the compiled servlet code and debug it with the VisualAge for Java debugger
  - Use the JSP execution monitor
- Outside VisualAge for Java (from WebSphere Studio or directly from the application server where the application has been installed):
  - Use external debugging tools, such as the Distributed Debugger

We now illustrate the different methods and their advantages and disadvantages.

Debugging JSPs in VisualAge for Java

In this case, we have the Java code needed to run the application in the VisualAge for Java repository (servlets, JavaBeans), and we publish the Web files (HTML, JSPs) in the WebSphere Test Environment folder:

```
d:\IBMVAJava\ide\project_resources\IBM WebSphere Test Environment\
                    hosts\default_host\default_app\web
```

With Studio, it is easier to define a *VisualAge* publishing target so that we can publish the Web files automatically (for more information about this, see “Publishing stages and publishing targets” on page 241).

We then launch the WebSphere Test Environment (WTE). There are a number of options related to JSPs and we can perform the debugging tasks in several ways combining these options.

Debug a JSP without importing the code to VisualAge for Java

To avoid importing the generated code to VisualAge for Java, we select the option *Load generated servlet externally* in the WTE (see Figure 11-7 on page 274). The code is stored in:

```
...\IBM WebSphere Test Environment\temp\JSP1_1\default_app
```

Later, if desired, we can import the code to the workbench to debug it.

When we are performing exhaustive debugging of the JSPs, changing the code frequently to fix errors or to improve details, importing the generated servlet each time increases the VisualAge for Java repository unnecessarily.

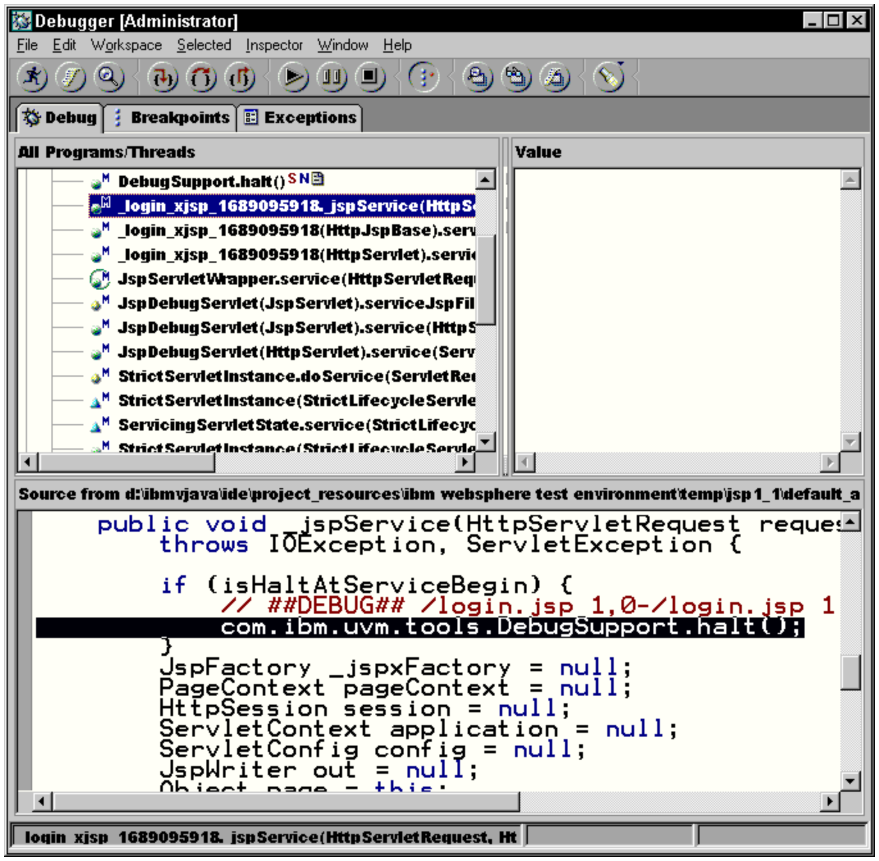
We have two suboptions for external loading of the generated code:

- Halt at the beginning of the service method
- Enable JSP source debugging

Halt at the beginning of the service method

This option sets a breakpoint at the beginning of the service method, after which we can continue the execution step by step or bypassing sections of code.

The code shown in the debugger window is the generated servlet code (Figure 17-53). We can set breakpoints and execute step by step as we would do with any Java class, but the Java code is stored outside of the repository.



```
DebugSupport.halt() SN
_login_xjsp_1689095918._jspService(HttpS
_login_xjsp_1689095918(HttpJspBase).serv
_login_xjsp_1689095918(HttpServlet).servi
JspServletWrapper.service(HttpServletReq
JspDebugServlet(JspServlet).serviceJspFil
JspDebugServlet(JspServlet).service(HttpS
JspDebugServlet(HttpServlet).service(Serv
StrictServletInstance.doService(ServletRe
StrictServletInstance(StrictLifecycleServle
ServicingServletState.service(StrictLifecyc
StrictServletInstance(StrictLifecycleServle
```

Source from d:\ibmvjava\ide\project_resources\ibm websphere test environment\temp\jsp1_1\default_a

```
    public void _jspService(HttpServletRequest reques
        throws IOException, ServletException {

        if (isHaltAtServiceBegin) {
            // ##DEBUG## /login.jsp 1,0-/login.jsp 1
            com.ibm.uvm.tools.DebugSupport.halt();
        }
        JspFactory _jspxFactory = null;
        PageContext pageContext = null;
        HttpSession session = null;
        ServletContext application = null;
        ServletConfig config = null;
        JspWriter out = null;
        Object page = this;
```

login xjsp 1689095918. jspService(HttpServletRequest, Ht

Figure 17-53 Halting at the beginning of the service method

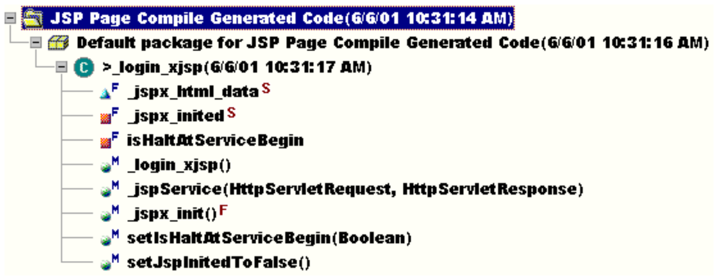
Enable JSP source debugging

This option allows to debug the JSP source with the VisualAge for Java debugger. We cannot change the code (because the generated Java servlet is not visible) and correct the possible errors. We should use the JSP editor we have used to create the pages, for example Studio, which provides a compiler that lets us do a first test). To enable a more exhaustive debugging, it is better to import the code and debug it as any other Java class.

Debug a JSP by importing the code

In this case, we do not select any options in the WTE window. The JSP compiler compiles and imports the code into the Workbench. We have to go through a first execution to get this generated code so that after we can debug it in the usual way by setting breakpoints for future executions.

The generated servlet code for the JSP can be found in the Workbench in the `JSP` Page Compile Generated Code project(Figure 17-54).



```
JSP Page Compile Generated Code(6/6/01 10:31:14 AM)
  Default package for JSP Page Compile Generated Code(6/6/01 10:31:16 AM)
    >_login_xjsp(6/6/01 10:31:17 AM)
      _jspx_html_data
      _jspx_inited
      isHaltAtServiceBegin
      _login_xjsp()
      _jspService(HttpServletRequest, HttpServletResponse)
      _jspx_init()
      setIsHaltAtServiceBegin(Boolean)
      setJspInitedToFalse()
```

Figure 17-54 JSP compiled code in VisualAge for Java

The generated servlet reads the HTML content of the JSP from a file stored under the generated code directory

```
...\IBM WebSphere Test Environment\temp\JSP1_1\default_app\etc\
```

and sends it back to the browser combined with the appropriate dynamic data.

However, debugging the JSP using the generated code means that we cannot see the HTML output (which might be interesting in some cases), and the generated servlet code is difficult to read, except for very simple JSPs.

We can fix small errors in the generated Java servlet for debugging purposes, but the errors must really be fixed in the JSP source code, and then recompiled.

Import of the generated servlet can fail if “bad” Java code has been inserted into the JSP (for example a missing bracket or semicolon). In this case the JSP compiler issues error messages. If the error is hard to diagnose, you can load the generated Java source servlet into a VisualAge for Java scrapbook window to analyze the code and get more meaningful error messages.

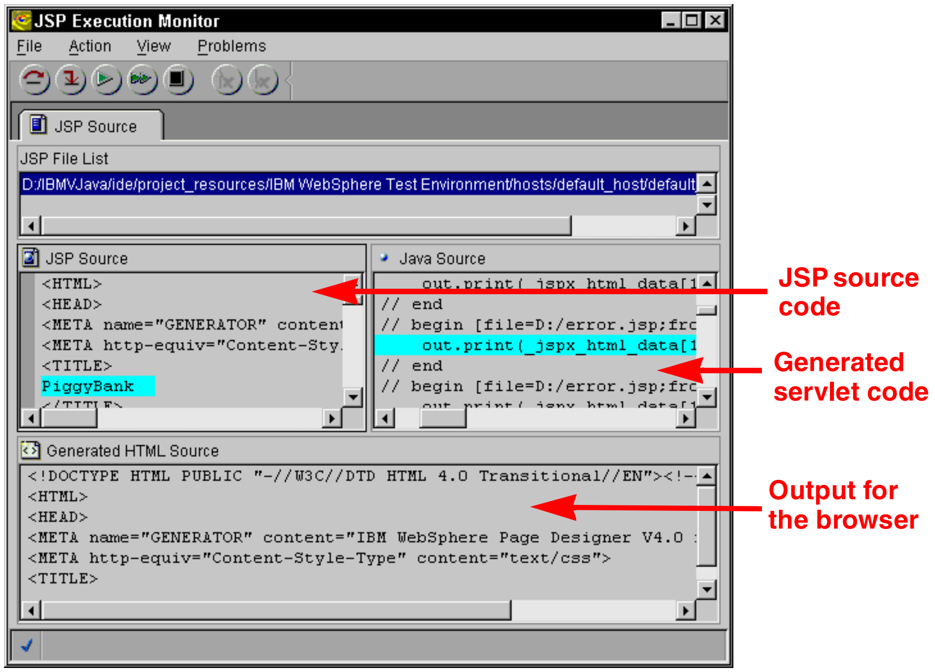
If we want to check the syntax errors directly in the JSP code, we can use the JSP Execution Monitor.

JSP execution monitor

This tool allows us to monitor the execution of the JSP and detect run-time errors. We can also set breakpoints in the JSP code and execute it step by step (in a similar way as how we would do it in the debugger).

In the WTE window, the option *Enable monitoring JSP execution* launches the tool when a JSP file is called in the browser. The option *Retrieve syntax error information* highlights syntax errors in the JSP source code in the monitoring tool window.

The monitor displays the JSP, the generated servlet code, and the HTML output (Figure 17-55). It is possible to change the display options in the *View* menu.



D:/IBMVJava/ide/project_resources/IBM WebSphere Test Environment/hosts/default_host/default

```
<HTML>
<HEAD>
<META name="GENERATOR" conten
<META http-equiv="Content-Sty
<TITLE>
PiggyBank
```

```
    out.print(_jspx_html_data[1
// end
// begin [file=D:/error.jsp;frc
    out.print(_jspx_html_data[1
// end
// begin [file=D:/error.jsp;frc
```

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.0 Transitional//EN"><!-
<HTML>
<HEAD>
<META name="GENERATOR" content="IBM WebSphere Page Designer V4.0
<META http-equiv="Content-Style-Type" content="text/css">
<TITLE>
```

Figure 17-55 JSP execution monitor

The *Step into IDebugger* (F5) option launches the debugger window with the generated servlet code so we can perform debugging tasks with a more suitable tool (and we are able to see the HTML output in the parallel *JSP Execution Monitor* window).

To set breakpoints in external files (for example, if we want to debug Java code that is not in the repository: from third parties or externally generated servlets):

- In the breakpoints tab of the debugger window, select *Methods -> External .class files breakpoints*, select the class file (from a directory or a JAR file), and then either *Set breakpoints in source* or *Break on method enter*.
- In the case of externally generated servlets, it is more useful to set the breakpoints directly in the JSP source code (or using the JSP Execution Monitor) if we do not want to import the code.

Debugging Studio JSPs: the Distributed Debugger

WebSphere Studio provides a compiler that allows us to make an initial check of the JSP code. However, this compiler does not provide options to set breakpoints or provide other more sophisticated debugging features.

The Distributed Debugger is an optional feature in the Studio Advanced Edition installation. It is the same debugger that is available with WebSphere Application Server and VisualAge for Java.

With the Distributed Debugger we can debug JSPs as well as any other Java code included in our Studio project. The steps to set up the debug server have been described before in "Debugging WebSphere Studio code" on page 508, so we do not repeat them here.

Let's suppose we have already published the JSPs to the server, assembled the application and deployed it. We can perform a first test on the JSPs by compiling them in the Server at deployment time or in Studio, before publishing. To include debug information in the compiled classes, we have to setup the Java Virtual Machine in the application server to be launched with the debug option (see "Enabling debugging support in WebSphere Application Server" on page 477).

After starting the server, we launch OLT (locally or remotely) and set breakpoints to begin the debug task.

The Distributed Debugger—because it is a line-based debugger—does not allow us to enter the Java code sections in the JSP, but we are able to set breakpoints in the compiled JSP methods (init, service), or line breakpoints. These breakpoints are visible in the *Source* pane.

We can step from one JSP executable line to the next without worrying about the underlying generated java code. This is good for a user who does not want to understand what set of Java source lines correspond to what JSP source line.

To control the execution, the two options available are *Step Debug* (it does the same as *Step Over* for JSPs) and *Run*. The JSPs variables are not visible in the *Locals* pane.

When we attempt to enter a JSP tag, the debugger displays whatever debug information was generated by the JSP processor. Variables declared in Java code blocks are accessible and can be added to the *Monitors* pane.

Debugging JSPs in WebSphere Application Server

Once JSPs have been compiled in the application server, they can be debugged in the same way as Java servlets. Breakpoints can be set to halt execution and all variables are available for monitoring.

Because the Java servlet is generated for a JSP, debugging is not quite as easy as for hand-written servlets. It takes some insight to understand the generated methods and Java code, and how that code relates to the original HTML code and the JSP tags. The HTML code is output as text constants that are created from the JSP source code.

18

Automating unit testing using JUnit

In this chapter we discuss the value of defining a unit testing strategy and employing automated unit testing in a development environment.

We then introduce JUnit, an open source framework for creating and running unit tests, and describe how it can be incorporated into the development process using examples from our PiggyBank application.

Unit testing

The first part of this chapter is a generalized discussion about unit testing. We define what we mean by the term *unit testing*, and discuss the benefits and problems associated with including a unit testing strategy in a development process. Finally we describe some of the benefits you may obtain from basing your unit tests upon an existing unit testing framework.

What is unit testing?

Before we delve into the details, let us first explain what unit testing means to us. During the course of a development project you often encounter many different terms associated with different types of testing, and it is common practice to dedicate individuals, teams or even entire departments to the testing function.

Unit tests, however, are informal tests that are generally executed by the developers of the application code. They are often quite low-level in nature, and test the behavior of individual software components such as individual Java classes, servlets or EJBs.

Because unit tests are usually written and performed by the application developer, they are often “white-box” in nature, that is to say they are written using knowledge about the implementation in mind, to test specific code paths, for example. This is not to say all unit tests have to be written this way—one common practice is to write the unit tests for a component based on the component specification *before* developing the component itself. Both approaches are valid—when defining your own unit testing policy you may want to make use of both.

Why unit testing?

On the face of it this is a question with a straightforward answer. We test to find defects in our code, and to verify that changes that we have made to existing code does not break that code. Perhaps it is more useful to look at the question from the opposite perspective, that is to say, why do developers *not* perform unit tests?

In general the simple answer is because it is too hard, and because nobody forces them to. Writing an effective set of unit tests for a component is not a trivial undertaking. Given the pressure to deliver that many developers find themselves subjected to, the temptation to postpone the creation and execution of unit tests in favour of delivering code fixes or new functionality is often overwhelming.

In practice, this usually turns out to be a false economy—developers very rarely deliver bug-free code, and the discovery of code defects and the costs associated with fixing them are simply pushed further out into the development cycle. This is inefficient—the best time to fix a code defect is immediately after the code has been written, while it is still fresh in the developer's mind. Furthermore, a defect discovered during a formal testing cycle must be written up, prioritized and tracked—all of these activities incur cost, and may mean that a fix is deferred indefinitely, or at least until it becomes critical.

Based on our experience, we believe that encouraging and supporting the development and regular execution of unit test cases ultimately leads to significant improvements in productivity and overall code quality. The creation of unit test cases need not be a burden—developers often find the intellectual challenge quite stimulating and ultimately satisfying. The thought process involved in creating a test can also highlights shortcomings a design which might not otherwise have been identified in a situation where the main focus is on implementation.

We recommend that you take the time to define a unit testing strategy for your own development projects. A simple set of guidelines and a framework that makes it easy to develop and execute tests will pay for itself surprisingly quickly.

Benefits of a unit testing framework

Once you have decided to implement a unit testing strategy in your project, the first hurdles to overcome are the factors that dissuade developers from creating and running unit tests in the first place. A testing framework can help by:

- Making it easier to write tests
- Making it easier to run tests

Tests are easier to write, because a lot of the infrastructure code that you require to support every test is already available. A testing framework also provides a facility that makes it easier to run and re-run tests, perhaps via a GUI. The more often a developer runs tests, the quicker problems can be located and fixed, because the delta between the code that last passed a unit test and the code that fails the test is smaller.

Testing frameworks also provide other benefits:

Consistency	Because every developer is using the same framework, all of your unit tests will work in the same way, can be managed in the same way, and report results in the same format.

Maintenance	Because a framework has already been developed and is probably already in use in a number of projects you spend less time maintaining your testing code.
Ramp-up time	If you select a popular testing framework, you may find that new developers coming into your team are already familiar with the tools and concepts involved.
Automation	A framework may offer the ability to run tests unattended, perhaps as part of a daily or nightly build (see “Automatic builds” on page 225).
Tool integration	A framework may have the ability to integrate with existing development tools such as Ant or VisualAge for Java.

JUnit

JUnit is an open source testing framework that is used to develop and execute unit tests in Java. It was written by Erich Gamma, one of the “Gang of Four” who wrote the classic book *Design Patterns*, and Kent Beck, who has also written extensively about object development and first described the eXtreme Programming (XP) software development process.

A good starting point for finding information about JUnit on the Web is the JUnit Web site:

`http://www.junit.org/`

This site contains documentation and links, as well as a free download that includes both the JUnit source and compiled code.

The rest of this chapter describes how we used JUnit to create and run unit tests for components of our example PiggyBank application, including a discussion about how to test Enterprise JavaBean (EJB) components. We also demonstrate how we can use Ant (described in “Using Ant to build a WebSphere application” on page 197) to automate the execution of test cases in a WebSphere environment.

While we briefly explain the JUnit features we take advantage of, we do not attempt to provide a comprehensive description of all of JUnit’s features—we recommend you consult the documentation included in the JUnit distribution.

The examples we describe in this section are included in the `junit` subdirectory of the additional Web material for this redbook. Appendix A, “Additional material” on page 557 describes how to obtain the Web material.

Installing JUnit

We downloaded Version 3.7 of JUnit, the latest version available at the time of writing, from a link on the JUnit Web site home page. We expanded the ZIP archive onto our local disk, creating the directory structure under `D:\junit3.7` shown in Figure 18-1.

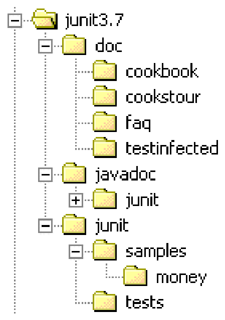


Figure 18-1 JUnit directory structure

The compiled JUnit code is located in the archive `junit.jar`, in the base directory. The source code is also available in the archive `src.jar` in the same location. Documentation is included in the package in the `doc` and `javadoc` directories.

Installing JUnit in VisualAge for Java

JUnit works particularly well in the VisualAge for Java environment—VisualAge's built-in incremental compilation allows you to leave the JUnit GUI running and re-run tests at the touch of a button as you modify your application code.

We decided to import the JUnit source code into our workspace in order to examine the code and allow us to step through it in the debugger if necessary. We created a new project named *JUnit*, and imported the source code into the new project from the JAR `src.jar` using the VisualAge *File > Import* menu option. We then versioned the project with the version name 3.7 to reflect the JUnit version number.

There is a more detailed discussion about how to integrate JUnit into VisualAge for Java on the JUnit Web site:

`http://www.junit.org/junit/doc/vaj/vaj.htm`

When we developed this chapter the latest version of VisualAge for Java described on this page was Version 3.5—the information is also applicable to Version 4.0 however.

Organizing our tests

We create test cases by extending the `junit.framework.TestCase` class. Each test case includes one or more individual tests. Each test is implemented in a method, which is given a name that starts with `test`, and describes the nature of the test performed, for example, `testDebit` tests the debit function of our component. The use of this naming convention is important, as we will explain shortly.

We create one test case class for each component we want to test, that is for each Java class or EJB. We place the test cases in a separate package from the tested components, created by appending the name `tests` to the package name:

```
itso.was4ad.webapp.view                    <=== component
itso.was4ad.webapp.view.tests              <=== test cases for component
```

We do this for two reasons:

- When we compile the test cases we cannot accidentally take advantage of additional privileges granted to us by virtue of being in the same package as the component under test.
- When we create production builds of our application we can easily filter out the test cases by excluding the contents of any package that ends with the name `tests`.

Test suites

Collections of test cases can be organized into test suites, managed by the `junit.framework.TestSuite` class. JUnit provides tools that allow every test in a suite to be run in turn and report on the results.

Both `TestCase` and `TestSuite` implement the Java interface `junit.framework.Test`, which allows us to organize our test cases into a hierarchy (Figure 18-2).

We create an additional `AllTests` test case class in each package containing test cases. This class defines a `suite` method that creates and returns a `TestSuite` comprising all of the test cases in the package.

We can use this hierarchy to select the tests we want to run, whether a single test case, or all of the test cases for a package, a module, or the entire PiggyBank application.

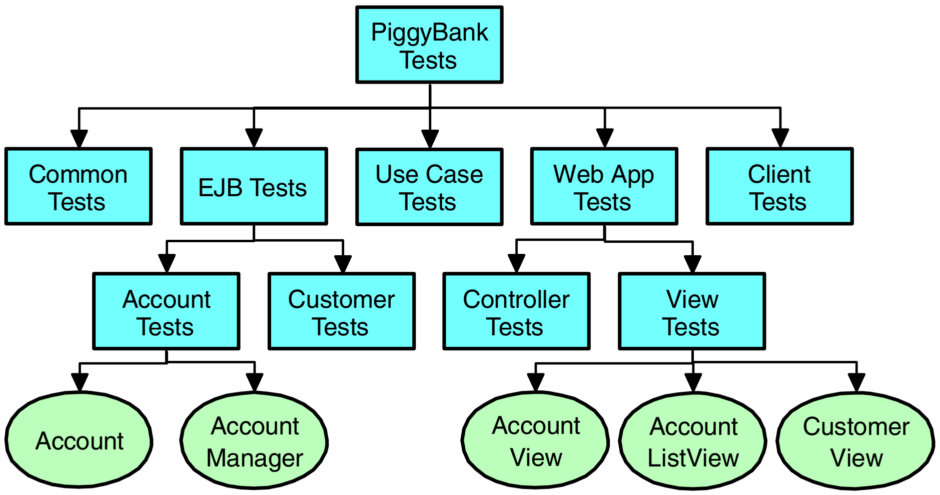


Figure 18-2 Hierarchy of test suites and test cases

Test case for a simple Java class

First we show how to create and run a test case for a simple Java class. The class we use for this example is the `AccountListView` class, which is part of the Web application module and located in the `itso.was4ad.webapp.view` package.

Expected behavior

This class is intended for use in JSPs—it manages an array of `AccountData` objects, typically obtained as a result of a call into the use case layer of the PiggyBank application.

The class is intended to allow a JSP to use the bean to iterate through each account in the list using the WebSphere `tsx:repeat` tag, extracting the information about the accounts in the list in turn without needing to code any explicit Java in the page. The page iterates through the elements in the array by invoking the `getNext` method, which can be called using the standard `jsp:getProperty` tag. When the end of the list is reached, the class throws an `ArrayIndexOutOfBoundsException`, which signals to the `tsx:repeat` code that the loop should be terminated.

An example of this usage from the page `accountDisplay.jsp` is shown in Figure 18-3.

```
<jsp:useBean id="accountList"
             class="itso.was4ad.webapp.view.AccountListView"  scope="request"/>
<P>Here are your account details:</P>
<TABLE border="1">
  <TR>
    <TD>Number</TD>
    <TD>Balance</TD>
    <TD>Type</TD>
  </TR>
  <tsx:repeat>
    <jsp:getProperty name="accountList" property="next"/>
    <TR>
      <TD><jsp:getProperty name="accountList" property="number"/></TD>
      <TD><jsp:getProperty name="accountList" property="amount"/></TD>
      <TD><jsp:getProperty name="accountList" property="type"/></TD>
    </TR>
  </tsx:repeat>
</TABLE>
```

Figure 18-3 Example usage of the AccountListView class

The current item in the list can be reset to the beginning using the `reset` method. This is useful where the same data may need to be included in a page twice, for example while building selection boxes in forms.

In addition to supporting iteration, the bean also performs formatting of the data for the Web channel, adding a currency symbol to the account balance and converting the `boolean` value indicating whether the account is a checking account into a string representing the account type, checking or savings.

Choosing what tests to write

We will write a number of tests to validate the behavior of the class:

- Test iteration through a complete data set
- Test partial iteration and reset
- Test the default no-argument constructor
- Test an attempt to use the data without first calling `getNext`

The second two tests in the list are common error situations that we foresee. We could also choose to test the formatting of data returned by the bean. We decided not to, however, because we know that the class uses the `AccountView` bean to format the data—we implement the formatting tests in the test case for that class.

At this stage we believe that this set of tests is adequate—of course there is nothing preventing us from adding more tests later if we decide we need them.

Tip: A good time to write a new test is when you find a new problem. If you write a test that reproduces the problem you can use the test to help you debug and fix it. Because we can often unit test components in isolation this can often be easier than setting up a complete test scenario for your application. It also means that you immediately notice the problem if you accidentally reintroduce it at some point in the future.

Writing the test case class

We create a new test case class `AccountListViewTests` in which to write our tests. This class extends the JUnit `TestCase` class, and is placed in the `itso.was4ad.webapp.view.tests` package. The outline of the class including the required constructor is illustrated in Figure 18-4.

```
package itso.was4ad.webapp.view.tests;

import itso.was4ad.data.*;
import itso.was4ad.webapp.view.*;
import junit.framework.*;
/**
 * JUnit tests for the AccountListView class
 */
public class AccountListViewTests extends TestCase {
    /**
     * AccountListViewTests constructor
     * @param name java.lang.String
     */
    public AccountListViewTests(String name) {
        super(name);
    }
}
```

Figure 18-4 Outline of the AccountListViewTests class

In addition to importing the package containing the JUnit framework, we also import the package containing the class we are testing, and the package `itso.was4ad.data`, which contains the `AccountData` class we also require for these tests.

Set-up and tear-down

Our tests require data to operate on—specifically they need a list of accounts in an array of `AccountData` objects. Rather than write code to create such an array in the body of each test method, we can create a method called `setUp` in our test case class. JUnit invokes this method before executing each test (Figure 18-5.)

```
/**
 * Set up some data we'll use in some of the tests
 */
public void setUp() {
    // Create some account data - make the contents predictable
    data = new AccountData[10];
    for (int i = 0; i < 10; i++) {
        data[i] =
            new AccountData(
                i % 4 + 100,                     // 4 Customer IDs
                1000 + i,                        // Unique account IDs
                i * 250,                         // Unique Balances
                (i % 2 == 0 ? true : false));   // Half checking
    }
}
```

Figure 18-5 AccountListViewTests setUp method

The `setUp` method creates an array in an instance variable `data`—we add the variable to the class as follows:

```
AccountData[] data = null;    // Data used by the tests
```

There is a corresponding `tearDown` method in which we can clean up any permanent resources such as files or database rows. Our test case does not create any permanent resources, however, so for this class we do not have to implement `tearDown`.

Testing iteration

Our first test is designed to exercise the iteration behavior—this is, after all, the primary purpose of the class. We create a new method named `testIteration`, shown in Figure 18-6.

First of all we create a new instance of the `AccountListView` class, using the data prepared by the `setUp` method. There are ten accounts in this array—we therefore expect to iterate through the accounts ten times, which we do in a `for` loop. We expect the accounts to be returned in the same order that they are specified in the original array, so we check them using the `assertEquals` method.

```
/**
 * Test the behavior when we iterate through the list
 */
public void testIteration() {
    AccountListView list = new AccountListView(data);

    // Make sure we iterate correctly through all 10 items
    for (int i = 0; i < 10; i++) {
        list.getNext();
        int accountId = 1000 + i;
        assertEquals("Item " + i + " incorrect", "" + accountId, list.getNumber());
    }

    // Past the end of the list now
    try {
        list.getNext();
        list.getNumber();
        fail("Expected ArrayIndexOutOfBoundsException");
    } catch (ArrayIndexOutOfBoundsException e) {
        // expected
    }
}
```

Figure 18-6 AccountListViewTests testIteration method

Note: Every get method in our view class returns a `String` formatted for insertion into a JSP. We must convert the account number we expect into a `String` in order to perform the comparison with the value from the view bean.

We expect the eleventh iteration to result in a `ArrayIndexOutOfBounds` exception. We test this by attempting to catch the exception. If the exception is raised when the test is executed, we simply discard it and exit the test method normally, which signals to the JUnit framework that the test was successful. If the exception is not raised, however, we cause the test to fail using the `fail` method.

The `assertEquals` and `fail` methods are provided by the JUnit framework. JUnit provides a number of methods that can be used to assert conditions and fail a test if the condition is not met. These methods are inherited from the class `junit.framework.Assert`, via `TestCase`, and are summarized in Table 18-1.

All of these methods include an optional `String` parameter that allows the writer of a test to provide a brief explanation of why the test failed—this message is reported along with the failure when the test is executed.

Table 18-1 JUnit assert methods

Method name	Description
`assertEquals`	Assert that two objects or primitives are equal. Compares objects using `equals`, and compares primitives using `==`.
`assertNotNull`	Assert that an object is not null
`assertNull`	Assert that an object is null
`assertSame`	Assert that two objects refer to the same object. Compares using `==`.
`assertTrue`	Assert that a `boolean` condition is `true`
`fail`	Fails the test

Testing reset

The next test tests the reset behavior. It also uses the array created by the `setUp` method, using the `data` instance variable to create an instance of our `AccountListView` class. It then iterates part-way through the list, checking the account numbers on the way, just to make sure we aren't stuck on the first item. We then invoke the `reset` method, and assert that the next account number returned by the bean is the number of the first account in the list. If this is the case the test is complete and we exit the test method normally.

The code for the `testReset` method is shown in Figure 18-7.

```
/**
 * Test the reset method
 */
public void testReset() {
    AccountListView list = new AccountListView(data);

    // Make sure we iterate correctly through the first 5 items
    for (int i = 0; i < 5; i++) {
        list.getNext();
        int accountId = 1000 + i;
        assertEquals("Item " + i + " incorrect", "" + accountId, list.getNumber());
    }

    // Now reset the view and make sure we're back at the beginning
    list.reset();
    list.getNext();
    assertEquals("Reset incorrect", "1000", list.getNumber());
}
```

Figure 18-7 AccountListViewTests testReset method

Testing the default constructor

The next test we write tests the behavior of the class when a new instance is created using the default no-argument constructor. This is important for a class intended to be passed to a JSP because the JSP uses the default constructor to create a new instance of the bean if the `useBean` tag is used with the `class` attribute, but no bean exists in the specified scope.

This could happen if a user attempts to access a JSP page directly, instead of via the appropriate servlet, for example. Under these circumstances we would like the page to behave gracefully, rather than fail with a `NullPointerException`. The code for the `testDefaultConstructor` method is shown in Figure 18-8.

```
/**
 * Test the behavior with the default constructor
 */
public void testDefaultConstructor() {
    AccountListView list = new AccountListView();
    try {
        list.getNext();
        list.getCustomerID();
        fail("Expected ArrayIndexOutOfBoundsException");
    } catch (ArrayIndexOutOfBoundsException e) {
        // expected
    }
}
```

Figure 18-8 AccountListView testDefaultConstructor method

We expect the class to allow an instance of the bean to be created, but to throw an `ArrayIndexOutOfBounds` exception when an attempt is made to use it—this immediately terminates any enclosing `tsx:repeat` loop.

Our test code creates a new instance of the class using the default constructor, then attempts to use it. If we catch the expected exception, all is well. If we do not catch the exception, we cause the test to fail by invoking the JUnit `fail` method.

Testing iteration without an initial getNext

The last of our tests investigates what happens when a JSP attempts to obtain data from our bean without first invoking the `getNext` method. We include this test because we anticipate that this is a mistake many JSP developers may make.

The behavior under these circumstances is undefined by our design—we did not consider this sequence of events until we started thinking about how to break our class. This usefully illustrates one of the benefits of our unit testing strategy—sooner or later a JSP developer is likely to make this mistake but at least now we know we can cope with it.

We expect the bean to handle this situation gracefully—it throws an `ArrayIndexOutOfBounds` exception, which is a reasonable thing to do. Having highlighted the issue, however, we may consider altering our design so that the bean throws an `InvalidOperation`, with a message explaining why—this may provide more assistance in debugging the broken JSP.

The code for the `testNoGetNext` method is shown in Figure 18-9.

```
/**
 * Test the behavior without an initial getNext()
 */
public void testNoGetNext() {
    AccountListView list = new AccountListView(data);
    try {
        list.getCustomerID();
        fail("Expected ArrayIndexOutOfBoundsException");
    } catch (ArrayIndexOutOfBoundsException e) {
        // expected
    }
}
```

Figure 18-9 AccountListViewTests testNoGetNext method

Building the tests

We compile our test case class along with the Web application code it tests. The only change we must make in order to compile the test code is to add the JUnit JAR to the compiler class path. The archive is located in the JUnit installation directory—in our case the full path to the JAR file is:

```
D:\junit3.7\junit.jar
```

If you are using Ant to build the application as described in “Using Ant to build a WebSphere application” on page 197, for example, you need to add the JUnit JAR file to the class path specified in the Web application `build.xml` build file, as Figure 18-10 illustrates.

```
 <path id="webapp.classpath">
    <pathelement location="${global.was.dir}/lib/j2ee.jar"/>
    <pathelement location="${global.junit.jar}"/>
    <pathelement path="${global.build.dir}/common"/>
    <pathelement path="${global.build.dir}/usecase"/>
  </path>
```

Figure 18-10 Updating the Web application build file to build the test case class

The `global.junit.jar` property defines the location of the JAR file—we added the property to the `global.properties` file (Figure 18-11).

```
global.junit.dir=D:/junit3.7
global.junit.jarfile=junit.jar
global.junit.jar=${global.junit.dir}/${global.junit.jarfile}
```

Figure 18-11 Updating the Ant global.properties file to specify JUnit file locations

If you are developing using VisualAge for Java and imported the JUnit code into the workspace as described in "Installing JUnit in VisualAge for Java" on page 521, you do not have to make any further changes in order to compile the code, because VisualAge for Java locates the JUnit classes in the workspace.

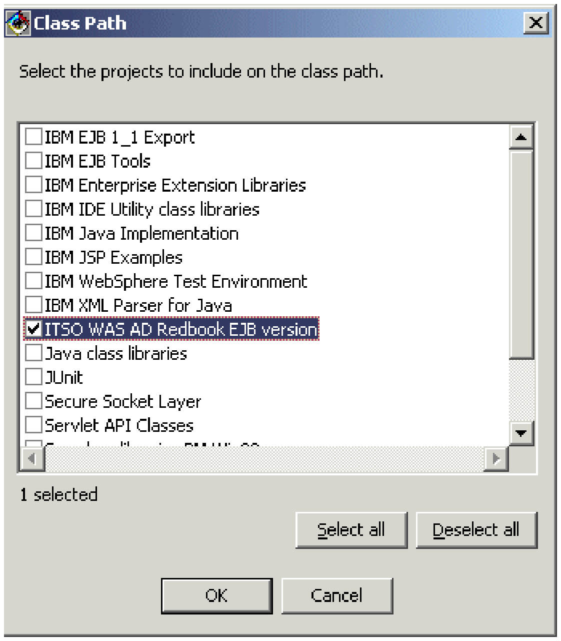
Running the tests

The JUnit framework provides both text and GUI test runner tools that can your tests and report the results. We illustrate how to execute the tests in out test case using a text-based user interface and a Swing GUI.

Before we can run either tool, we must first make sure that all the classes we need are on our class path. In this case, in order to run the tests in the class `AccountListViewTest` we need the JUnit code plus the PiggyBank Web application and common code in our class path.

To run the tools from VisualAge for Java, we must add the projects containing the code we want to test to the class path for each tool—we can do this by locating the tool runner class in the VisualAge GUI and selecting *Properties* from the context menu of the class. We then select the *Class Path* tab in the properties dialog. We then click the *Edit* button and select the appropriate project (Figure 18-12).

We also use the properties dialog to specify the command-line arguments to pass to the tool runner class—we can set them in the dialog as described, or have the dialog pop-up when we run the class by selecting *Run -> Run main with* from the context menu.


Class Path

Select the projects to include on the class path.

- [ ] IBM EJB 1_1 Export
- [ ] IBM EJB Tools
- [ ] IBM Enterprise Extension Libraries
- [ ] IBM IDE Utility class libraries
- [ ] IBM Java Implementation
- [ ] IBM JSP Examples
- [ ] IBM WebSphere Test Environment
- [ ] IBM XML Parser for Java
- [✓] ITSO WAS AD Redbook EJB version
- [ ] Java class libraries
- [ ] JUnit
- [ ] Secure Socket Layer
- [ ] Servlet API Classes

1 selected

Figure 18-12 Adding a project to the TestRunner class path in VisualAge for Java

Running the text-based test runner

The text-based test runner included with JUnit is started from the `junit.textui.TestRunner` class. We can run the tests in our test case class `AccountListViewTests` by passing the name of the class to the test runner. Figure 18-13 shows the test runner being invoked from the command line using the `java` command. The command is entered on a single line—it has been split onto two lines in the example for formatting purposes.

```
D:\itso4ad\dev\src>java junit.textui.TestRunner
                        itso.was4ad.webapp.view.tests.AccountListViewTests
....
Time: 0.03

OK (4 tests)
```

Figure 18-13 Running the text-based test runner from the command line

Each dot (`.`) output by the tool represents the start of a test. We have four tests in our test case, so there are four dots. Once all the tests are complete the test runner tells us how long they took, and summarizes the results—in this case all of our tests were successful.

Running the Swing-based test runner

The Swing-based test runner included with JUnit is started from the `junit.swingui.TestRunner` class. We also supply the name of the class to test on the command-line with this test runner, for example:

```
java junit.swingui.TestRunner itso.was4ad.webapp.view.tests.AccountListViewTests
```

When we execute the Swing test runner the GUI is displayed and the tests in our test case executed.

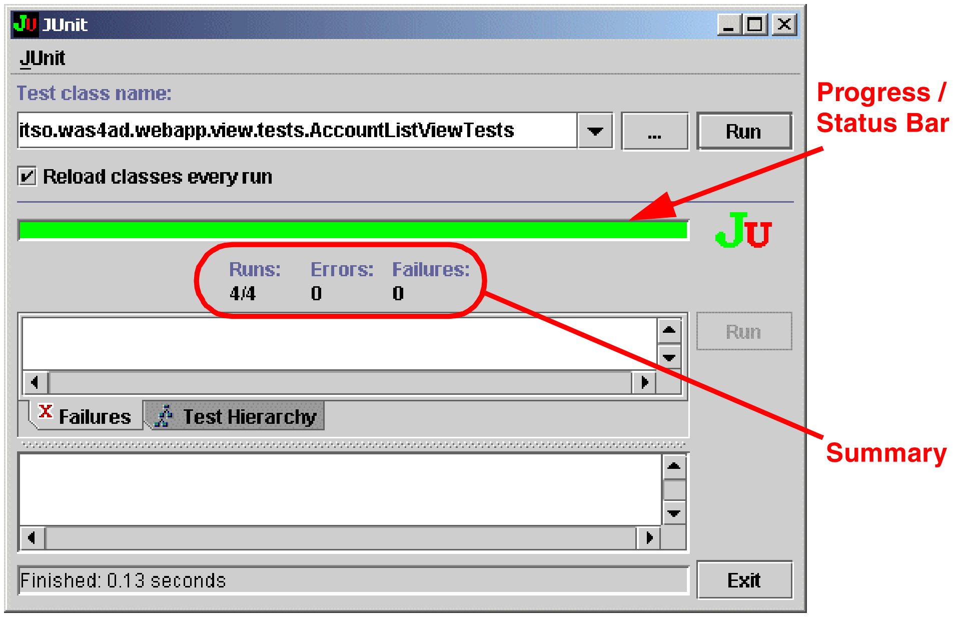


Figure 18-14 Swing-based test runner

In Figure 18-14 we can see the results of our tests on the `AccountListView` class. The progress bar is green, which indicates that all of the tests ran successfully—any failures would have been indicated by a red bar. This observation is confirmed by the result summary—it reports that of the four tests run, none failed, and none resulted in an error.

Tip: A test is considered to be *successful* if the test method returns normally. A test *fails* if one of the methods from the `Assert` class signals a failure. An *error* indicates that an unexpected exception was raised by the test method, or the `setUp` or `tearDown` method invoked before or after it.

We can re-run the test by clicking *Run*. This is especially useful in a VisualAge for Java environment, because any code changes we make are automatically picked up without restarting the tool.

Note: The JUnit test runner is also able to reload classes—it achieves this using a specialized class loader. While this works well for simple tests, as we will see later on, we are unable to use this facility when testing code running in a WebSphere container, because WebSphere uses its own specialized class loaders.

Failed tests

So far we have not seen any failed tests, although we can assure you this was not the case when we were initially developing this chapter. At this point we introduce an intentional defect into our code in order to demonstrate this scenario.

We introduce the defect by editing the default constructor of the ArrayListView class, commenting out the line that invokes another constructor with an empty array. The modified code is shown highlighted in Figure 18-15.

```
/**
 * AccountListView default constructor
 */
public AccountListView() {
       //this(new AccountData[0]);
}
```

Figure 18-15 Introducing a defect in the AccountListView class

We recompile this class—in VisualAge this simply involves saving the modified method—and re-run our tests.

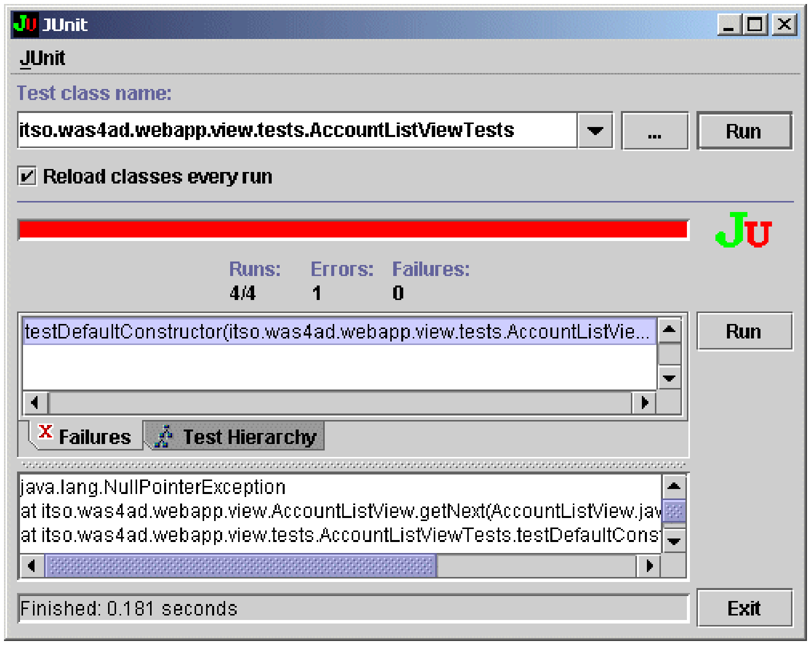
The output from the text-based test runner is shown in Figure 18-16. It still shows four dots, because four tests were attempted. After the last dot however, we see an `E`. This character represents an error—if the test had been failed by one of the methods from `Assert` we would have seen `F` instead.

After the tests complete we get a summary of the failures—in this case there is just one—it tells us there was a `NullPointerException` when we tested the default constructor.

```
D:\itso4ad\dev\src>java junit.textui.TestRunner
                        itso.was4ad.webapp.view.tests.AccountListViewTests
....E
Time: 0.03
There was 1 error:
1) testDefaultConstructor(itso.was4ad.webapp.view.tests.AccountListViewTests)
java.lang.NullPointerException
        at itso.was4ad.webapp.view.AccountListView.getNext(AccountListView.java:59)
        at itso.was4ad.webapp.view.tests.AccountListViewTests.testDefaultConstructor(
AccountListViewTests.java:39)

FAILURES!!!
Tests run: 4,  Failures: 0,  Errors: 1
```

Figure 18-16   Test failure in the text-based test runner

Figure 18-17 shows the same failure in the Swing-based GUI—the progress bar is red (this can be seen more clearly in the PDF version of this book), and the summary reports a single error. The description panel at the bottom of the window shows the `NullPointerException` relating to the failure highlighted in the center panel.



itso.was4ad.webapp.view.tests.AccountListViewTests

☑ Reload classes every run

Runs: 4/4   Errors: 1   Failures: 0

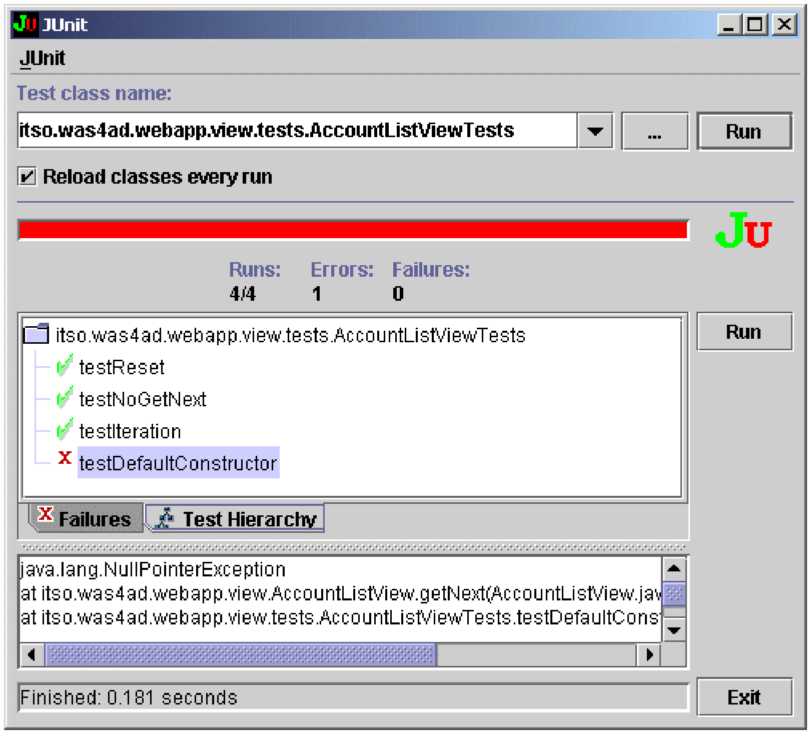
testDefaultConstructor(itso.was4ad.webapp.view.tests.AccountListVie...

```
java.lang.NullPointerException
at itso.was4ad.webapp.view.AccountListView.getNext(AccountListView.jav
at itso.was4ad.webapp.view.tests.AccountListViewTests.testDefaultCons
```

Finished: 0.181 seconds

Figure 18-17   Test failure in the Swing-based test runner

Clicking on *Test Hierarchy* in this window displays the tree-based view of the hierarchy of tests shown in Figure 18-18—this view shows the tests that passed and failed using icons. The hierarchy view becomes more useful when we organize our test case classes into suites.



```
itso.was4ad.webapp.view.tests.AccountListViewTests
  testReset
  testNoGetNext
  testIteration
  testDefaultConstructor
```

```
java.lang.NullPointerException
at itso.was4ad.webapp.view.AccountListView.getNext(AccountListView.jav
at itso.was4ad.webapp.view.tests.AccountListViewTests.testDefaultCons
```

```
Finished: 0.181 seconds
```

Figure 18-18 Displaying the test hierarchy in the Swing-based test runner

Adding the test case to a test suite

The next step we will demonstrate is the assembly of a number of tests into a test suite, which we can then organize into a hierarchy as we described earlier in "Test suites" on page 522.

Writing the AllTests class

The first step is to create a new class named `AllTests` in the same package as our test case classes, `itso.was4ad.webapp.view.tests`. This new class also extends the TestCase class—Figure 18-19 shows the outline of the class source.

```
package itso.was4ad.webapp.view.tests;

import junit.framework.*;
/**
 * Runs all of the tests in this package
 */
public class AllTests extends TestCase {
   /**
    * AllTests constructor
    * @param name java.lang.String
    */
   public AllTests(String name) {
      super(name);
   }
}
```

Figure 18-19 Outline of the AllTests class

The next step is to define a `suite` method that returns a `TestSuite` object containing all of the tests in the suite. We create a new instance of `TestSuite`, providing a descriptive name for the suite in the constructor (Figure 18-20).

```
/**
 * Returns a test suite containing all tests in this package
 * @return junit.framework.Test
 */
public static Test suite() {
   TestSuite suite = new TestSuite("All web application view tests");

   // Add any new tests here
   suite.addTestSuite(CustomerViewTests.class);
   suite.addTestSuite(AccountViewTests.class);
   suite.addTestSuite(AccountListViewTests.class);

   // Return the test suite
   return suite;
}
```

Figure 18-20 Suite method source

We then add tests to the suite using the `addTestSuite` method. This method takes a `Class` object as a parameter and adds all the methods in the class with names that begin with `test` to the suite—this is the reason why we followed this convention when we created the methods.

We have two other tests in this particular package, in addition to the `AccountListViewTests` class we developed earlier.

We also have tests that exercise the `CustomerView` and `AccountView` classes in the same package. The tests in all three test case classes are added to the test suite, which is then returned as the result of the `suite` method. Any new test cases we create in this package are added to the suite by adding a corresponding `addTestSuite` call to the `suite` method.

For convenience we also create a `main` method in the `AllTests` class. This is a simple shortcut that allows us to execute the tests by invoking the class directly, rather than passing the class name to a test runner (Figure 18-21).

```
/**
 * Run this test suite
 * @param args java.lang.String[]
 */
public static void main(String[] args) {
   junit.textui.TestRunner.run(suite());
}
```

Figure 18-21 AllTests main method

Running the test suite

This `main` method starts a text-based test runner by passing it the TestSuite returned by the suite method—it would be just as simple to start one of the GUI test runners, however. We can now run all the tests in the suite using the command shown in Figure 18-22.

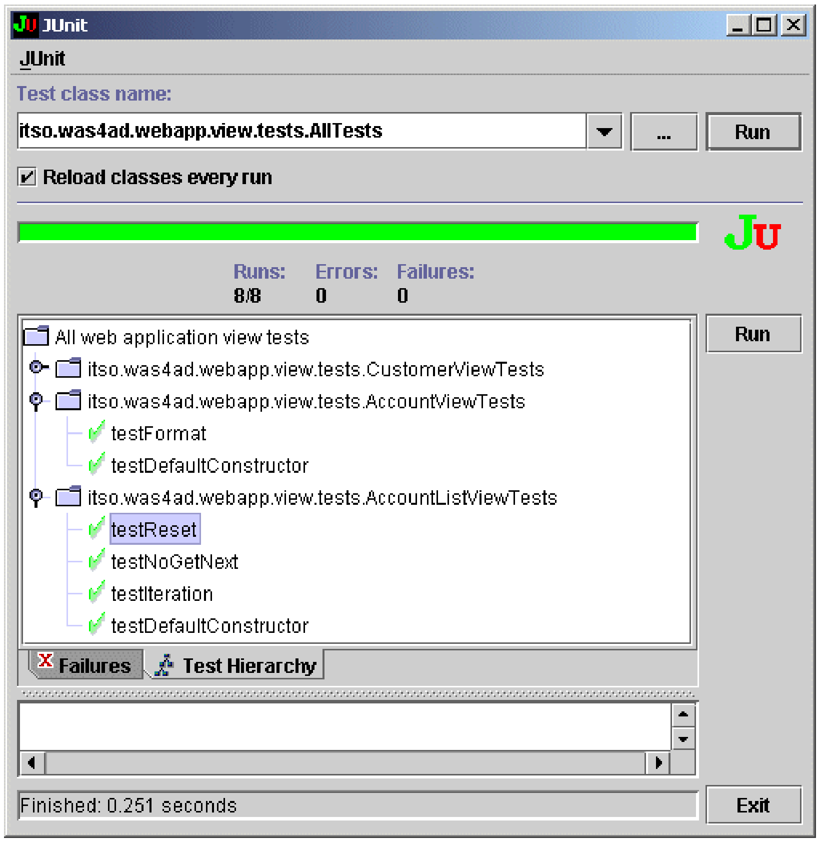
```
D:\itso4ad\dev\src>java itso.was4ad.webapp.view.tests.AllTests
........
Time: 0.03

OK (8 tests)
```

Figure 18-22 Using the main method to run the test suite

We can also run the tests in the Swing GUI by issuing the command:

```
   java junit.swingui.TestRunner itso.was4ad.webapp.view.tests.AllTests
```

The GUI displaying the hierarchy of test results from the test suite is shown in Figure 18-23.



```
All web application view tests
    itso.was4ad.webapp.view.tests.CustomerViewTests
    itso.was4ad.webapp.view.tests.AccountViewTests
        testFormat
        testDefaultConstructor
    itso.was4ad.webapp.view.tests.AccountListViewTests
        testReset
        testNoGetNext
        testIteration
        testDefaultConstructor
```

Finished: 0.251 seconds

Figure 18-23 Running a test suite in the Swing GUI test runner

Creating the hierarchy

We complete the hierarchy of test cases by creating `AllTests` classes in the packages `itso.was4ad.webapp.tests`, and `itso.was4ad.tests`. These suites correspond to all the code in the Web application module, and the entire PiggyBank application respectively.

The classes are almost identical, however instead of adding tests to the suite using `addTestSuite`, they use the `addTest` method (remember `TestSuite` implements the `Test` interface), passing the output of each `AllTests.suite` method as a parameter, for example:

```
suite.addTest(itso.was4ad.webapp.view.tests.AllTests.suite());
```

Test case for an EJB

We've already seen how to test a relatively simple component—now we look at how to use JUnit to test an EJB component. We use the `Account` EJB for our example—this is a container managed persistent (CMP) entity bean, although many of the issues we discuss are appropriate to both entity and session beans.

Writing the test case

We create a new class for out tests named `AccountTests`. The classes that make up the account EJB are located in the `itso.was4ad.ejb.account` package, so we place our test case class in a package named `itso.was4ad.ejb.account.tests`. The outline of the class code is included in Figure 18-24.

```
package itso.was4ad.ejb.account.tests;

import junit.framework.*;
import itso.was4ad.helpers.HomeHelper;
import itso.was4ad.ejb.account.*;
import itso.was4ad.data.*;
import itso.was4ad.exception.*;
import javax.naming.*;
import javax.ejb.*;
import javax.rmi.*;
import java.rmi.*;
import java.util.*;
/**
 * JUnit tests for the Account EJB
 */
public class AccountTests extends TestCase {
   private static final String ACCOUNT_HOME = "java:comp/env/ejb/Account";
   /**
    * AccountTests constructor
    * @param name java.lang.String
    */
   public AccountTests(String name) {
      super(name);
   }
}
```

Figure 18-24 AccountTests class source code outline

The list of `import` statements is relatively large compared to our earlier example—this is partly because the interface to the EJB uses more code from our application, but also because our test case class is an EJB client. Note also the constant defining the JNDI name we use to locate the EJB.

Running as an EJB client

All of our tests are going to run as clients of our EJB, with the EJB running in WebSphere. This approach allows us to test the behavior of the EJB from a client's perspective, which is after all how it will be used.

This is not the only choice however—we could, for example, test the class from the container's perspective, where the test creates an instance of the EJB implementation class directly and invokes EJB life-cycle methods such as `ejbActivate` directly.

This requires a little more effort but may pay dividends. With CMP beans you could eliminate the need for a persistent store completely, setting instance variables yourself using the Java reflection API. With bean-managed persistent (BMP) beans on the other hand you may want to exercise the persistence code more thoroughly, running tests that check the data written to the persistent store directly instead of through the EJB.

Because all our tests have to locate the account EJB home, we created a simple helper method in our test case class that test methods can use to easily obtain the home interface. The code for the `getAccountHome` method is shown in Figure 18-25.

```
/**
 * Helper method that locates the Account EJB's home interface
 * @return itso.was4ad.ejb.account.AccountHome
 * @exception javax.naming.NamingException
 */
private static AccountHome getAccountHome() throws NamingException {
    InitialContext context = new InitialContext();
    return (AccountHome) PortableRemoteObject.narrow(
          context.lookup(ACCOUNT_HOME),
          AccountHome.class);
}
```

Figure 18-25 getAccountHome helper method

Set-up and tear-down

Because the EJB is an entity bean, most of our test cases require data on which to operate—the exceptions are those that test the creation of data. We use the `setUp` and `tearDown` methods to create and remove our test data.

We have two choices when it comes to managing the data in the database—we can either use the EJB to manage the data, or code JDBC directly to the database. Both options have their advantages and disadvantages. If we only ever use the bean to access the database, for example, we may not find certain types of persistence problem. On the other hand, using the EJB itself is much easier, and the set-up and tear-down become extensions of the test case. In the end we decided to use the EJB in the interests of expediency.

Figure 18-26 shows the `setUp` method.

```
/**
 * Set up some data used by the tests
 */
public void setUp() throws
   NamingException, CreateException, RemoteException, InvalidOperation {

   // Get the home interface
   AccountHome home = getAccountHome();

   // Create some accounts and put money in some of them
   home.create(1000, 100, false);
   home.create(1001, 101, false).credit(1000);
   home.create(1002, 102, true).credit(10000);
   home.create(1003, 102, false).credit(2500);
   home.create(1004, 103, true).credit(10);

}
```

Figure 18-26 AccountTests setUp method

We need to take extra care with the tear-down method—it needs to be robust enough to cope with anything the tests we write might do to the data. We must take care to ensure that the test data is always removed—if for some reason an account is not removed the next test will fail as our set-up method, which is admittedly not particularly robust, will throw a `DuplicateKeyException` before we even get to run the test.

Our solution (Figure 18-27), is to attempt to remove any account with a number in the range from 1000 to 1009. This includes the accounts we create in the set-up method, but also allows for tests to create their own accounts. If an account does not exist, we simply ignore it and carry on, because tests may also remove accounts.

Finally, we must make sure the accounts are empty before we remove them, because the application enforces a business rule that does not allow us to remove an account that has a balance.

```
/**
 * Delete the data used by the tests. CAUTION This method
 * removes any account with a number in the range 1000 - 1009!
 */
public void tearDown() throws NamingException, RemoteException {
    // Get the home interface
    AccountHome home = getAccountHome();

    // Delete any accounts left lying around
    for (int i = 1000; i < 1010; i++) {
        try {
            // Look for the account
            AccountKey key = new AccountKey(i);
            Account acct = home.findByPrimaryKey(key);

            // Empty the account and remove it
            int balance = acct.getAccountData().getAmount();
            if (balance != 0) {
                acct.debit(balance);
            }
            acct.remove();
        } catch (Exception e) {
            //Ignore
        }
    }
}
```

Figure 18-27 AccountTests tearDown method

Writing the test methods

Because we are testing an entity EJB we must test both the home and remote interfaces. The tests we perform on the home interface are listed in Table 18-2.

Table 18-2 Methods testing the Account EJB home interface

Test method	Description
`testCreate`	Test creation of a new account
`testFindByCustomerID`	Test finder that returns a collection of accounts
`testRemove`	Test the straightforward removal of an account with no balance
`testRemoveWithBalance`	Test the removal of an account which has an outstanding balance—this is not allowed by the business rules that govern the application

The tests on the remote interface of the `Account` EJB are listed in Table 18-3.

Table 18-3 Methods testing the Account EJB remote interface

Test method	Description
`testCredit`	Test a straightforward credit to the account
`testDebit`	Test a straightforward debit from the account
`testDebitOverdraw`	Test a debit that takes more money from the account than is available—our simple application does not provide an overdraft facility
`testIsOwnedBy`	Test that we can correctly determine whether a particular customer owns an account
`testNegativeCredit`	Test a credit of a negative amount to an account
`testNegativeDebit`	Test a debit of a negative amount from an account

The individual test methods do not introduce any new JUnit features, so we only include a single example here, `testDebitOverdraw` (Figure 18-28).

```
/**
 * Test debit of more money than is in the account
 */
public void testDebitOverdraw() throws NamingException, FinderException,
RemoteException, BusinessException {
    // Locate one of the previously set up accounts
    AccountKey key = new AccountKey(1004);
    Account account = getAccountHome().findByPrimaryKey(key);

    // Get the balance
    int balance = account.getAccountData().getAmount();

    // Attempt to debit more than we have
    try {
       account.debit(balance + 1);
       fail("Should throw InsufficientFunds");
    } catch (InsufficientFunds e) {
       // expected
    }

    // Make sure the balance is the same
    assertEquals("Balance changed after InsufficientFunds", balance,
account.getAccountData().getAmount());
}
```

Figure 18-28 testDebitOverdraw method in the AccountTests class

Updating the test suite hierarchy

We want to include our new EJB test in our test suite hierarchy. We do this by creating `AllTests` classes in the `itso.was4ad.ejb.account.tests` and `itso.was4ad.ejb.tests` packages, as described in "Adding the test case to a test suite" on page 536. We also add the EJB tests to the `suite` method in the top-level `itso.was4ad.tests.AllTests` class.

Building the tests

As with the Web application tests, because we compile the EJB tests at the same time as the EJB code, all we have to do to compile our tests is to add the `junit.jar` archive to the compiler class path.

Packaging the tests

Our `AccountTest` test case is an EJB client so we must execute it in an environment where it has access to the services provided by the WebSphere container. The specific requirement is that we be able to perform the lookup on the EJB using the JNDI name `java:comp/env/ejb/Account`.

Because the JUnit test runners are standalone Java programs, the easiest way to do this is to create a new J2EE client module and package the tests into it. We can then run the tests using the WebSphere client container `launchClient`. We chose to achieve this by updating the `main` method of our top-level AllTests class to start the test runner for us (Figure 18-29).

```
/**
 * Run all the PiggyBank tests
 * Usage: itso.was4ad.tests.AllTests [-gui]
 * @param args java.lang.String[]
 */
public static void main(String[] args) {
    // Check the command line args
    if (args.length > 0 && args[0].equals("-gui")) {
        // Run the GUI tool - tell it to use the system classloader
        junit.swingui.TestRunner runner = new junit.swingui.TestRunner();
        runner.setLoading(false);
        runner.start(new String[] {AllTests.class.getName()});
    } else {
        // Run the text version
        junit.textui.TestRunner.run(suite());
    }
}
```

Figure 18-29 Top-level AllTests main method

We update the method to check for the `-gui` flag in the command-line argument. If the flag is present we start the Swing GUI test runner, otherwise we start the text-based test runner. If we are using the GUI test runner we explicitly set a flag disabling JUnit's class reloading facility—this uses a class loader that is unable to locate our test classes when we package them in an EAR file.

We use AAT to create a new client module, specifying the top-level `AllTests` class as the main class—this is the only class we want to include in the module, because it does not belong to any one component. We also define the local EJB JNDI reference used to locate the account EJB ,and bind it to the same global JNDI name specified in the bindings for our EJB module.

For simplicity we add the test client module to our single application EAR file. This allows us to use the class path defined in the test module's JAR manifest to include the common, use case, and EJB JAR files in the test client's class path.

This approach does not work for the Web application, however, because the Web application classes are located in `WEB-INF/classes` in the WAR file. We worked around this by explicitly including all of the Web application classes in the test client module JAR file. This is less than ideal because now we have two copies of these classes in our EAR archive. It does not affect the execution of the actual application code, however, so we decided to live with the situation.

The final EAR file and the code used to create it are included in the additional material, described in Appendix A, "Additional material" on page 557.

Running the tests

At this point we now have a single EAR file containing all of our application code—including the test case classes, and out client module that we use to run the tests.

Before we can run the tests, however, we must deploy the application into an application server and start it. This is described in Chapter 16, "Deploying to the test environment" on page 431.

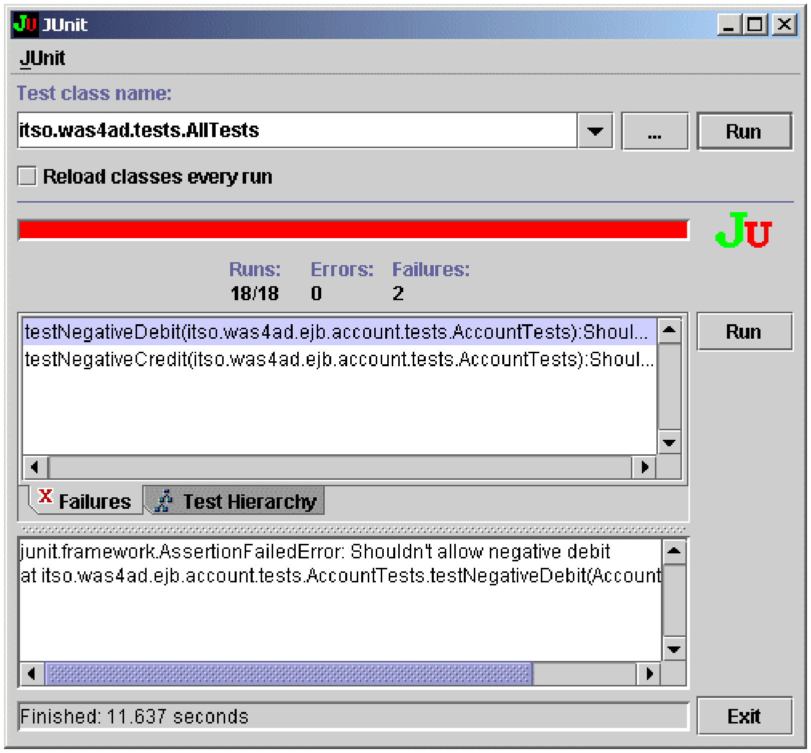
Once the application is up and running in WebSphere, we are finally able to start our tests. We invoke the test runner using the WebSphere `launchClient` command. There are now two client modules in the EAR file—the standalone PiggyBank Swing client and our test module.

If you simply run the standard `launchClient` command without specifying which client you want to run, WebSphere appears to always execute the one that is described first in the enterprise application's deployment descriptor.

The first application in our case is the PiggyBank client, therefore, to run the our unit tests we have to use the -CCjar option to tell the client container which client to run:

```
launchclient piggybank.ear -CCjar=piggybank-test.jar -gui
```

This command runs all of our unit tests using the Swing GUI (Figure 18-30).



Test class name: itso.was4ad.tests.AllTests

Runs: 18/18 Errors: 0 Failures: 2

testNegativeDebit(itso.was4ad.ejb.account.tests.AccountTests):Shoul...
testNegativeCredit(itso.was4ad.ejb.account.tests.AccountTests):Shoul...

junit.framework.AssertionFailedError: Shouldn't allow negative debit
at itso.was4ad.ejb.account.tests.AccountTests.testNegativeDebit(Account

Finished: 11.637 seconds

Figure 18-30 Running all PiggyBank unit tests in the Swing GUI

As you can see, two of our tests failed—it seems our PiggyBank application is quite happy to allow bank accounts to be credited and debited negative amounts. Fortunately this is just an example application, so in time-honored fashion, we leave the correction of our code as an exercise for the reader.

We can re-run all the tests by clicking the uppermost *Run* button. If we simply want to rerun one of the failing tests, we select the test in the center panel and click the lower *Run* button. This gives us the opportunity to diagnose the problem using the configurable logging mechanism described in “Message logging” on page 330 to dynamically turn on debug messages for the offending component.

Automating unit testing using Ant

We are now at the stage where we have created some test cases for our application, and we know how to run them. This is a very good start, but it is essential to remember that tests are only useful if they are run. One way to make sure that this happens is to schedule an automatic process that runs through all of our tests, perhaps in conjunction with an automatic build process as described in “Automatic builds” on page 225.

In this section we describe how we can extend the Ant build scripts we developed in “Using Ant to build a WebSphere application” on page 197 to automatically install the application into WebSphere AEs, start the application server, and run our unit tests. We chose AEs for these examples because AEs is ideal for use as a unit test environment on a developer’s desktop machine—the scripts we develop can be used during day-to-day development as well as by the automatic daily build process.

All of the code we developed for this example is available in the additional material—see Appendix A, “Additional material” on page 557.

Building the tests

In “Packaging the tests” on page 545 we described how we created an application client JAR for the test runner, and included it in our EAR file. We must extend our Ant build scripts to achieve the same thing.

Creating the directory structure

The first thing to do is create a new subdirectory in our source tree for our test module. We named our directory `test`, locating it with our other subproject directories in `D:\ITSO4AD\dev\src`. Into this directory we place a single Java source file for the top-level `AllTests` class, `itso\was4ad\tests\AllTests.java`. We also created a `META-INF` directory for the client meta-data, which we extract from the JAR file created by AAT using the Java SDK `jar` command.

Creating the build file

Next we must create a build file, `build.xml`, for Ant to use to build the test module. We start with the same basic build file we used before, and customize it for the test module.

First we update the local properties and path—the class path for the compile must include all of our subprojects because the `AllTests` class refers to them all in order to build a suite of all tests. The changes are shown in Figure 18-31.

```
<!-- Set up local properties and paths-->
 <property name="test.build.dir" value="${global.build.dir}/test"/>
 <property name="test.jar.name" value="piggybank-test.jar"/>
 <property name="test.jar.file"
           value="${global.module.dir}/${test.jar.name}"/>
 <path id="test.classpath">
   <pathelement location="${global.junit.jar}"/>
   <pathelement path="${global.build.dir}/common"/>
   <pathelement path="${global.build.dir}/ejb"/>
   <pathelement path="${global.build.dir}/usecase"/>
   <pathelement path="${global.build.dir}/webapp"/>
   <pathelement path="${global.build.dir}/client"/>
 </path>
```

Figure 18-31 Local properties and path from the test module build file

The only other significant change we have to make is to the `package` target. We must create a JAR file that includes not just the `AllTests` class and client meta-data, but also all the compiled classes from the Web application, as described in "Packaging the tests" on page 545. The resulting `package` target is shown in Figure 18-32.

```
<target name="package" depends="init,compile">
   <echo>Packaging ${ant.project.name}</echo>
   <mkdir dir="${global.module.dir}"/>
   <jar jarfile="${test.jar.file}"
        manifest="META-INF/MANIFEST.MF"
   >
     <fileset dir="${test.build.dir}"/>
     <fileset dir="${basedir}">
       <include name="META-INF/*"/>
       <exclude name="META-INF/MANIFEST.MF"/>
     </fileset>
     <!-- Hack for webapp files -->
     <fileset dir="${global.build.dir}/webapp"/>
   </jar>
   <echo>Finished packaging ${ant.project.name}</echo>
 </target>
```

Figure 18-32 Package target from the test module build file

Updating the master build file

The next step involves updating the master build file, `build.xml` located in the `D:\ITSO4AD\dev\src` directory. First of all we must update the `compile`, `package` and `clean` targets to delegate to the test subproject's build file. We add the test subproject so that its targets always execute last (Figure 18-33).

```
<target name="package" depends="init">
   <echo>Packaging ${ant.project.name}</echo>
   <ant dir="common" target="package"/>
   <ant dir="ejb" target="package"/>
   <ant dir="usecase" target="package"/>
   <ant dir="client" target="package"/>
   <ant dir="webapp" target="package"/>
   <ant dir="test" target="package"/>
   <ant dir="ear" target="package"/>
   <echo>Finished packaging ${ant.project.name}</echo>
 </target>
```

Figure 18-33 Updated master build file package target

We also update the `itso4ad.source.path property` so that the `document` target picks up the source code in the test subproject when it generates documentation.

Running the tests

To run the tests using Ant we must create targets to perform the following actions:

- Start WebSphere Application Server
- Stop WebSphere Application Server
- Run the unit tests

We already have targets that create and install an EAR file, as described in “Packaging the EAR file” on page 230 and “Installing the EAR file” on page 234.

Starting and stopping WebSphere Application Server

We create two new targets in the master build file to start and stop WebSphere AEs.

The `start` target depends upon the `stop`, `package` and `install` targets—by executing this target we first stop WebSphere if it is running, rebuild the application EAR file if necessary, and install it into the application server. Only then do we actually start the server. This sequence allows a developer to refresh the unit test environment with the latest code by issuing a single Ant command.

The `start` target starts the application server using the WebSphere `startServer` script—this script starts AEs in the background (Figure 18-34). The target has only been tested on Windows systems and will only work on that platform. It should prove relatively simple to update for other platforms, however.

```
<target name="start" depends="init,stop,package,install">
   <echo>Starting WebSphere AEs</echo>
   <exec executable="${global.was.dir}/bin/startServer.bat"
         os="Windows NT,Windows 2000"
   />
   <echo>WebSphere AEs started</echo>
 </target>

 <target name="stop" depends="init">
   <echo>Stopping WebSphere AEs</echo>
   <exec executable="${global.was.dir}/bin/stopServer.bat"
         os="Windows NT,Windows 2000"
   />
   <echo>WebSphere AEs stopped</echo>
 </target>
```

Figure 18-34 Master build file start and stop targets

The stop target is very similar, although it does not depend on any targets other than the standard `init` target. It uses the WebSphere `stopServer` command to stop the application server. An example of the output generated by this target is shown in Figure 18-35

```
stop:
     [echo] Stopping WebSphere AEs
     [exec] IBM WebSphere Application Server
     [exec] Command Line Runtime Utility Program
     [exec] Copyright (C) IBM Corporation, 2001
     [exec]
     [exec] Loading configuration from file.
     [exec] Using the specified configuration file:
     [exec]     D:\WebSphere\AppServer\config\server-cfg.xml
     [exec] The diagnostic host name was read as "localhost".
     [exec] The diagnostic port was read as "7000".
     [exec] Issuing command to stop server.
     [exec] The stop server command completed successfully.
     [exec] Examine the server log files to verify that the server has stopped.
     [echo] WebSphere AEs stopped
```

Figure 18-35 Stopping WebSphere AEs using the stop target

Running the tests

We also create two new targets to run the JUnit tests in the test module—`test` runs the unit tests using the text-based test runner, whereas `testgui` runs the Swing GUI test runner. The two targets are shown in Figure 18-36.

```
 <target name="test" depends="init">
   <echo>Running JUnit tests</echo>
   <exec executable="${global.was.dir}/bin/launchclient.bat"
         os="Windows NT,Windows 2000"
   >
     <arg line="${global.module.dir}/piggybank.ear"/>
     <arg line="-CCjar=piggybank-test.jar"/>
   </exec>
   <echo>JUnit test complete</echo>
 </target>

 <target name="testgui" depends="init">
   <echo>Running JUnit tests</echo>
   <exec executable="${global.was.dir}/bin/launchclient.bat"
         os="Windows NT,Windows 2000"
   >
     <arg line="${global.module.dir}/piggybank.ear"/>
     <arg line="-CCjar=piggybank-test.jar"/>
     <arg line="-gui"/>
   </exec>
   <echo>JUnit test complete</echo>
 </target>
```

Figure 18-36 Master build file test and testgui targets

Note that we did not make either of the targets that execute the tests depend upon the `start` target. There are two primary reasons for this:

- A developer using the JUnit tests to reproduce a problem may not necessarily want to restart WebSphere every time to re-run the tests.
- The `startServer` command is asynchronous—when the command completes successfully, the start of the application server is not complete. If we attempt to run tests immediately after the `start` target completes the tests may fail since the application may not have completed starting up.

The output from the `test` target is shown in Figure 18-37. As you can see we still have to fix the debit and credit problem with negative amounts.

```
D:\itso4ad\dev\src>ant test
Buildfile: build.xml

init:
     [echo] Build of itso4ad started at 2134 on July 19 2001

test:
     [echo] Running JUnit tests
     [exec] IBM WebSphere Application Server, Release 4.0
     [exec] J2EE Application Client Tool, Version 1.0
     [exec] Copyright IBM Corp., 1997-2001
     [exec]
     [exec] WSCL0012I: Processing command line arguments.
     [exec] WSCL0013I: Initializing the J2EE Application Client Environment.
     [exec] WSCL0035I: Initialization of the J2EE Application Client Environment
                       has completed.
     [exec] WSCL0014I: Invoking the Application Client class itso.was4ad.tests.AllTests
     [exec] ...........F.F......
     [exec] Time: 12.237
     [exec] There were 2 failures:
     [exec] 1) testNegativeDebit(itso.was4ad.ejb.account.tests.AccountTests)
                junit.framework.AssertionFailedError: Shouldn't allow negative debit
     [exec]     at itso.was4ad.ejb.account.tests.AccountTests.testNegativeDebit(
                AccountTests.java:196)
     [exec]     at itso.was4ad.tests.AllTests.main(AllTests.java:29)
     [exec]     at com.ibm.websphere.client.applicationclient.launchClient.
                createContainerAndLaunchApp(launchClient.java:430)
     [exec]     at com.ibm.websphere.client.applicationclient.launchClient.main(
                launchClient.java:288)
     [exec]     at com.ibm.ws.bootstrap.WSLauncher.main(WSLauncher.java:63)
     [exec] 2) testNegativeCredit(itso.was4ad.ejb.account.tests.AccountTests)
                junit.framework.AssertionFailedError: Shouldn't allow negative credit
     [exec]     at itso.was4ad.ejb.account.tests.AccountTests.testNegativeCredit
                (AccountTests.java:180)
     [exec]     at itso.was4ad.tests.AllTests.main(AllTests.java:29)
     [exec]     at com.ibm.websphere.client.applicationclient.launchClient.
                createContainerAndLaunchApp(launchClient.java:430)
     [exec]     at com.ibm.websphere.client.applicationclient.launchClient.main(
                launchClient.java:288)
     [exec]     at com.ibm.ws.bootstrap.WSLauncher.main(WSLauncher.java:63)
     [exec]
     [exec] FAILURES!!!
     [exec] Tests run: 18,  Failures: 2,  Errors: 0
     [exec]
     [echo] JUnit test complete

BUILD SUCCESSFUL

Total time: 23 seconds
```

Figure 18-37 Output from the test target

Conclusions

We hope the discussions presented in this chapter have encouraged you to consider including a unit testing strategy and some sort of unit testing tool in your own development projects.

We remain convinced that the approaches outlined here lead to improved productivity and overall code quality, as well as improving the day-to-day of developers working in your team. Our experience in developing this chapter has merely reinforced this opinion—the simple tests described here uncovered numerous embarrassing bugs that, fortunately for us, you will never see.

Part 5

Appendixes

A

Additional material

This redbook refers to additional material that can be downloaded from the Internet as described below.

Locating the Web material

The Web material associated with this redbook is available in softcopy on the Internet from the IBM Redbooks Web server. Point your Web browser to:

`ftp://www.redbooks.ibm.com/redbooks/SG246134`

Alternatively, you can go to the IBM Redbooks Web site at:

ibm.com/redbooks

Select the **Additional materials** and open the directory that corresponds with the redbook form number, SG246134.

Using the Web material

The additional Web material that accompanies this redbook includes the following:

- readme.txt—short instructions
- sg246134code.zip—all sample code in ZIP format

System requirements for downloading the Web material

The following system configuration is recommended for installing VisualAge for Java and WebSphere Application Server to work with the additional Web material:

Hard disk space:	3 GB
Operating System:	Windows NT, 2000
Processor:	400 MHz or better
Memory:	385 MB, recommended 512 MB

How to use the Web material

Create a subdirectory (folder) on your workstation and download the `sg246134code.zip` file into this folder. Unzip the file onto a hard drive to create this directory structure and files:

```
sg246134
   sampcode                  - master directory
      piggybank              - PiggyBank application EAR
      repository             - VisualAge for Java repository
      studio                 - WebSphere Studio archive file
      ant                    - build and source files for Ant
      log4j                  - source code for Log4J
      junit                  - source code for JUnit
      struts                 - source code for Jakarta Struts
      wsbcc                  - code for WebSphere Business Components
      rose                   - Rose model for the PiggyBank application
      wte                    - html, jsp, webapp files
```

The ***piggybank*** subdirectory contains the code of the PiggyBank application:

```
piggybank.ear      enterprise archive for installation in WebSphere
piggybank.jar      complete source code
table.dll          DDL for customer and account tables
earexpanded        directory with expanded EAR file
warexpanded        directory with expanded WAR file (from EAR file)
```

The **repository** subdirectory contains the exported repository from VisualAge for Java (`piggybank.dat`). This repository contains the code for all versions of the example PiggyBank application.

The **studio** subdirectory contains a Studio archive file (`piggybank-all.wsr`) with HTML and JSP files for the PiggyBank application (basic, Struts, WSBCC). Note that some files may be out of date as compared to the latest Java source code in the Struts and WSBCC directories.

The **ant** subdirectory contains Ant build files and the source code to build the basic PiggyBank application, as described in Chapter 9, “Development using the Java 2 Software Development Kit” on page 183. It also contains generated javadoc documentation and modules for the application.

The **log4j** subdirectory contains a single source file implementing the Log4J version of the PiggyBank log wrapper class discussed in “Using Log4J” on page 354.

The **junit** subdirectory contains Ant build files and the source code to build the JUnit version of the application described in Chapter 18, “Automating unit testing using JUnit” on page 517.

The **struts** subdirectory contains Ant build files and the source code to build the Struts version of the PiggyBank application, as described in “Jakarta Struts” on page 284.

The **wsbcc** subdirectory contains the source code for the PiggyBank WSBCC example discussed in “WebSphere Business Components Composer” on page 303.

The **rose** subdirectory contains the Rose model discussed in Chapter 6, “Modeling and code generation” on page 123.

The **wte** subdirectory contains the Web application HTML and JSP files for the basic, Struts, and WSBCC versions of the PiggyBank for testing in VisualAge for Java. This includes .webapp files and a default.servlet_engine file for configuration of the servlet engine with multiple Web applications. Note that some files may be out of date as compared to the latest Java source code in the Struts and WSBCC samples.

Tip: Some chapters refer to an **ITSO4AD** directory for the sample code. You have to copy relevant portions of the sample code to such a directory to match the description in the chapters.

Installing and running the PiggyBank application

The EAR file supplied in the **piggybank** subdirectory is ready for installation into WebSphere. The EAR contains a deployed EJB JAR for which code has been generated for the PiggyBank CMP EJBs to use DB2 UDB Version 7 as the target database. If you want to use another database, the EJB JAR file must be redeployed (see “EJB deployment tool” on page 418).

Before installing the EAR file into WebSphere you must:

- Create the database tables used by the PiggyBank application—a DDL file is supplied in the piggybank subdirectory
- Create a JDBC driver and DataSource for the PiggyBank CMP EJBs to use—the EAR contains bindings that expect the data source to be located at `jdbc/WAS4AD` in the global JNDI namespace—you may modify this binding when you install the module
- Remove or disable any existing Web application that uses the root context / and shares the same virtual host as the PiggyBank Web application—if you intend to use the `default_host` virtual host you have to remove or disable the WebSphere samples that also use the root context / in order to start the PiggyBank Web application

The EAR file should install into either WebSphere AE or AEs. Once the PiggyBank is installed and started, you can access the application as follows:

- Access the Web application by opening a Web browser on the virtual host name you used to install the application, for example:

  `http://localhost:9080/`

- Start the standalone Swing client by issuing the command:

  ```
  launchclient <path to piggybank.ear>\piggybank.ear
  ```

You can use the Swing client to enter data for the application to use, or enter sample data by hand using SQL commands.

Note: In WebSphere Application Server Version 4.0.1, you may have to update the EAR file to specify a binding of the Web application to the virtual host (for example, default_host). When we developed the PiggyBank using Version 4.0, a default of default_host was taken if none was supplied, but in Version 4.0.1, the Web application fails to start using the EAR file shipped with the sample code.

Importing the sample code into VisualAge for Java

To import the PiggyBank sample code into VisualAge for Java, select the *File -> Import* menu option, then select *Repository* and click *Next*. Select *Local repository* and in the *Repository name* field enter the location of the `piggybank.dat` file extracted from the ZIP file.

Next, click *Projects* and click *Details*.

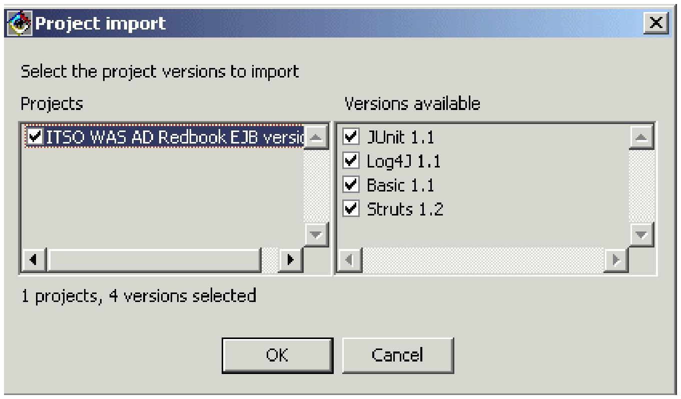

Project import
Select the project versions to import
Projects
Versions available
ITSO WAS AD Redbook EJB versi
JUnit 1.1
Log4J 1.1
Basic 1.1
Struts 1.2
1 projects, 4 versions selected
OK
Cancel

Figure 18-38 VisualAge for Java project import dialog

In the dialog select all four versions of the PiggyBank project (Figure 18-38) and click *OK*. Finally uncheck *Add most recent edition to workspace* and click *Finish* to import the example code into your workspace.

To work with one of the PiggyBank versions, add the appropriate project version to your workspace. Each project version has a comment describing the contents of the project and listing dependencies on third party software not supplied with the additional material.

The WSBCC sample code is also included in the repository—import the package into your repository in the same way, selecting *Packages* instead of *Projects*.

Attention: The PiggyBank example EJBs use J2EE features not supported by VisualAge for Java. As a result, you cannot test the PiggyBank EJB code inside VisualAge without modifying it. This can be achieved without affecting any other components, however. See “Developing EJBs in VisualAge for Java” on page 266 for information on developing EJBs to version 1.1 of the EJB specification in VisualAge for Java.

Using the Ant samples

If you want to work with the sample code that uses Ant, you must first obtain and install Ant, as described in “Using Ant to build a WebSphere application” on page 197.

If you want to build the JUnit, Struts, or Log4J samples, you must also obtain the appropriate third party code, as described in the relevant sections.

You must also install WebSphere Application Server, either Advanced Edition (AE) or Advanced Edition, Single Server (AEs).

Once you have downloaded and installed the software you need, you must edit the `global.properties` file (located in the `src` directory) to specify the locations of the components Ant requires to build the application.

To build the code, open a command window and change to the `src` directory. To rebuild the entire application issue the command:

```
ant clean package
```

To rebuild only modified components, issue the command:

```
ant package
```

If you have WebSphere AEs installed, you can build, install and start the PiggyBank application using the command:

```
ant start
```

For more information refer to the appropriate chapter for the component you are working with.

Note: The Ant examples include database schema and map files for DB2 UDB version 7. If you want to use another database you must create new schema and map files—see “EJB deployment tool” on page 418.

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 567.

- *Programming J2EE APIs with WebSphere Advanced*, SG24-6124
- *Enterprise JavaBeans for z/OS and OS/390 CICS Transaction Server V2.1*, SG24-6284
- *EJB Development with VisualAge for Java for WebSphere Application Server*, SG24-6144
- *Design and Implement Servlets, JSPs, and EJBs for IBM WebSphere Application Server*, SG24-5754
- *Programming with VisualAge for Java Version 3.5*, SG24-5264
- *WebSphere V3.5 Handbook*, SG24-6161
- *Version 3.5 Self Study Guide: VisualAge for Java and WebSphere Studio*, SG24-6136
- *How about Version 3.5? VisualAge for Java and WebSphere Studio Provide Great New Function*, SG24-6131
- *Servlet and JSP Programming with IBM WebSphere Studio and VisualAge for Java*, SG24-5755
- *Revealed! Architecting Web Access to CICS*, SG24-5466
- *IMS Version 7 and Java Application Programming*, SG24-6123
- *Migrating WebLogic Applications to WebSphere Advanced Edition*, SG24-5956
- *WebSphere Personalization Solutions Guide*, SG24-6214
- *User-to-Business Patterns Using WebSphere Advanced and MQSI: Patterns for e-business Series*, SG24-6160
- *WebSphere Scalability: WLM and Clustering Using WebSphere Application Server Advanced Edition*, SG24-6153

- *Patterns for e-business: User-to-Business Patterns for Topology 1 and 2 using WebSphere Advanced Edition*, SG24-5864
- *The XML Files: Using XML for Business-to-Business and Business-to-Consumer Applications*, SG24-6104
- *The XML Files: Using XML and XSL with IBM WebSphere 3.0*, SG24-5479
- *CCF Connectors and Database Connections Using WebSphere Advanced Edition Connecting Enterprise Information Systems to the Web*, SG24-5514
- *WebSphere V3 Performance Tuning Guide*, SG24-5657
- *WebSphere Application Servers: Standard and Advanced Editions,* SG24-5460
- *VisualAge for Java Version 3: Persistence Builder with GUIs, Servlets, and Java Server Pages*, SG24-5426
- *IBM WebSphere and VisualAge for Java Database Integration with DB2, Oracle, and SQL Server*, SG24-5471
- *Developing an e-business Application for the IBM WebSphere Application Server*, SG24-5423
- *The Front of IBM WebSphere Building e-business User Interfaces,* SG24-5488
- *VisualAge for Java Enterprise Version 2: Data Access Beans - Servlets - CICS Connector*, SG24-5265
- *VisualAge for Java Enterprise Version 2 Team Support,* SG24-5245
- *Creating Java Applications Using NetRexx*, SG24-2216

Other resources

These publications are also relevant as further information sources:

- *Enterprise Java Programming with IBM WebSphere*. Kyle Brown, et al. Addison-Wesley Professional, May 2001. ISBN: 0201616173
- *Design Patterns: Elements of Reusable Object-Oriented Software*. Erich Gamma, et al. Addison-Wesley Publishing Company, January 1995. ISBN: 0201633612
- *Patterns In Java*, *Volume 1*. Mark Grand. John Wiley & Sons, September 1998. ISBN: 0471258393
- *The Rational Unified Process, An Introduction*. Philippe Kruchten. 2nd ed. Addison-Wesley Publishing Company, March 2000. ISBN: 0201707101

- *Rules and Patterns for Session Facades*. Kyle Brown. June 2001; article from WebSphere Developer Domain at:

  `http://www7b.boulder.ibm.com/wsdd/library/techarticles/0106_brown/sessionfacades.html`
- *Complement Copy Helper Access Beans with Value Beans in VisualAge for Java*. Willy Farrel. July 2001; article from VisualAge Developer Domain, at:

  `http://www7.software.ibm.com/vad.nsf/Data/Document4083?OpenDocument&p=1&BCT=3&Footer=1`
- *Apache Struts and VisualAge for Java, Part 1: Building Web-based Applications using Apache Struts.* Kyle Brown. May 2001; article from VisualAge Developer Domain at:

  `http://www7.software.ibm.com/vad.nsf/Data/Document2557?OpenDocument&p=1&BCT=3&Footer=1`
- *Apache Struts and VisualAge for Java, Part 2: Using Struts in VisualAge for Java 3.5.2 and 3.5.3.* Kyle Brown. May 2001; article from VisualAge Developer Domain at:

  `http://www7.software.ibm.com/vad.nsf/Data/Document2558?OpenDocument&p=1&BCT=3&Footer=1`
- *Using the Distributed Debugger — Part 2: Stepping Through Code*. Joe Winchester. May 2001; article from VisualAge Developer Domain at:

  `http://www7.software.ibm.com/vad.nsf/Data/Document4383?OpenDocument&p=1&BCT=3&Footer=1`
- *Using the IBM Distributed Debugger — Part 1: Installing and Troubleshooting*. Joe Winchester. March 2001; article from VisualAge Developer Domain at:

  `http://www7.software.ibm.com/vad.nsf/Data/Document4378?OpenDocument&p=1&BCT=3&Footer=1`
- *Automated Builds with VisualAge for Java and Ant*. Glenn McAllister. January 2001; article from VisualAge Developer Domain at:

  `http://www7.software.ibm.com/vad.nsf/Data/Document4366?OpenDocument&p=1&BCT=3&Footer=1`
- *Mapping Rational Rose Models to the VisualAge for Java EJB Development Environment*. Christina Li. December 2000; article from VisualAge Developer Domain at:

  `http://www7.software.ibm.com/vad.nsf/Data/Document2447?OpenDocument&p=1&BCT=3&Footer=1`
- *Development Best Practices for Performance and Scalability*. Harvey Gunther. September 2000; WebSphere white paper at:

  `http://www-4.ibm.com/software/webservers/appserv/ws_bestpractices.pdf`

Referenced Web sites

These Web sites are also relevant as further information sources:

- `http://www.ibm.com/software/webservers/appserv/`
  WebSphere Application Server
- `http://www.ibm.com/software/webservers/studio`
  WebSphere Studio
- `http://www.ibm.com/software/ad/vajava/`
  VisualAge for Java
- `http://www.ibm.com/software/vad/`
  VisualAge Developer Domain
- `http://www7b.boulder.ibm.com/wsdd/`
  WebSphere Developer Domain
- `http://www.ibm.com/software/data/`
  Database and Data Management
- `http://www.ibm.com/framework/patterns`
  IBM Patterns for e-business
- `http://alphaworks.ibm.com`
  IBM alphaWorks
- `http://jakarta.apache.org`
  The Jakarta Project
- `http://xml.apache.org`
  The Apache XML Project
- `http://java.sun.com`
  Sun Java Technology
- `http://www.junit.org`
  JUnit Home Page
- `http://www.w3.org`
  W3C, World Wide Web Consortium
- `http://www.omg.org`
  Object Management Group
- `http://www.rational.com`
  Rational Home Page
- `http://jcp.org`
  Java Community Process
- `http://visibleworkings.com`
  Visible Workings, Java Tracing
- `http://www.software.ibm.com/vadd/`
  EJB deployment tool (EJBDeploy) download
- `http://www.ics.uci.edu/pub/ietf/http/related/iso639.txt`
  ISO-639 language codes

- `http://userpage.chemie.fu-berlin.de/diverse/doc/ISO_3166.html`
  ISO-3166 country codes
- `http://www.software.ibm.com/webservers/appserv/library.html`
  WebSphere JRas implementation

How to get IBM Redbooks

Search for additional Redbooks or Redpieces, view, download, or order hardcopy from the Redbooks Web site:

ibm.com/redbooks

Also download additional materials (code samples or diskette/CD-ROM images) from this Redbooks site.

Redpieces are Redbooks in progress; not all Redbooks become Redpieces and sometimes just a few chapters will be published this way. The intent is to get the information out much quicker than the formal publishing process allows.

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.

Special notices

References in this publication to IBM products, programs or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM product, program, or service is not intended to state or imply that only IBM's product, program, or service may be used. Any functionally equivalent program that does not infringe any of IBM's intellectual property rights may be used instead of the IBM product, program or service.

Information in this book was developed in conjunction with use of the equipment specified, and is limited in application to those specific hardware and software products and levels.

IBM may have patents or pending patent applications covering subject matter 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 the IBM Director of Licensing, IBM Corporation, North Castle Drive, Armonk, NY 10504-1785.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact IBM Corporation, Dept. 600A, Mail Drop 1329, Somers, NY 10589 USA.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The information contained in this document has not been submitted to any formal IBM test and is distributed AS IS. The use of this information or the implementation of any of these techniques is a customer responsibility and depends on the customer's ability to evaluate and integrate them into the customer's operational environment. While each item may have been reviewed by IBM for accuracy in a specific situation, there is no guarantee that the same or similar results will be obtained elsewhere. Customers attempting to adapt these techniques to their own environments do so at their own risk.

Any pointers in this publication to external Web sites are provided for convenience only and do not in any manner serve as an endorsement of these Web sites.

Abbreviations and acronyms

AAT	application assembly tool
ACL	access control list
API	application programming interface
AWT	abtract windowing toolkit
BLOB	binary large object
BMP	bean-managed persistence
CCF	Common Connector Framework
CICS	Customer Information Control System
CMP	container-managed persistence
CORBA	Component Object Request Broker Architecture
DBMS	database management system
DCOM	Distributed Component Object Model
DDL	data definition language
DLL	dynamic link library
DTD	document type description
EAB	Enterprise Access Builder
EAR	enterprise archive
EIS	Enterprise Information System
EJB	Enterprise JavaBeans
EJS	Enterprise Java Server
FTP	File Transfer Protocol
GUI	graphical user interface
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protoco
IBM	International Business Machines Corporation
IDE	integrated development environment
IDL	Interface Definition Language
IIOP	Internet Inter-ORB Protocol
IMS	Information Management System
ITSO	International Technical Support Organization
J2EE	Java 2 Enterprise Edition
J2SE	Java 2 Standard Edition
JAF	Java Activation Framework
JAR	Java archive
JDBC	Java Database Connectivity
JDK	Java Developer's Kit
JFC	Java Foundation Classes
JMS	Java Messaging Service
JNDI	Java Naming and Directory Interface
JSDK	Java Servlet Development Kit
JSP	JavaServer Page
JTA	Java Transaction API
JTS	Java Transaction Service
JVM	Java Virtual Machine
LDAP	Lightweight Directory Access Protocol
MFS	message format services
MVC	model-view-controller
OLT	object level trace
OMG	Object Management Group
OO	object oriented
OTS	object transaction service
PAO	procedural adapter object
RAD	rapid application development

RDBMS	relational database management system
RMI	Remote Method Invocation
SCCI	source control control interface
SCM	software configuration management
SCMS	source code management systems
SDK	Software Development Kit
SPB	Stored Procedure Builder
SQL	structured query language
SSL	secure socket layer
TCP/IP	Transmission Control Protocol/Internet Protocol
UCM	Unified Change Management
UDB	Universal Database
UML	Unified Modeling Language
UOW	unit of work
URL	uniform resource locator
VCE	visual composition editor
VXML	voice extensible markup language
WAR	Web application archive
WAS	WebSphere Application Server
WML	Wireless Markup Language
WSBCC	WebSphere Business Components Composer
WTE	WebSphere Test Environment
WWW	World Wide Web
XMI	XML metadata interchange
XML	eXtensible Markup Language

Index

IBM

Redbooks

WebSphere Version 4 Application Development Handbook

WebSphere Version 4 Application Development Handbook

Complete guide for WebSphere application development

How to make the best use of available tools

Product experts reveal their secrets

This IBM Redbook provides detailed information on how to develop Web applications for IBM WebSphere Application Server Version 4 using a variety of application development tools.

The target audience for this book includes team leaders and developers, who are setting up a new J2EE development project using WebSphere Application Server and related tools. It also includes developers with experience of earlier versions of the WebSphere products, who are looking to migrate to the Version 4 environment.

This book is split into four parts, starting with an introduction, which is followed by parts presenting topics relating to the high-level development activities of analysis and design, code, and unit test. A common theme running through all parts of the book is the use of tooling and automation to improve productivity and streamline the development process.

In Part 1 we introduce the WebSphere programming model, the application development tools, and the example application we use in our discussions.

In Part 2 we cover the analysis and design process, from requirements modeling through object modeling and code generation to the usage of frameworks.

In Part 3 we cover coding and building an application using the Java 2 Software Development Kit, WebSphere Studio Version 4, and VisualAge for Java Version 4. We touch on Software Configuration Management using Rational ClearCase and provide coding guidelines for WebSphere applications. We also cover coding using frameworks, such as Jakarta Struts and WebSphere Business Components.

In Part 4 we cover application testing from simple unit testing through application assembly and deployment to debugging and tracing. We also investigate how unit testing can be automated using JUnit.

In our examples we often refer to the PiggyBank application. This is a very simple J2EE application we created to help illustrate the use of the tools, concepts and principles we describe throughout the book.

SG24-6134-00 ISBN 0738423181