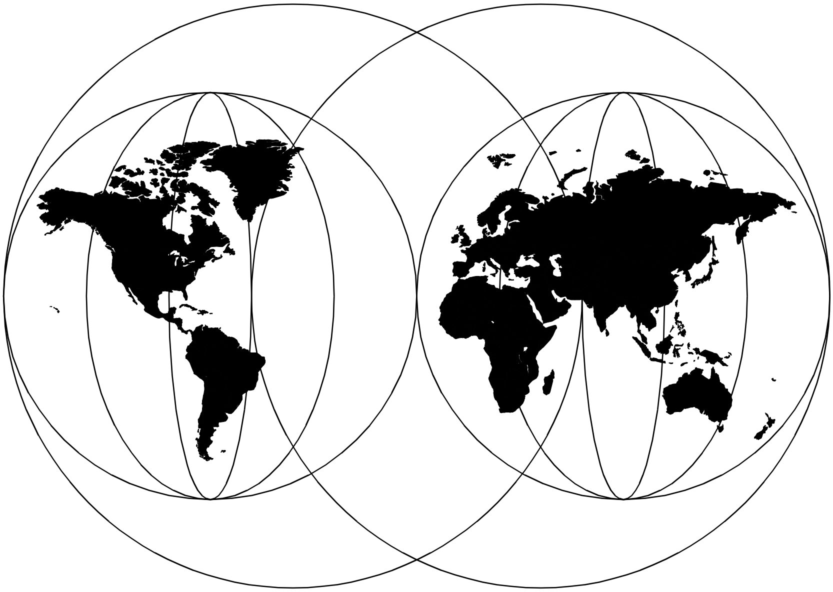
VisualAge for Java Enterprise Version 2 Team Support

Mark Fitzpatrick Uwe Kopf Janice Sullivan
Ueli Wahli



International Technical Support Organization

http://www.redbooks.ibm.com

SG24-5245-00

SG24-5245-00

International Technical Support Organization

VisualAge for Java Enterprise Version 2 Team Support

September 1998

Take Note!

Before using this information and the product it supports, be sure to read the general information in Appendix E, "Special Notices" on page 149.

First Edition (September 1998)

This edition applies to Version 2.0 of VisualAge for Java Enterprise, Program Number 5801-AAR, for use with the OS/2, Windows 95, or Windows NT operating system.

Note

Some captured windows in this book are based on a pre-GA version of a product and may be slightly different when the product becomes generally available.

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

Figures

Tables

Preface

This book is intended to familiarize VisualAge for Java developers with the team programming environment that is part of VisualAge for Java Enterprise Version 2. This book does not cover the basics of Java programming using VisualAge for Java or discuss the other features of the Enterprise product.

The book describes the challenges that a development team faces when working with VisualAge for Java. The book uses short scenarios to illustrate the functions and processes that enable team programming, and the different roles of the team members.

The book covers the complete process of installing and setting up clients and servers, programming in a team environment, and the administrative aspects of maintaining multiple versions of application systems in a shared repository.

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.

Mark Fitzpatrick is a Principal Consultant at the Distributed Systems Technology Centre in Brisbane, Australia. He has worked in the IT business for 20 years. Mark's areas of expertise include object-oriented technologies, distributed architectures such as CORBA, and application development in languages such as Smalltalk and Java.

Uwe Kopf is an IT Consultant at IBM Germany. He has 12 years of experience in application development. Uwe's areas of expertise include VisualAge for Java, VisualAge for Smalltalk, and object-oriented technology.

Janice Sullivan is an IT Specialist at IBM Canada. She has 7 years of experience in application development. Janice has worked at IBM for 13 years. Her areas of expertise include VisualAge for Smalltalk, object-oriented technology, and VisualAge for Java.

Ueli Wahli is a Consultant AD Specialist at the IBM International Technical Support Organization in San Jose, California. Before joining the ITSO 14 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. His e-mail address is *wahli @ us.ibm.com*.

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

Maggie Cutler, Elsa Barron, Emma Jacobs, and Alan Tippett
International Technical Support Organization, San Jose Center

Leigh Davidson and Gary Bist
VisualAge for Java Information Development, IBM Toronto Lab, Toronto

Christophe Elek
VisualAge for Java Support Team, IBM Toronto Lab, Toronto

U. Wahli (Ueli Wahli)

Comments Welcome

Your comments are important to us!

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

❑ Fax the evaluation form found in “ITSO Redbook Evaluation” on page 165 to the fax number shown on the form.

❑ Use the electronic evaluation form found on the Redbooks Web sites:

For Internet users:	`http://www.redbooks.ibm.com`
For IBM Intranet users:	`http://w3.itso.ibm.com`

❑ Send us a note at the following address:

`redbook@us.ibm.com`

1 Introduction

The software development process is becoming more and more complex. End users are demanding that more function be delivered in less time. Many companies are extending their core business applications to enable new users to work in new ways through their intranet and the Internet, and new applications are required to run on many different platforms. All this often results in the need for large development teams to design, build, and maintain applications. Additionally the teams are often forced to maintain or expand existing code in a very short time.

Java programmers need development tools that enable them to work together in a highly dynamic environment. They require facilities that easily allow them to manage multiple versions of their work and switch quickly between the different versions. VisualAge for Java Enterprise provides an extremely flexible, productive, and secure built-in team environment for managing the software life cycle process.

Overview

At its simplest level, the architecture of the VisualAge for Java Enterprise team environment is a two-tier client/server model: multiple developer workstations connected to a single file server.

Residing on the file server is a shared file, which stores all code for all developers of the development team. This file is called the *repository*.

Each developer workstation has a set of executable files that are common to every client as well a unique file that contains a single developer's working code set. This file is called the *workspace*.

The client connection to the server is established over a local area network (LAN), and communication between client workspaces and the repository server is through TCP/IP.

Figure 1 shows the VisualAge for Java Enterprise team development environment.

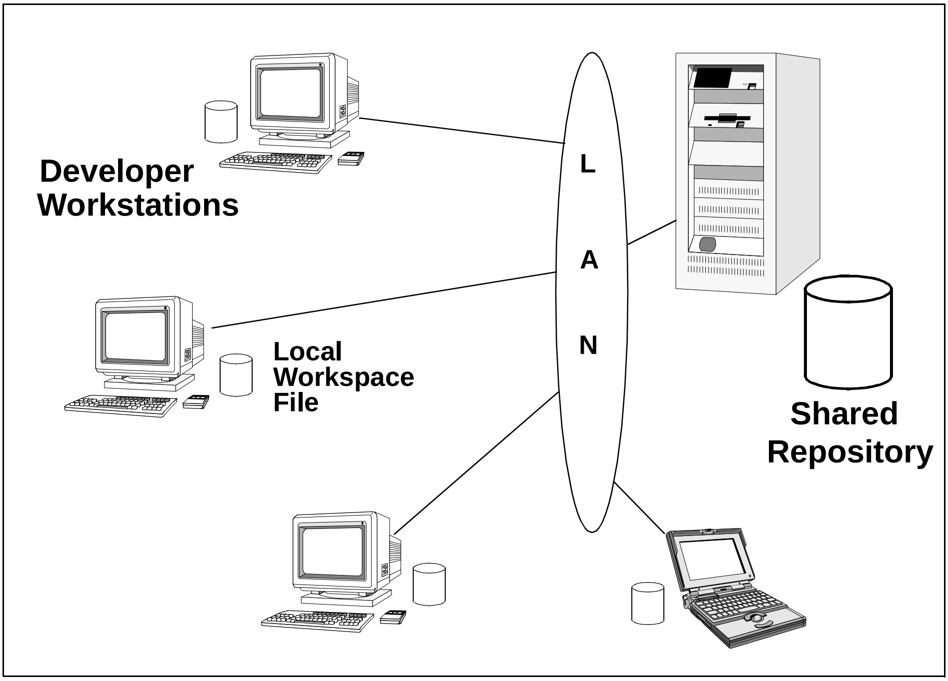


Figure 1. VisualAge for Java Enterprise Team Development Environment

The repository is a large binary file that stores the source and bytecodes of all developer workspaces connected to it. It can be thought of as an object-oriented database that houses all development objects.

The workspace file is unique to each client that is connected to the repository. It contains the bytecodes for the development environment and all program elements that the developer has loaded and is working with. A developer makes changes to code inside the workspace. These changes are always saved immediately into the repository.

Starting VisualAge for Java causes the local workspace file to be loaded into memory and connected to the repository. A VisualAge for Java team client cannot run without a live connection to a repository.

Developers can add program elements, for example, classes or packages, from the repository into their workspace. Only loaded program elements are subject to change by a developer. Generally, many more program elements are stored in the repository than are loaded in a developer's workspace. Developers can also delete program elements from their workspace. Deleted program elements still exist in the repository and can be added back into the workspace.

The workspace file also defines the context of execution when applets and applications are tested during development. All classes and packages that are required to successfully run a program must be loaded into the workspace.

Adding program elements from the repository is a way of easily sharing code among developers working with the same repository. In contrast to other file-based source code management systems, code changes are immediately available to other developers in the group. This does not mean that each developer is directly informed about any changes made by other users. A changed piece of code must be loaded into the workspace in order for it to be accessible. Therefore each developer has full control over which program elements reside in the workspace.

A powerful system of ownership supports the dynamic, concurrent VisualAge for Java team environment. Each program element must have an owner. Thus developers have the freedom and flexibility to make changes and try new things, and at the same time the integrity of each program element is ensured. The ownership model assigns distinct responsibilities to different team members, imposes discipline on the team during the development cycle, and facilitates tracking of changes at maintenance time.

In this book we provide details on the concepts, terminology, and use of the VisualAge for Java Enterprise team environment.

2 Installation and Setup

In this chapter we provide installation details and information about setting up the developer workstations and the server program that controls developer access to the shared repository.

We also cover how to set up client passwords to restrict access to the repository.

Installation of the Team Environment

Follow these steps to install the VisualAge for Java Enterprise team environment:

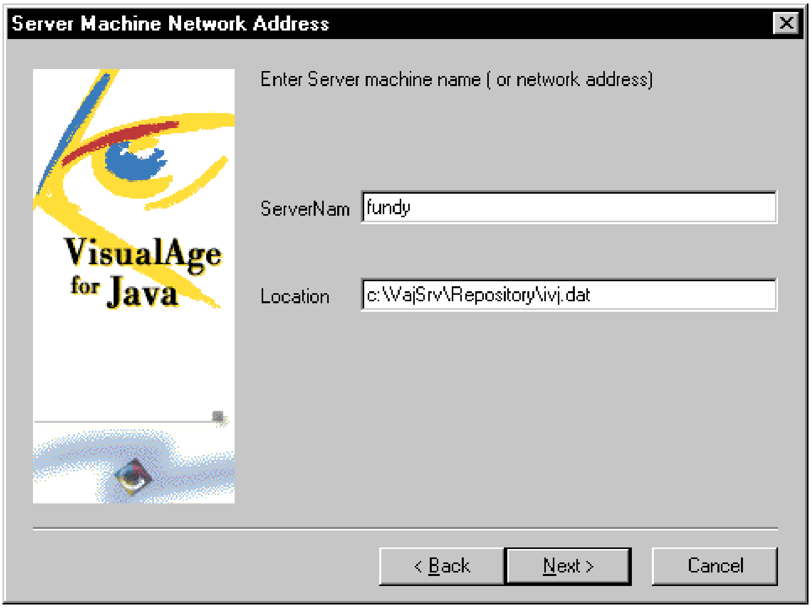
1. Install the server.

   On the server machine, install the server code of VisualAge for Java by following the instructions in *emsrv622.pdf* on the product CD. In this document we assume that a directory named ***C:\VajSrv*** is used for the server code. On NT, the server can be started as a service. See "EMSRV on Windows NT" on page 131 for detailed instructions.

   Create a subdirectory for the repository file, for example, *C:\VajSrv\Repository*, and copy the ivj.dat file from the server CD into that directory.

2. Install a client.

   From a client developer workstation, install VisualAge for Java Enterprise by invoking the *setup.exe* (or install.exe for OS/2) in the installation directory of the product CD. One prompt asks if the repository is on your local machine or on a server. Select server and enter the server host name (or TCP address) and the file name of the repository (Figure 2).



Enter Server machine name (or network address)

ServerNam fundy

Location c:\VajSrv\Repository\ivj.dat

Figure 2. Server Machine Network Address

3. Verify the ide.ini file.

   On the client machine, edit the *ide.ini* file in the *\ide\program* directory to specify the location of the repository. The only section to change in this file is the first one; leave all other sections unchanged. Here is a sample:

   ```
   [JavaDevelopment]
   ServerAddress=fundy      (or 9.1.150.41)
   DefaultName=c:\VajSrv\Repository\ivj.dat
   OpenReadOnly=false
   ```

   Copy the modified ide.ini file to the server as a reference for subsequent client installations.

   For a local repository the ide.ini file would read:

   ```
   [JavaDevelopment]
   ServerAddress=
   DefaultName=../repository/ivj.dat
   ```

4. Start the ENVY/Manager Server (EMSRV) process on the server.

   [A bit of history: The term *ENVY/Manager* comes from the VisualAge for Smalltalk environment where it is used to refer collectively to the shared repository, the EMSRV program, and the client dynamic link libraries (DLLs) and primitives. In this book we use the phrase *VisualAge for Java Enterprise server* (or just *team server)* to replace the term *ENVY/Manager*. However, we retain the term *EMSRV* because it is the actual name of the program that runs on the server.]

   As an example, you can start EMSRV in the VajSrv directory as:

   ```
   emsrv -u <username> -p <password>
   ```

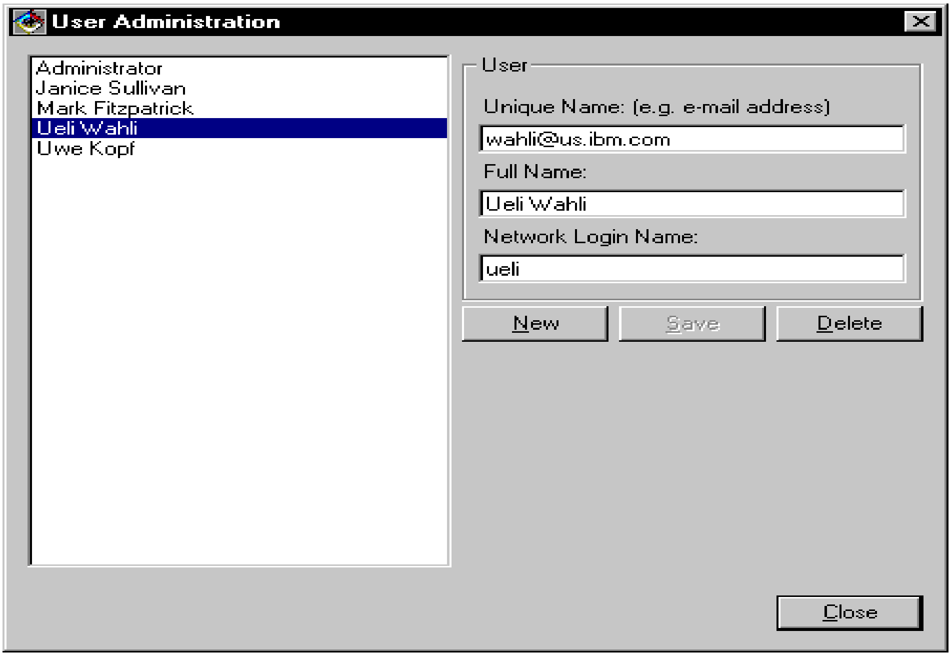
   Read "Accessing the VisualAge for Java Enterprise Server" on page 9 and refer to Appendix B, "EMSRV" on page 127 before completing this step so that you will understand the details of running EMSRV.

5. Start the VisualAge for Java client.

   You can start the VisualAge for Java client from the IBM VisualAge for Java icon in the program folder created at installation. An informational message tells you that the workspace is connecting to a new repository. Answer OK to proceed with this step. Next a dialog asks you to select a user for this workspace. There is initially only one choice; select *Administrator* and proceed to enter the network login name for this user. (See "Users" on page 17 for the definition of the network login name.)

6. Create new users.

   The administrator creates new users from the *Admin* menu option of the Repository Explorer. Select *Users...* to open the dialog to create, modify, or delete users (see Figure 3).



Administrator
Janice Sullivan
Mark Fitzpatrick
Ueli Wahli
Uwe Kopf

Unique Name: (e.g. e-mail address): wahli@us.ibm.com

Full Name: Ueli Wahli

Network Login Name: ueli

Figure 3. User Administration Dialog

You should add a user definition for each developer that will be installing VisualAge for Java. Refer to "Users" on page 17 for details on the attributes of user objects.

If this workspace will belong to another developer, you should *Change Workspace Owner...* before exiting and saving VisualAge for Java. The user will be saved along with the workspace.

7. Install subsequent clients.

   Install subsequent clients as in step 1. The developer can edit the *ide.ini* file as in step 3 or copy the file from the server where it was saved. The local repository, *d:\IBMVJava\ide\repository\ivj.dat*, on all clients can be deleted if not needed locally. When the client is started, the developer selects the correct user object for the workspace.

Accessing the VisualAge for Java Enterprise Server

In VisualAge for Java Enterprise, developers access the common repository by communicating through EMSRV, a server process that manages multiple concurrent client access to a single file—the repository.

The EMSRV process, when running on a suitable network machine, provides access to the repository file for multiple VisualAge for Java clients.

VisualAge for Java Enterprise provides an EMSRV program for the following environments:

❑ OS/2 Warp Version 4.0
❑ AIX Version 4.2.1 (including smp)
❑ Windows NT Workstation or Server Version 4.0
❑ Novell NetWare Version 3.12, 4.1, and 4.11
❑ Novell IntranetWare Version 4.11
❑ Sun Solaris Version 2.6
❑ HP-UX Version 10.20

Typically, EMSRV is installed and maintained by your network or system administrator with assistance from the VisualAge for Java Enterprise administrator. Appendix B, “EMSRV” on page 127 and Appendix C, “EMADMIN” on page 133 provide details of the programs and parameters to use when you install and maintain EMSRV.

Client developer workstations communicate with EMSRV through TCP/IP. Figure 4 shows the topology of a VisualAge for Java Enterprise team environment.

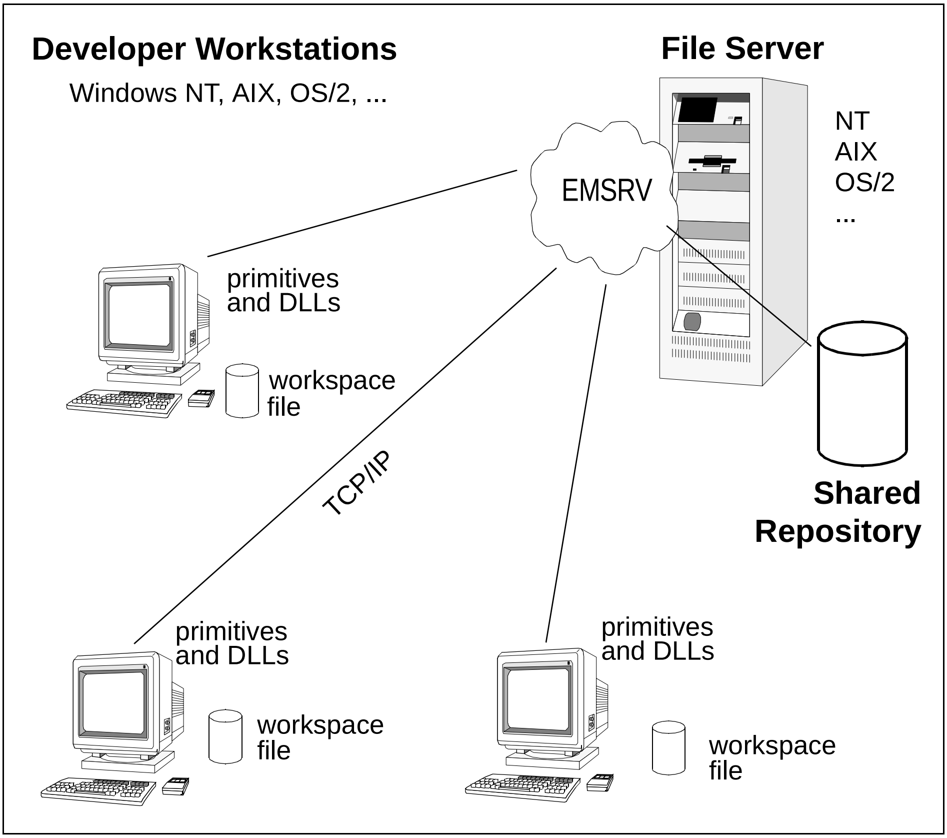


Figure 4. VisualAge for Java Enterprise Team Architecture

It is possible for a VisualAge for Java developer to connect directly to a repository by using native file I/O. However, because the file is then locked, it is not possible for any other client to access the repository. Therefore the use of EMSRV and its multiuser connection services is mandatory in the VisualAge for Java Enterprise team environment.

Attention!	Do not develop on the server machine
	It is possible to install and run a VisualAge client on the server with EMSRV and the repository, but we do not recommend this configuration. Applications under development can affect the operation of a workstation. If development is performed on the server and the machine crashes, the shared repository could become corrupted and all team members would be affected.

Password Protection for Clients

To support multiple concurrent developers, each repository contains a collection of user objects. Each workspace that connects to the repository must have a valid user object associated with it. It is possible for a developer, while VisualAge for Java is running, to change the user of the workspace.

Changing the user of the workspace alters the capabilities and permissions of the developer inside the workspace and the repository. You can imagine that this may not always be desirable. With VisualAge for Java Enterprise, a user ID and password can be assigned to each developer defined in the repository. Password protection is controlled through EMSRV and provides the following three configurations:

- ❑ **No password protection**

  This is the default behavior of VisualAge for Java Enterprise. Developers are allowed to switch to other users at will. When VisualAge for Java is closed and saved, the current user is saved with the workspace and will be the user associated with the workspace the next time VisualAge for Java is started.

- ❑ **Use VisualAge for Java user IDs and passwords for all developers**

  This option requires the maintenance of a *passwd.dat* file in the EMSRV (VajSrv) directory on the server. To enable user verification with a *passwd.dat* file, EMSRV must be started with the -rp parameter. You should ensure that users do not have access to the file.

  The *passwd.dat* file is a plain text file. Each user needs one line, consisting of a user name and the associated password. If a VisualAge for Java user is not in this file, that user will not be authorized to use EMSRV.

  The user name in the *passwd.dat* file must match the network login name of the VisualAge for Java user object definition. (Refer to “Users” on page 17 for a complete description of user objects and their attributes.) The user name is followed by one space and then the password. Passwords may not include spaces, and we recommend that they not be the same as the user’s actual system logon password.

  When starting up VisualAge for Java, the developer is prompted for the password of the current user. After a third try with an invalid password, VisualAge exits.

  Once logged on with the workspace started, the developer can still choose the *Change Workspace Owner* option but will have to know the password of the new user to successfully change the owner of the workspace.

Here is a sample *passwd.dat* file:

```
mark aussie
uwe schwabe
jan canada
ueli swiss
```

Important Info	Only one passwd.dat file per EMSRV process
☞	The passwd.dat file is unique to an EMSRV process and not to a repository. If a single EMSRV is managing access to multiple repository files, users defined in all of the repositories must be included in the one passwd.dat file.

□ **Use native operating system user IDs and passwords**

This option does not require a *passwd.dat* file. When starting EMSRV, specify the -rn parameter on Windows NT and the -v parameter on AIX. This option is not available on OS/2. When starting VisualAge for Java Enterprise or using the *Change Workspace Owner* option, a developer is prompted for the password of the current user.

When the administrator defines new users to VisualAge for Java Enterprise, the network login name of the user object must match the operating system logon ID for this option of user authentication.

3 Understanding the Basics

In this chapter we describe the concepts and terminology used in the team environment of VisualAge for Java Enterprise. We explain the four basic management services that VisualAge for Java Enterprise offers: managing system and program elements, version management, change management, and configuration management.

Because VisualAge for Java Enterprise is based on an ownership model, team members assume specific roles at different times. We give details of these roles here and explain how they provide the basis for change control in the VisualAge for Java Enterprise team environment.

Program and System Elements

VisualAge for Java Enterprise distinguishes between *program elements*, which are managed by the developers, and *system elements*, which are managed by the administrator of the repository (Figure 5).

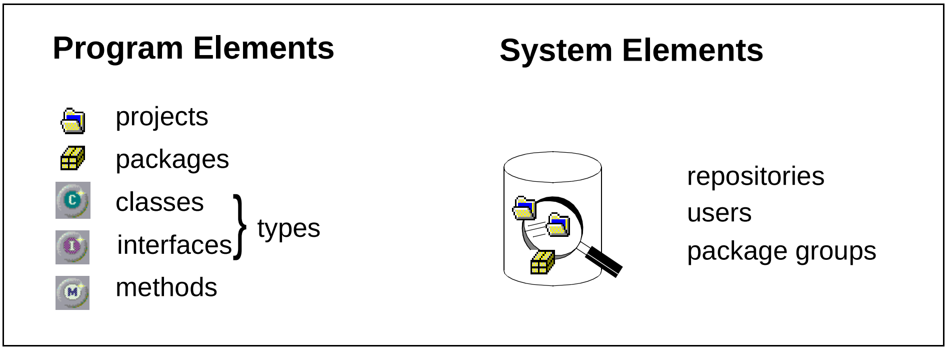


Figure 5. Program and System Elements

Program Elements

Program elements are the building blocks of an application and are managed by the developers. VisualAge for Java Enterprise handles projects, packages, classes (and interfaces), and methods.

Projects

The project is the unit of organization within VisualAge for Java. Projects are used to group related packages in a way that is meaningful to the users of VisualAge. A project has no analogy in the Java language.

Packages

A VisualAge for Java package has exactly the same meaning in VisualAge as it has in the standard Java language.

Classes

Classes and interfaces are the building blocks in VisualAge just as they are in the Java language. The VisualAge for Java development environment sometimes refers to classes and interfaces with the generic term *type*. Although we might occasionally adopt the same convention in this book, we generally prefer to use the term *class*. Whenever we refer to a class we implicitly refer to an interface as well. For example, if we talk about versioning a class, it is understood that the same operation can be applied to an interface. If this rule does not apply, we explicitly say so in the text.

Methods

A method in VisualAge for Java is exactly the same as a method in the standard Java language.

For a complete description of program elements, see Marc Carrel-Billiard and John Akerley, *Programming with VisualAge for Java,* Prentice Hall, 1998.

Relationships between Program Elements

When dealing with program elements in VisualAge, it is often helpful to consider the relationship between the various elements:

- ❑ Inside the workspace there is a *contained in* relationship between program elements. Any given element is always contained in another program element at a higher level. A method is *contained in* a class, a class is *contained in* a package, and a package is contained in a project (see Figure 6).

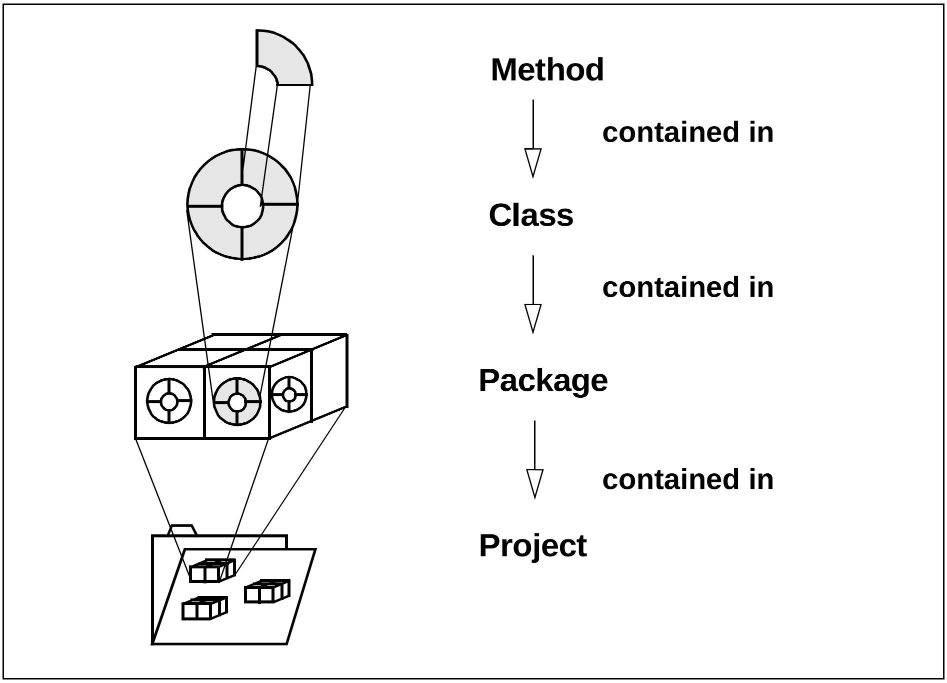


Figure 6. Contained in Relationships between Program Elements

❑ When packages and projects reside in the repository, a *part of* relationship can exist between them. A part of relationship is less restrictive than a contained in relationship. For example, a package can be part of several projects. However, it is not possible to have more than one project with the same package being loaded in the workspace at a time (see Figure 7).

A package can reside in a repository without being part of a project. For example, this is the case if you import a package that is not assigned to a project. When you load a package into the workspace, it must be assigned to a project.

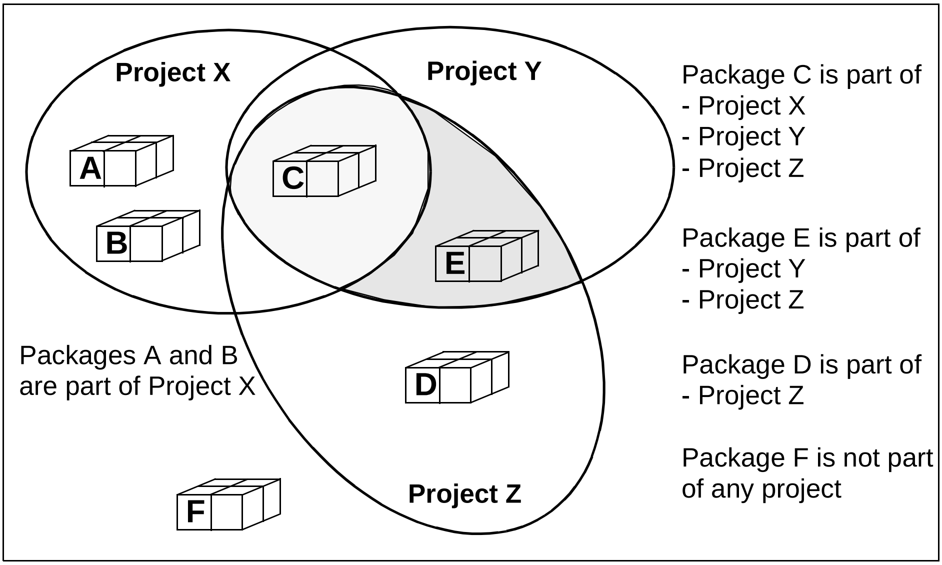


Figure 7. Part of Relationship between Packages and Projects in the Repository

System Elements

System elements are used to support the team environment itself. They are maintained by the administrator of the repository.

Repositories

A repository is a binary file where all parts of a software system are stored. It can be accessed by multiple users concurrently through a LAN. These users can reside on multiple platforms supported by VisualAge for Java Enterprise.

Each repository is managed by a special user object called the *Administrator*. This user preexists in every VisualAge for Java repository and cannot be deleted. (See “The Role of the Administrator” on page 76.)

Users

VisualAge for Java Enterprise maintains a list of authorized users for each repository. Only the administrator is allowed to create, change, and delete users from the repository.

A user is described by three attributes:

❑ **Unique name**

The unique name is the user’s identifier. It must have a unique value. We suggest using the e-mail address to make sure it is a unique value in your team or company. Companywide unique names avoid problems when moving packages between different organizations, because user objects are also imported when you import a package from another repository. The unique name is case sensitive.

❑ **Full name**

The full name is used by the system to refer to a particular user. Any list of users as well as the Properties window of a program element show the full name of a user. To avoid confusion, we recommend using unique values for the full names.

❑ **Network login name**

The network login name is the user ID of the user and is used to identify the user to the repository. If you choose to enable password verification at logon, the network login name is the user ID against which the password is verified. For more information about passwords, refer to “Password Protection for Clients” on page 11. A network login name must be provided when you create a user. However, if you choose not to use password verification at logon, the network login name is not used.

Package Groups

Each edition of a package has a package group associated with it. A package group contains a list of users who are allowed to create classes in this package. The user who created the package is by default the *package group owner*. The owner is the only one who is allowed to add or delete other users to or from the package group (see “Change Management” on page 25). The owner can also pass the ownership to another user.

A package that is undergoing split stream development can have different package groups assigned to each development stream. Each development stream is represented by a different edition of the package.

Note: The package owner is indicated in the middle lower pane of the Managing page of the workbench with a leading >. Be aware that this is the same symbol used for indicating unreleased editions!

Version Management

Program elements can exist in different states. Once a program element has reached a certain state that you want to save as a baseline, you can mark the state as immutable. To apply any changes to this piece of code, you first have to create a copy of it. This mechanism is commonly known as *version control*.

VisualAge for Java Enterprise provides three constructs to manage changes to program elements (see Figure 8):

❑ Open edition

❑ Version

❑ Scratch edition

All three constructs can be treated as editions. In other words, you can think of a version as an immutable edition.

We refer to a program element that is currently loaded in the workspace as the *current edition*. A current edition can be either an open edition or a version. It can never be a scratch edition, because a scratch edition does not exist in the repository and therefore cannot be loaded from the repository. Open editions, versions, and scratch editions represent the *status of a program element*.

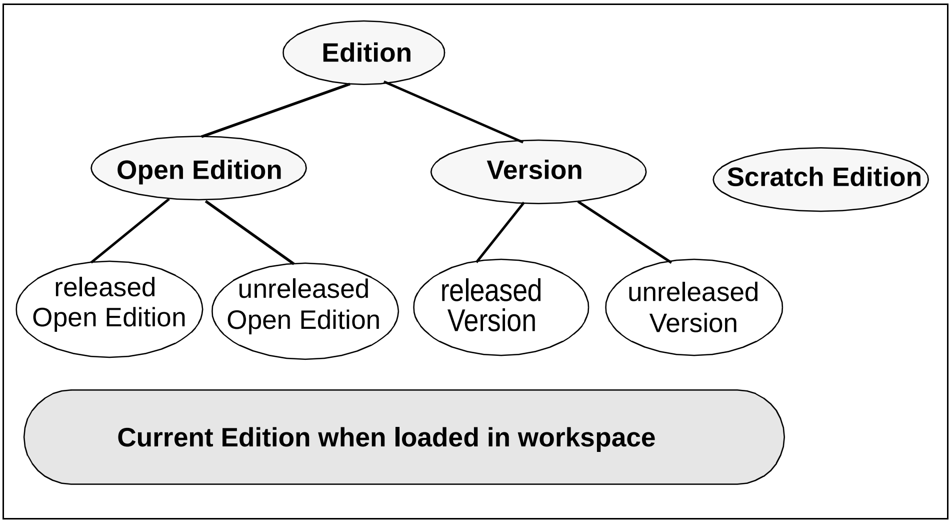
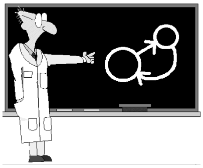


Figure 8. Terms for Editions of Program Elements

Open Edition

An open edition is a program element that you can change. It is indicated by a time stamp containing the date and time of its creation. You cannot change this unique identifier. All open editions of program elements have a time stamp.

Tip	Time stamps
	Time stamps of methods are not shown in the various standard browsers, but they can be seen in method properties. Time stamps of open editions are referred to as the *Version Name* in the Properties window.

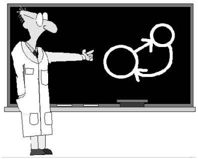
You can have more than one open edition of a program element in the repository, but only one can be loaded in the workspace at any given time. Thus developers have the flexibility to work down several paths on the same program element.

You can create an open edition either manually or automatically. Remember that a version is an immutable program element. You can explicitly create an open edition from a version. An open edition is automatically created by VisualAge for Java Enterprise if you make changes to an existing open edition of a method. When you save the changes, a new open edition of the method is automatically created. Creating an open edition from an existing version actually makes an editable copy of the program element. The version, which is not editable, remains in the repository.

Version

A version is an edition that you cannot change. It is indicated by a version name.

VisualAge for Java provides automatic naming of versions, starting with 1.0 as the first version name of a new element and then adding one to the last number (1.1, 1.2). You do not have to accept this default name. Instead you can use your own, more specific name, that also can include letters. We suggest using specific prefixes in the version name so it is easy to determine which version was created by which developer and at what time.

Tip	Version name
	Version names should be used carefully. We recommend using `<initial>-<date>-<version number>` for a version name; for example, `JS-6Mar-2.1`.

A version is always created from an open edition. To version a project or package, all program elements at a lower level have to be versions. In the case of versioning packages, versioning of the contained classes may result in interaction with the different class owners, because only the class owner is allowed to release a class or interface. As a project owner you are allowed to version the containing packages individually or by versioning the project itself. Note that older versions still exist in the repository (see Figure 9).

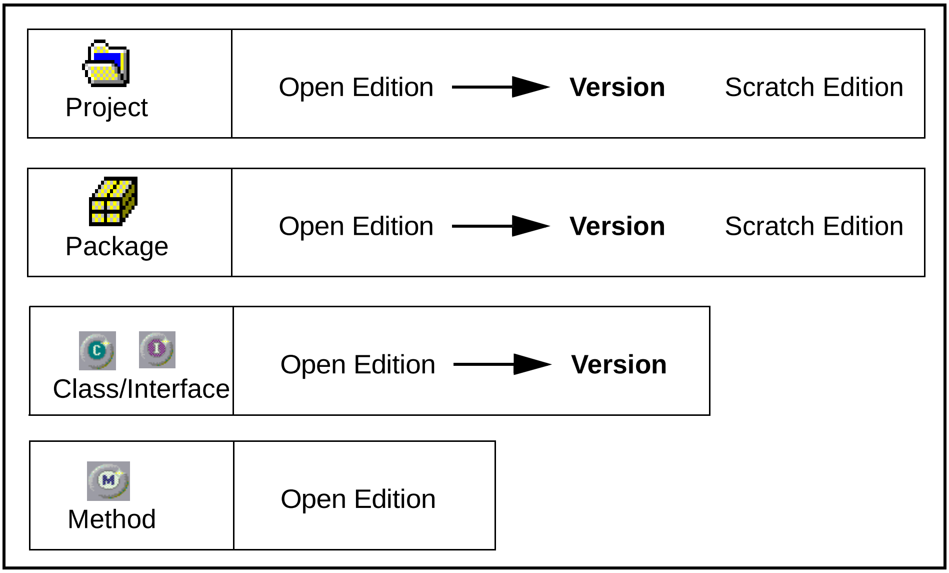


Figure 9. Versioning Program Elements

You can have more than one version of a program element in the repository, but only one is loaded in the workspace at a time.

When you deploy an application, we strongly recommend that all program elements be versions.

Scratch Edition

A scratch edition is a private edition and not visible to other users. It resides in the user's workspace, not in the repository.

There are two situations where the construct of a scratch edition is helpful:

❑ You want to do some testing with a versioned package that you own.

You have to make changes to a contained class that are not intended to be visible to other users. When you save the changes made in the class, a new open edition of the class is created as well as a scratch edition of the package. Because a scratch edition is a private edition, it is not seen in the repository. Therefore it will not fill the repository with unnecessary editions of the package that do not refer to any kind of baseline code. After testing you can reload the original version of the package.

❑ You have to make changes to a class within a package, but that package is owned by another user.

Usually, the owner has to create a new open edition of the package before you can start making changes (which also means creating a new open edition of the class). If the owner of the package is not available to create a new open edition, you would not be able to continue your work. However, the concept of scratch editions enables you to work with your own private edition of the package (which is created automatically by VisualAge for Java Enterprise).

You can have only one scratch edition of a package or a project in your workspace at a time. There are no scratch editions of a class. If you make changes to a version of a class, you always get an open edition. A scratch edition is indicated by displaying **<>** around the name of the original version.

History of Program Elements

VisualAge for Java Enterprise keeps track of all states of all program elements in the repository. They can be viewed by the Repository Explorer. A complete history of all program elements is available.

VisualAge for Java Enterprise also provides a management query facility to search for different program elements in different states that are currently loaded in the workspace. You can add owner and/or class developer information as additional arguments for the query. For more information about the query facility, see “Management Query” on page 67.

Configuration Management

Many different program elements are created during development, and most of them will exist in many different editions and versions in the repository. There are relationships among the program elements. For example, a certain version of a package contains several classes. Each class itself may exist in multiple editions or versions. A configuration management tool has to keep track of which version or open edition of each program element is contained in a particular version or open edition of the containing program element. The summary of these references is called a configuration. For example, you can think of a version of a package as a named, versioned configuration of class versions. VisualAge for Java Enterprise allows multiple configurations of the same element.

Different configurations are maintained:

- ❑ Each version or open edition of a class represents a configuration of method editions.
- ❑ Each version or open edition of a package represents a configuration of class versions or open editions.
- ❑ Each version or open edition of a project represents a configuration of package versions or open editions.

Releasing Program Elements

A configuration represents the relationship between an open edition or version of a program element and the open editions or versions of the program elements that it contains when it is loaded in the workbench. To add a specific version or open edition of a program element to a configuration is referred to as *releasing* (see Figure 10).

In other words, releasing determines which edition of a program element is loaded when a specific configuration is added to the workspace by a developer.

Different rules exist for releasing different program elements. The terms of ownership used in the discussion that follows are described in detail in "Ownership" on page 25.

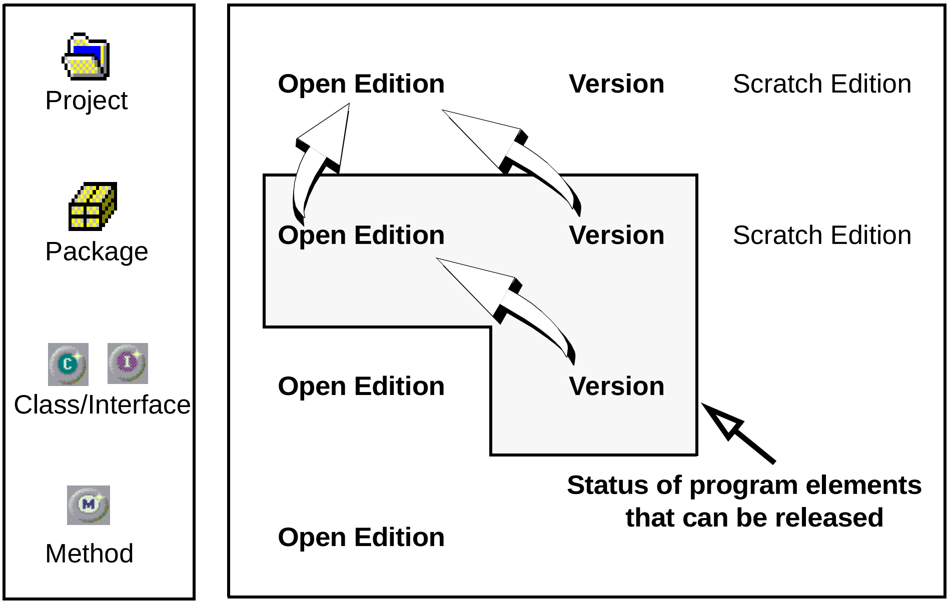


Figure 10. Releasing Program Elements

Releasing Methods

Each time a method is created or changed, a new edition is created and automatically released to the edition of the class. There is no explicit release mechanism for methods.

Releasing Classes

Only versions of a class or interface can be released. Classes are blocks of executable code in Java. When you work on a class or interface, it is an open edition, and it may not be in a state where it should be used by other developers. Once a class or interface is released to a package, it is used by all developers when they load the current configuration of the package into their workbench. To ensure that only classes with well-defined and tested behavior are loaded, only versions of a class can be released.

Important Info	Who can release a class?
☞	Only the class owner can release a class version into the containing package, whereas only the class developer can version the class.

Releasing Packages

A package can be released as an open edition or a version:

❑ Releasing an open edition of a package

When you release an open edition of a package, it is not necessary to have all contained classes versioned and released. However, this does not mean that open editions of contained classes are going to be released. Instead, the most recent released version of the classes will be part of the resulting configuration of the package. For example, say you have an interface, MyInterfaceA, as an open edition currently loaded in the workbench and a released Version 5.0 in the repository. When the open edition of the package is released into the project, the configuration of the project contains not the current (open) edition of the interface but the released version in the repository.

❑ Releasing a version of a package

Before you can version a package, all contained classes and interfaces have to be versioned and released. Then you can version the package and release it into a project.

When you create a new package in a project, the initial open edition of the new package is automatically released to the project.

Important Info	Who can release a package?
☞	The package owner and the owner of the containing project are allowed to release an open edition or a version of a package into a project.

Change Management

Change Management is implemented by VisualAge for Java Enterprise through the concept of ownership of program elements.

Ownership

Traditional change management tools are often based on a concept of reserving elements. A user reserves (or checks out) a program element to prevent other users from modifying it concurrently. In most of these systems any user is allowed to reserve elements.

Element ownership as implemented in VisualAge for Java Enterprise is the basis for an alternative strategy that enables developers in a team to work dynamically and concurrently but always have the appropriate control over the development process. For example, multiple developers may change a class more or less simultaneously and even create versions of it. However, there is only one individual, the class owner, who is responsible for the main stream of the development for that class. The class owner is the only one who can release the class version into the containing package. Therefore, although more than one developer is allowed to change a class, only the class owner determines which of the existing versions is visible to the other team members when they load the containing package.

Owners are typically members who coordinate the work of many developers. Depending on the size and the complexity of the product, your team might have one project owner, and a package owner for each package. Each program element managed by VisualAge for Java Enterprise has an owner. The owner is responsible for integrity and maintenance during the whole life cycle of the program element. Certain operations are performed only by an owner.

Assigning Ownership

Ownership can be assigned to a user explicitly or implicitly. A user gets implicit ownership of a program element when it is created. A user can get explicit ownership of projects, packages, and classes by having the ownership transferred from the current owner. The current owner relinquishes ownership on that program element. Therefore, although ownership is not immutable during the development process, only one person at a time is responsible for the class functionality.

Methods are implicitly owned by the owner of the containing class.

Scope of Ownership

If you are the owner of a class, you are the owner of all existing editions of that class within the package edition (see also “Class Owner” on page 27). If you are the owner of a package or a project, you are the owner of only the current edition of the package or project. Thus different editions of packages and projects with different owners exist in the repository.

This different behavior is due to the fact that VisualAge for Java Enterprise treats classes as elements that have executable behavior. Therefore editions are single steps to achieve the desired functionality of a class, and only one user, the class owner, is responsible. In contrast, packages and projects are constructs used for organizational and administrative purposes. Different users can be responsible for different editions.

Team Roles

Change control is based on a number of different roles for the developers in a team:

❑ Class developer

❑ Class owner

❑ Package owner

❑ Project owner

Still another role is implemented in VisualAge for Java Enterprise—the role of administrator. We will discuss this special role in “The Role of the Administrator” on page 76, because the administrator’s tasks are not tightly related to the developer’s tasks within a typical development cycle. The administrator manages the repository as a whole independently from certain development projects.

Class Developer

A class developer is a user who has created an edition of a class or interface. Any user registered by the administrator to work with the repository can make changes to any class or interface. A class developer does not have to be the class owner. A class developer does not have to be in the package group to which the class belongs. Creating an open edition of a class automatically records the current user as the class developer. This behavior provides the flexibility that is necessary in a highly concurrent development environment.

How do I become a class developer?

- ❑ By creating an open edition of an existing class
- ❑ By creating a new class in a package in which you are a package group member. In this case you also become the owner of the class.

What can I do as a class developer?

- ❑ Add the open edition to your workbench
- ❑ Version the open edition
- ❑ Change the class definition
- ❑ Change, save, delete, or reload a method

Class Owner

A class owner is the team member responsible for the integrity of a class or interface in a package edition. Class ownership promotes the creation and maintenance of quality code for the following reasons:

- ❑ Classes are not equivalent to modules in the traditional structured programming sense. Classes are building blocks for present and future systems. To improve code quality and reuse, class ownership helps to ensure the reliability and generality of a class throughout the life cycle.
- ❑ The semantic dependencies of an object-oriented model (such as inheritance) increase the risk that a poorly considered change could have a unfortunate ripple effect. Only a class owner can change a class or interface and be reasonably expected to understand the ramifications of that change.
- ❑ The semantics of check-in/check-out are unclear when inheritance is used. If an abstract superclass is checked out, for example, should all of its concrete subclasses be reserved as well?
- ❑ Long-term class ownership introduces cultural changes in the software team. When developers know they will be maintaining a class for a significant period of time, they will be less likely to program short-term fixes.

Class ownership provides the controls necessary for serious software development while offering the flexibility to let developers experiment quickly. The key is that only the class owner can release a class into its containing package. Other developers can create an open edition of a class, change the open edition, and even version the open edition. The class owner, however, determines the main stream of development of the class, through releasing the class.

How do I become a class owner?

- By creating a new class or interface in a package edition in which you are a package group member (you also become the developer of the class).
- By explicit change of ownership: any member of the package group (including you) can set you as the owner of a class in the package.

Important Info	Who can change class ownership?
☞	The fact that any member of a package group can reassign ownership of a class is the sole exception to the concept that only the owner of an existing program element can change the ownership.

What can I do as a class owner?

- Release a version of the class
- Delete a class

The scope of class ownership is restricted to a single package edition. Therefore, if a package has multiple streams of development, the class might have a different owner in each stream.

Package Owner

A package owner is responsible for the overall status of a package. The package owner also manages the package group and therefore coordinates the class developers working on that package.

How do I become a package owner?

- By creating a new package in a project you own
- By getting the ownership of an existing package transferred by the current owner of the package. You must be a member of the package group.

What can I do as a package owner?

- Add new users to the package group of the package
- Delete users from the package group, as long as they do not own any class
- Transfer ownership to another member of the package group
- Create an open edition of the package
- Create a version of the package
- Release an open edition or a version of the package

Project Owner

The project owner is responsible for the organization of the project. The project owner is the only one who can create packages in a project and maintain the integrity of the whole project.

How do I become a project owner?

❑ By creating a new project

❑ By getting the ownership of an existing project transferred by the current owner of the project

What can I do as a project owner?

❑ Create an open edition of the project

❑ Create a version of the project

❑ Create packages in the project

❑ Delete packages from the project, which means change the configuration of the project

❑ Transfer ownership of the project to any other user defined in the repository

These team roles are dynamic in VisualAge for Java Enterprise. Your role may change many times during development without you being aware of it. For example, if you are a package owner and you create a class in that package, you act as a package owner. When editing and versioning the class, you act as a class developer. When releasing the versioned class to the package, you are doing that as the class owner. You do not notice all of these role changes except when you are about to perform a task that requires privileges from another team role that is not assigned to you.

From the descriptions of the roles, we can deduce that there are two roles that any developer can automatically assume without regard for any controls on the repository. They are class developer and project owner. Therefore you are always allowed to start your own development project and you are always allowed to work with all released program elements in the repository. However, you are not allowed to make any changes to a package and its containing classes if you are not a member of the package group.

Attention!	Can any user delete any project?
	We know that deleting from a workspace is the equivalent of an *unload* operation and does not delete from the repository. At this point you may be thinking that deleting projects and deleting packages are similar operations. They are not. Deleting a project from your workspace does not change any configuration. However, unloading a package from your workspace changes the configuration of the project of which that package is a part. Therefore any developer can delete any project from his or her workspace, but only the owner of a package or the owner of the containing project can delete a package.

4 The Team Development Process

In this chapter we describe the team development process in more detail. To get the most out of the team programming features of VisualAge for Java Enterprise, it is important that you adhere to some well-established modes of operation, which we call *development patterns*. We examine several such patterns, ranging from the simple to the complex. We also discuss what happens when development has finished and your work has to be delivered to the end user.

Activity Patterns for Daily Development

In this section we discuss how you can use VisualAge for Java Enterprise for daily development activities. We first introduce the basic development pattern, which describes how programmers work with class editions and versions. We then explore four different development scenarios ranging from a simple, single developer environment to large projects involving multiple developers working on multiple packages.

The Basic Development Pattern

Figure 11 shows the most fundamental development pattern in VisualAge for Java. As a class developer, you move through a cycle of creating a new edition, making changes to it, and versioning it.

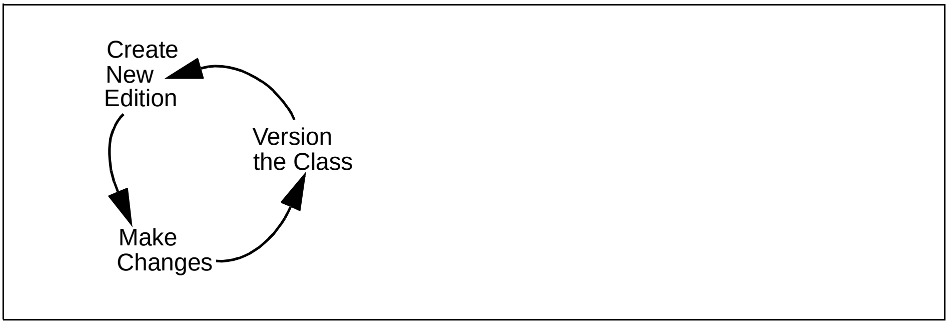


Figure 11. The Basic Development Pattern for Classes

You can enter this cycle at different points depending on the overall context in which you are working. You leave this cycle having created a new version of the class. This basic pattern forms the core of all the work you do in VisualAge for Java and is repeated in all of the scenarios we explore.

The frequency with which you version your class depends on personal style and the overall team environment. A rule of thumb is to version the class or interface whenever you reach a known state to which you may want to return. Versioning classes at least on a daily basis would be typical practice. This pattern naturally supports an incremental development style that is usual in object-oriented programming.

When you reach a point in the cycle where you want to preserve your code, version it and then move on to the next phase with the confidence that if anything goes wrong you can always return to a known state. If you are working as part of a team, there may be a requirement to version your classes at defined intervals, so that your classes can be seen by other developers or be released by the class owner for the creation of a package baseline.

Single Package, Single Developer

This scenario, a small application consisting of a single package where only one person is responsible for all the work, is the simplest way of working with VisualAge for Java Enterprise. The application may use classes from other packages but does not have to change them. Figure 12 shows the overall process. Although this is a simple scenario, it is probably representative of the many small applications that will be developed in VisualAge for Java.

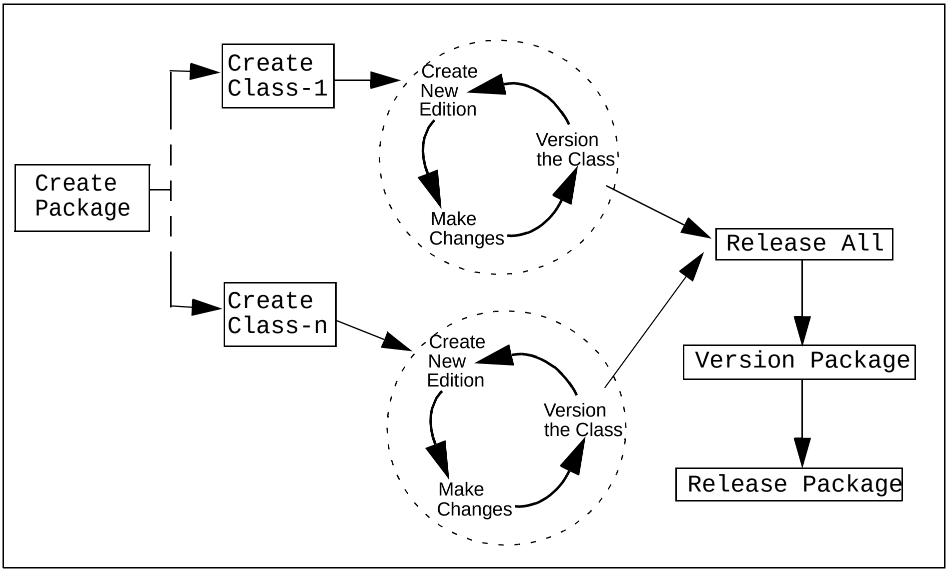


Figure 12. Single Developer Working with a Single Package

In this scenario you are a single user playing the role of package owner, class owner, and class developer. The project owner creates a package in the appropriate project and assigns ownership of the package to you. You create all the classes and interfaces required for the application. For each class you cycle through the standard development pattern by creating successive versions as you develop and debug the application.

Work continues in parallel on each class until you are completely satisfied that your application works correctly. Now you release all classes and interfaces into the package edition and create a version of the package. Finally, either you or the project owner releases the package.

Single Package, Multiple Developers

In this scenario multiple developers work on an application that is being developed as a single package. The scenario introduces the issues that arise when developers must cooperate with each other to do their work. Typically, in this situation, the lead developer acts as the package owner. The project owner creates the package and assigns ownership to the lead developer.

After the initial analysis of the problem domain has been completed and a first pass design has been carried out, the set of classes and interfaces that make up the application are known. The lead developer allocates responsibility to each developer for a number of classes. The lead developer adds the users to the package group so that they will have the authority to create classes and interfaces within the package.

Figure 13 shows the normal working pattern in this scenario.

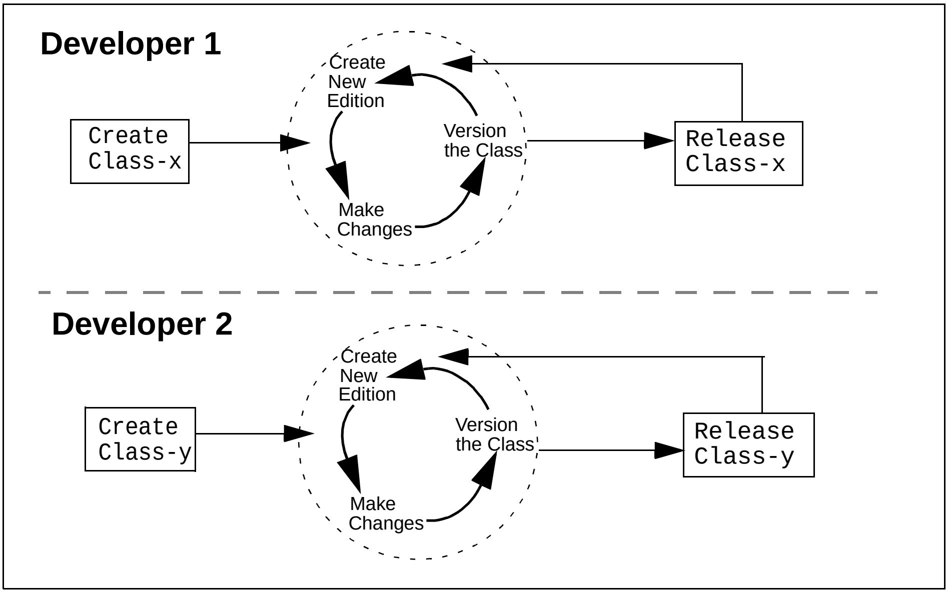


Figure 13. Working Pattern for Class Owners

When a developer creates a class or interface, he or she becomes the owner of that type. This is one of the key principles of the team environment. The ownership model reinforces the notion of responsibility. Other developers can make changes to a class they do not own, but, as we see later in this scenario, the owner must eventually accept or reject those changes and make them available to the rest of the team.

Each developer, who is also a class owner, works independently on the classes for which they have responsibility. The project manager has established a project plan that lays out various milestones in the application development process. These milestones generally represent the achievement of a certain level of functionality within the application and present an opportunity for all developers to synchronize with each other and have a common view of the current state of the overall application. This is normally called establishing a *baseline.* A baseline allows inconsistencies and discrepancies (which could be related to analysis, design, or development) to be caught early in the development process before they have become too entrenched. Baselines also allow regular deadlines to be set, which is good for programmer productivity!

When approaching a deadline, the lead developer expects programmers to have all their classes in a working state and instructs them to release the classes. The lead developer establishes a new package baseline by creating a new version of the package. Each developer then reloads the new package version to access the latest version of all classes in the package. This process is shown in Figure 14.

The class owners resume the normal development pattern until the next project milestone and the production of a new baseline.

Unfortunately, things do not always proceed smoothly. Invariably a developer working on a class finds that some new function is needed in another class or discovers a bug in another class that adversely affects his or her own progress. In these situations the developer does not have to immediately bother the owner of the other class in order to move forward. The developer simply creates a new edition of the class in question and makes any required changes in that edition. By creating a new edition of the class, the developer becomes the class developer of that private edition.

When the developer is satisfied with the changes, he or she creates a new version of that class and gives it a meaningful name indicating the source of the changes. The developer consults with the owner to discuss the changes and the reasons for them. It is now up to the class owner to decide whether those changes are valid and fit in with the overall purpose of the class. In a fast-moving development project, it is quite likely that the class in question has been further developed by the owner in the meantime. In this situation the owner must reconcile the changes and merge them into a new version (see Figure 15 on page 37). Once the changes have been reconciled, the owner resumes a normal development pattern.

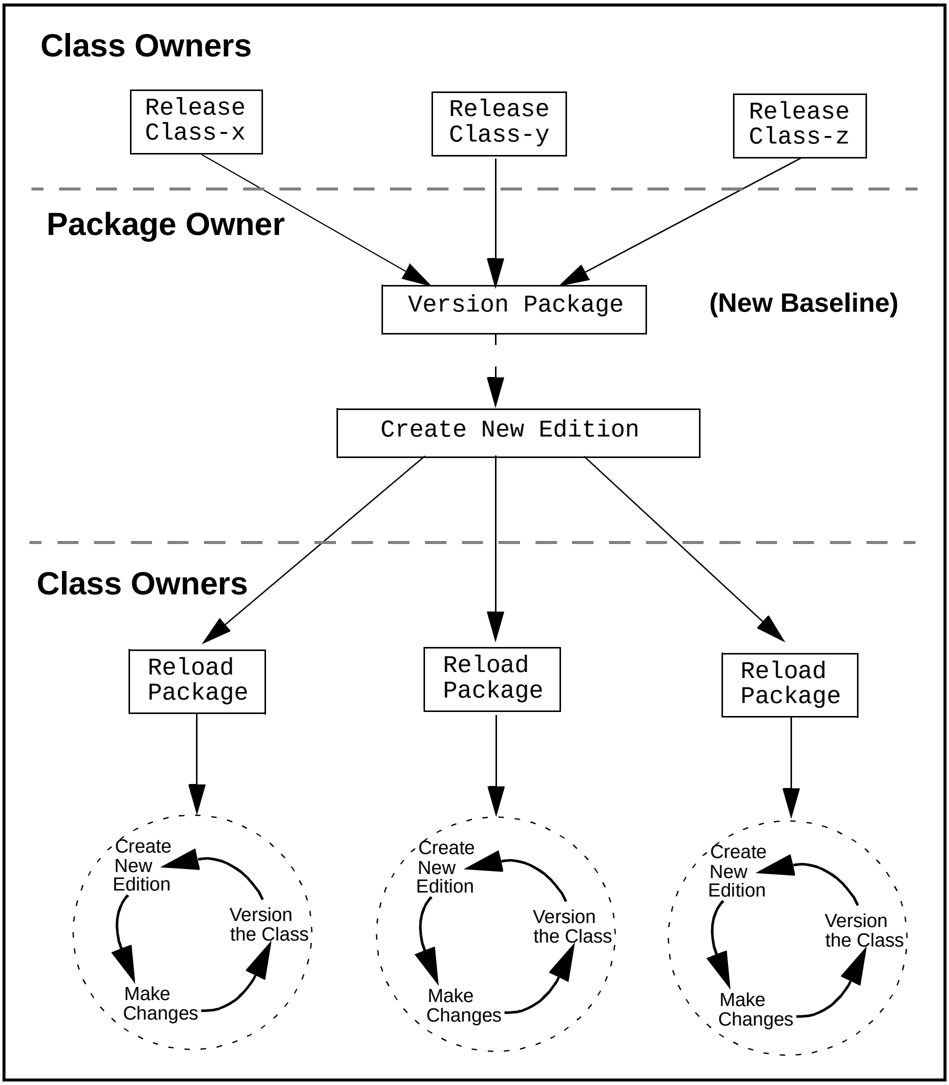


Figure 14. Establishing a Package Baseline

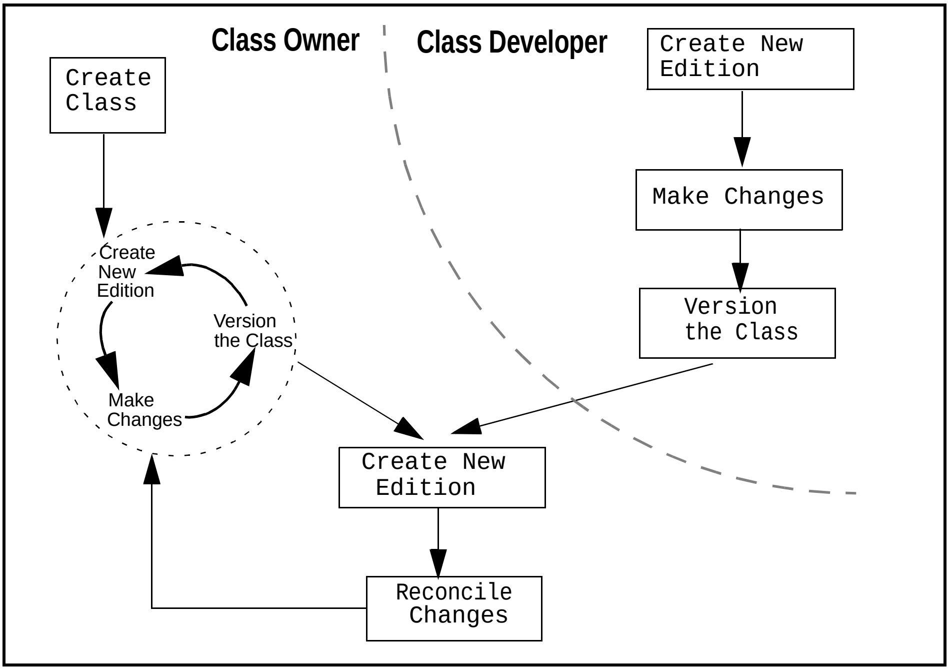
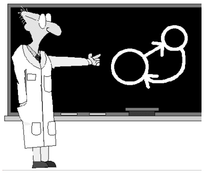


Figure 15. Merging Divergent Class Changes

Change reconciliation is made easy through the use of the comparison browser in VisualAge for Java. The owner views class and method definitions side-by-side with the differences highlighted. The owner selectively loads one or the other of the versions into a new edition. Where changes to a single method clash, the owner must manually edit the method to reconcile the changes.

Merging changes like this should take place on a regular basis. It is much less error prone to have frequent small reconciliations than it is to wait for a long time and try to reconcile a larger set of changes in one operation.

Tip	Do you have to wait for a baseline to see changes?
	Developers need not wait until a new baseline is established to see changes made to other classes. Informal interaction among team members is a normal daily occurrence. A class owner can announce to the colleagues that a new version of a class, *which is fit for general use,* is available. Other developers may explicitly load that version into their workspace. Creating a package baseline simply formalizes this process.

Multiple Packages, Multiple Developers

We now move on to a more complex scenario, which is typical of a large-scale project. In this scenario we have simultaneous development of multiple packages. The scope of the project is such that we have many distinct subsystems, which are naturally partitioned into separate packages. The project team is divided into smaller teams, each of which has responsibility for a particular package. The project owner is likely to be the chief architect and creates the packages and assigns ownership of each package to the corresponding team leader. Each team leader adds the team members to the package group.

All the patterns of development we have seen so far are also present in this scenario: the basic development pattern for classes; the establishment of package baselines; and the reconciliation of changes. Change reconciliation may also be necessary across package boundaries. There are always interpackage dependencies, and a developer of one package may have to change a class of another package. This is not a problem. Membership of a package group is not required in order to create a new edition of a class in that package and become the edition developer. The process of managing the reconciliation is exactly the same.

In this large-scale project scenario we now must consider managing project baselines. We define and discuss two useful and typical kinds of project baselines: *standard* and *rolling*.

Standard Project Baseline

A standard project baseline is directly analogous to a package baseline. With a standard project baseline you first create baselines of all the packages in the project. You then release all the package versions to the project. This step can be carried out by either the individual package owners or, more probably, by the project owner. Once all the packages have been released, the project owner creates a new baseline by versioning the project. This version represents an immutable state of the project, its packages, and all their classes. It is likely that this step will be carried out only at major project milestones and, of course, at the end of the project. Figure 16 shows the complete process.

If work is to continue after the creation of the baseline, a new project edition is created, and development continues.

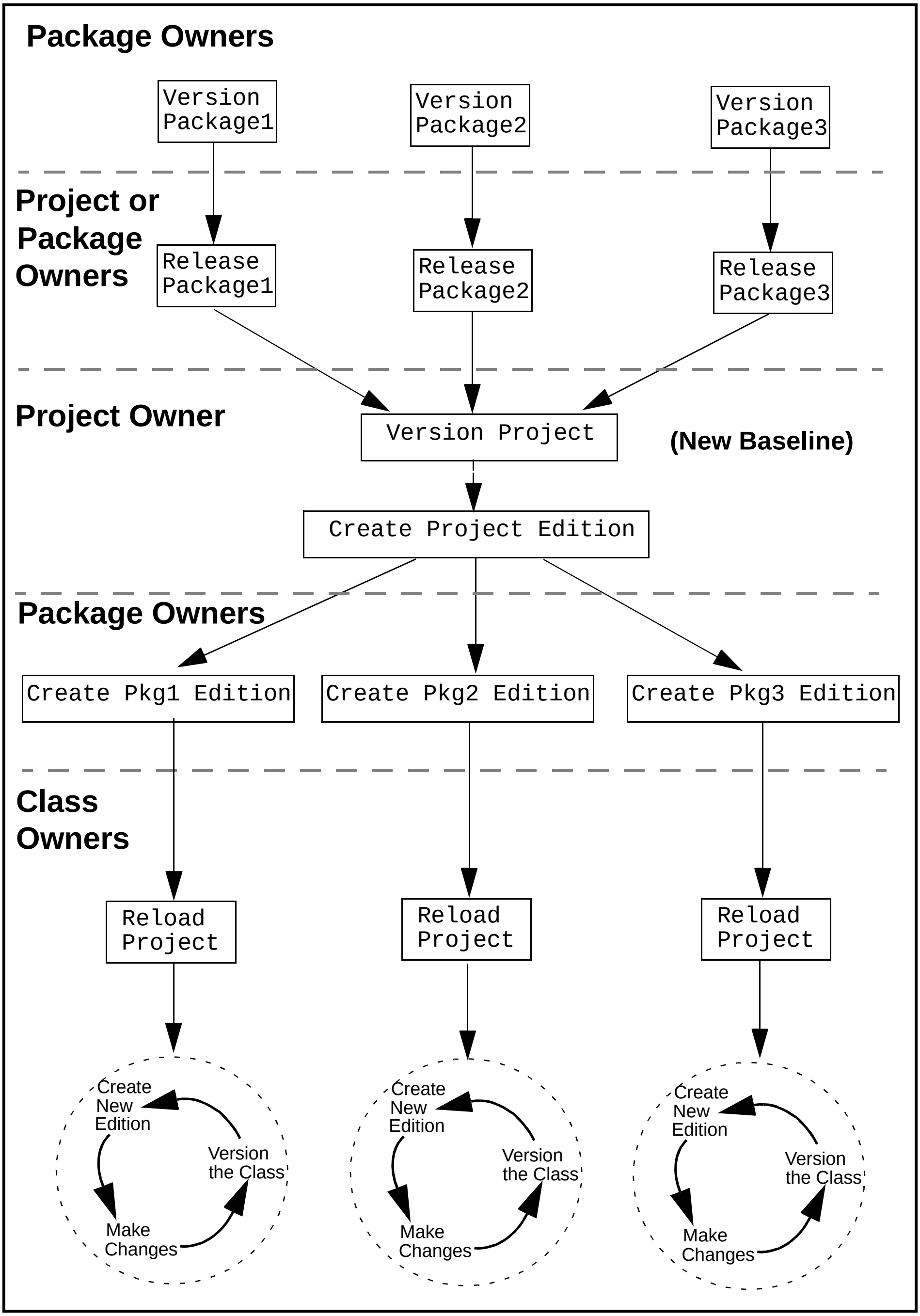


Figure 16. Establishing a Project Baseline

Rolling Project Baseline

The standard project baseline can be quite a burdensome process. The more packages you have in the project, the more difficult it is to ensure that you can produce a baseline for all packages at the same time.

Fortunately VisualAge for Java provides a much more convenient way for programmers to have access to an edition of the project that provides a snapshot of the current state of the application. Figure 17 shows the rolling project baseline process.

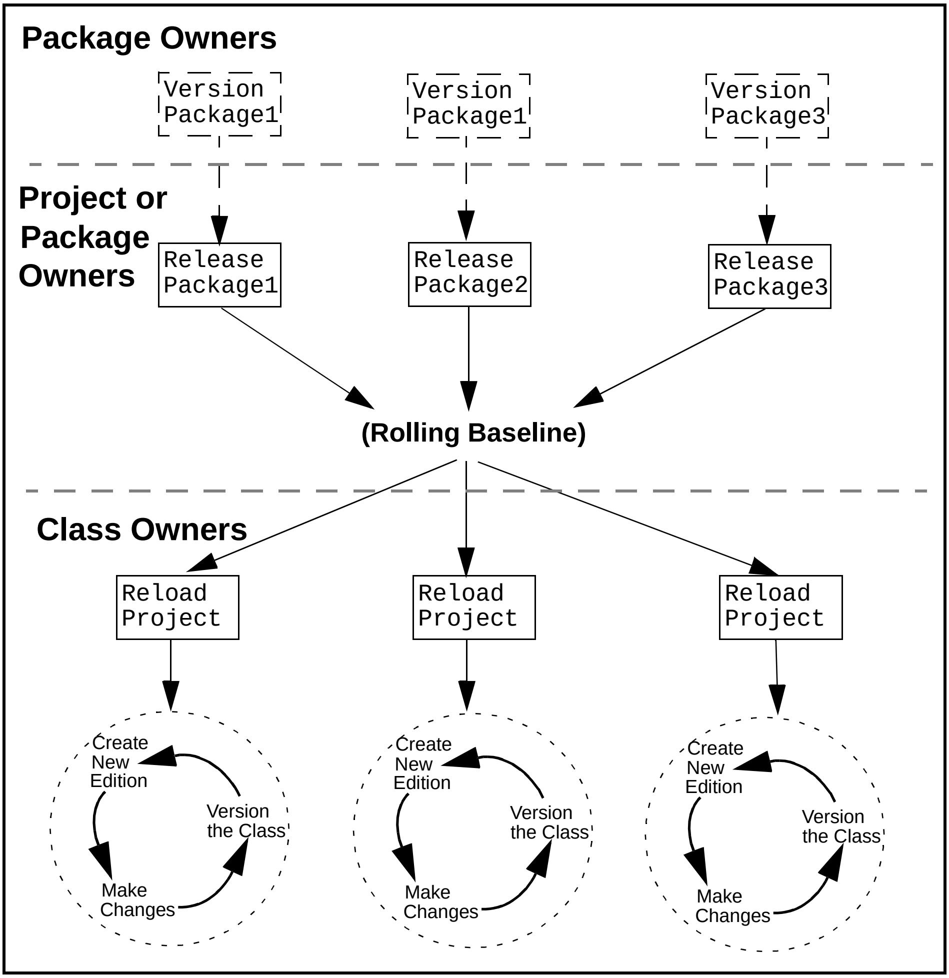
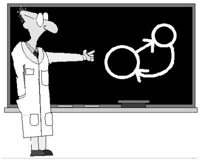


Figure 17. A Rolling Project Baseline

The key to the rolling project baseline process is that packages do not have to be versioned before they are released to the project edition. (In Figure 17 the broken boxes indicate optional steps.) The ramifications of this are quite profound. By releasing an open package edition to the project, you are ensuring that when a class is released into that package it automatically becomes available to any developer who reloads the current edition of the project into the workspace. Hence the use of the term *rolling*—the project baseline is constantly changing as new classes and interfaces are released.

Tip	Use caution with rolling project baselines!
	As with any powerful feature, caution should be exercised when using rolling project baselines. The release of a buggy class could cause inconvenience for any programmer who loads it. Releasing a class into a released package has the effect of bypassing any integration testing that would typically occur at the package level. The ability to freely combine the release of versioned packages and open edition packages into your project enables you to exercise a degree of fine tuning over the process. For example, open edition releases could be restricted to packages where the class owners are senior programmers and can be trusted to fully test their classes before release.

Multiple Parallel Streams

This scenario shows the power of VisualAge for Java Enterprise in handling complex development situations. Suppose you are part of a software house developing an application for a customer. The application is being developed as a single package. At a certain stage in the development process, your salesperson manages to sell the system to another customer. Naturally the system has to be tailored to the new customer’s needs.

You continue system development until all of the code that is common to both customers has been written and tested. You are now ready to enter split stream development mode. Using the tested and stable version of the package as a base, you create two new editions of the package: one for each customer. Each edition of the package can have a separate owner and package group associated with it (see Figure 18).

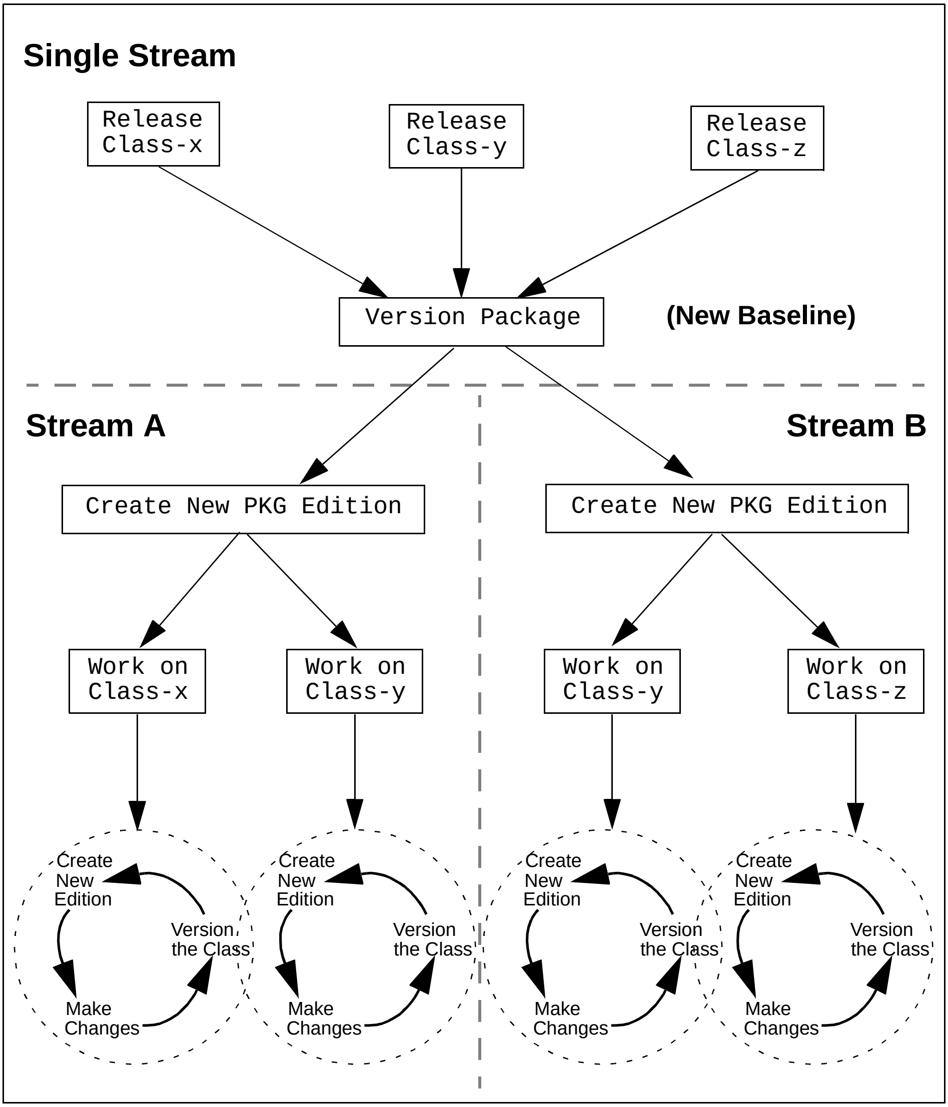


Figure 18. Split Stream Development

Development proceeds as if there were two distinct packages. The same class or interface can be modified independently in each package edition. The ownership of common classes can be modified to reflect the changed responsibilities. In essence, the owner of the class or interface edition is associated with the package edition that holds that edition.

With split stream development, there is no intention to rejoin the streams. The advantage is having a common code base from which both streams developed. If this code base has to change (as it invariably will), a new edition of the package version at the root of the split stream should be created. Each of the split streams can then be brought up-to-date by simply reloading the changed program element in question and performing any reconciliations that might be required.

Project Wrapup and Delivery

When a project has been completed, you must go through the process of delivering it to your customer. This process is usually the same whether you are a software house delivering to an external customer or an IT department delivering an internal system to your end users.

The first step is to produce a standard project baseline as described earlier. You may want to leave this version of the project as the reference release version. Alternatively you may want to set up a special project just to hold the released version, reinforcing the separation of development and released versions of the project. Reinforcement would simply be a matter of policy on your part. VisualAge for Java Enterprise guarantees the separation of different versions of the same project.

In either event, you should add extensive comments to the project edition, *before* versioning. We recommend that you establish a standard format for comments that must be added to a project edition before it is versioned for release.

When you have a completed project version, you should choose a delivery mechanism. VisualAge for Java Enterprise offers a number of options for exporting your work so that it can be delivered to the customer:

❑ Export .class and/or .java files

❑ Produce a JAR file

❑ Export to another repository

Exporting can include all of the required resources (see “Project Resources” on page 70) and generate HTML files for applets. These export options are described in the documentation accompanying the VisualAge for Java Enterprise product.

5 Team Java in Action

In this chapter we put some of the concepts and tasks that have been described in the previous sections into action. We start by telling you about our fictitious software development company that has adopted VisualAge for Java Enterprise as its standard development environment. We then outline the requirements for new development that the company is undertaking for an existing client. This new project gives us a context within which to present some typical activities as they might occur over the development life cycle.

Introducing JUM Software Associates

JUM Software Associates is an independent software vendor specializing in object-oriented systems development. It has historically done both C++ and Smalltalk development but in the last couple of years has begun to invest in Java, which is now starting to become a major portion of its business.

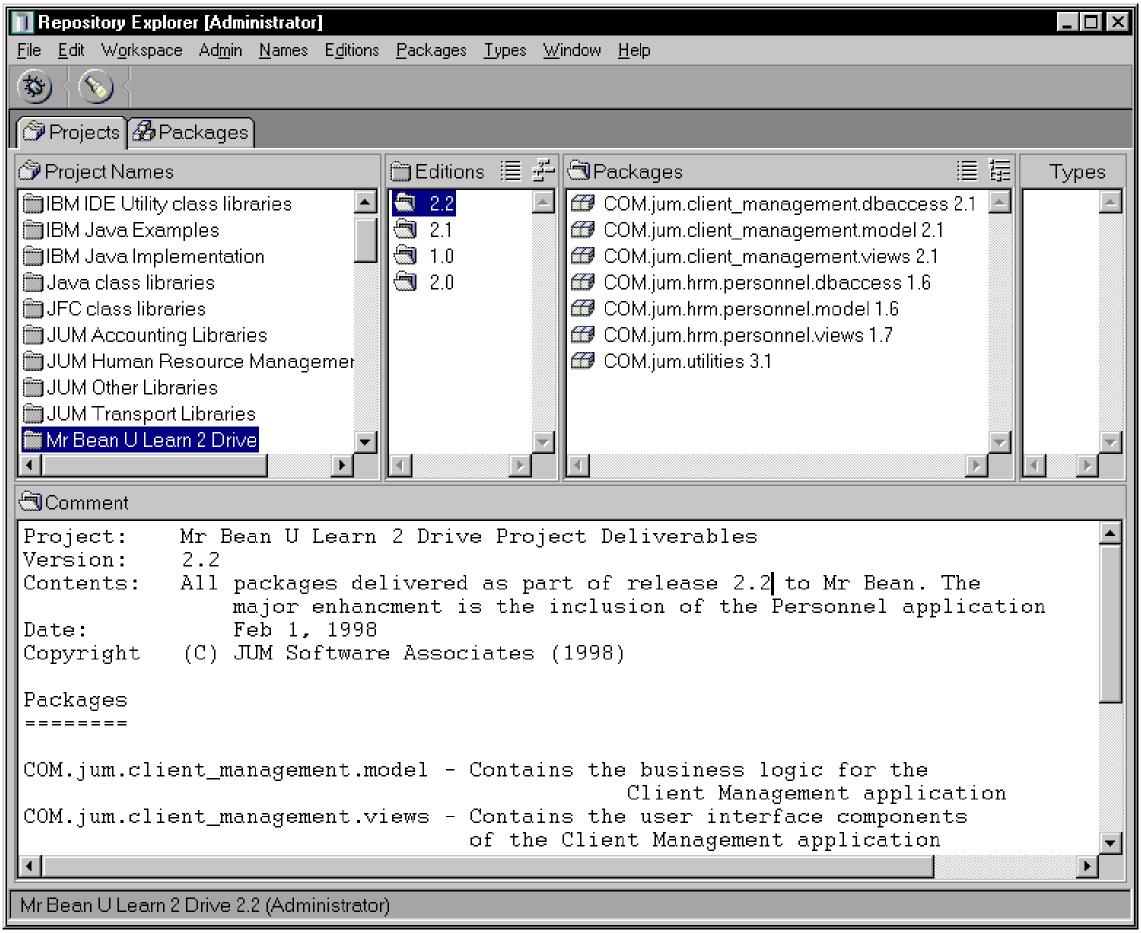
JUM sells a couple of software packages that it has developed for small businesses. The offerings focus on the basics of managing a small business, implementing traditional general ledger, accounting, and payroll functions. Although JUM sells its software off-the-shelf, a large portion of its clients request customization and implementation services, which JUM routinely performs.

Current Environment

JUM has been using VisualAge for Java for all its Java application development since the product was initially available. It currently has all of its projects organized within one central repository. It is likely though that the company may move to multiple repositories as its team and work products grow.

For each client deliverable, a project edition is created that contains all of the required packages for the application. On project completion, the project is versioned, and the owner is changed to the administrator. When a client requests an update (either a fix or enhancement), the administrator assigns a new project owner appropriately. The new owner takes over responsibility for the new edition of the project through customer delivery.

Figure 19 shows us a view of the JUM Repository.


Repository Explorer [Administrator]
File Edit Workspace Admin Names Editions Packages Types Window Help
Projects
Packages
Project Names
Editions
Packages
Types
Comment

IBM IDE Utility class libraries
IBM Java Examples
IBM Java Implementation
Java class libraries
JFC class libraries
JUM Accounting Libraries
JUM Human Resource Managemer
JUM Other Libraries
JUM Transport Libraries
Mr Bean U Learn 2 Drive

2.2
2.1
1.0
2.0

COM.jum.client_management.dbaccess 2.1
COM.jum.client_management.model 2.1
COM.jum.client_management.views 2.1
COM.jum.hrm.personnel.dbaccess 1.6
COM.jum.hrm.personnel.model 1.6
COM.jum.hrm.personnel.views 1.7
COM.jum.utilities 3.1

```
Project:    Mr Bean U Learn 2 Drive Project Deliverables
Version:    2.2
Contents:   All packages delivered as part of release 2.2 to Mr Bean. The
                major enhancment is the inclusion of the Personnel application
Date:           Feb 1, 1998
Copyright   (C) JUM Software Associates (1998)

Packages
========

COM.jum.client_management.model - Contains the business logic for the
                                           Client Management application
COM.jum.client_management.views - Contains the user interface components
                                  of the Client Management application
```

Mr Bean U Learn 2 Drive 2.2 (Administrator)

Figure 19. A View of the JUM Repository

A New Project for Mr. Bean’s U-Learn 2-Drive School

Mr. Bean’s U-Learn 2-Drive School runs a driver’s education program that focuses on preparing its students for license testing through road instruction and practice.

Mr. Bean’s U-Learn 2-Drive School is an existing client of JUM and has already purchased JUM’s Human Resource Management applications.

Application Requirements

Mr. Bean has come back to JUM with requirements for a new function to be added to his current system. He now wants to do online scheduling of his cars, clients, and instructors. The developers at JUM have analyzed these requirements, prepared a project plan, and received approval from Mr. Bean to go ahead with the new development.

The Project Team

With approval from Mr. Bean to proceed with the new development, a project team consisting of the following members is assembled at JUM. (The titles we use for the project team members are typical of what we see in many development organizations.)

- ❑ Project manager - Peter
- ❑ Project leader - Mark
- ❑ Architect - Mark
- ❑ User interface specialist - Janice
- ❑ Database specialist - Uwe
- ❑ Developers - Janice, Uwe, Mark, Luc, Tasha, and Jui

These people form the core team that will be responsible for the design, development, and delivery of Mr. Bean's scheduling system. Additionally, another JUM employee, Anne, is part of the extended team. Anne is a system administrator and has been managing JUM's VisualAge for Java Enterprise development environment.

VisualAge for Java Roles

Now we examine how we can assign each of these team members to the various roles in the VisualAge for Java Enterprise team environment. Although some of the role titles are the same as real-world project titles, the roles are not. Therefore we provide details for each role to distinguish between *real-world project roles* and VisualAge for Java Enterprise *team roles*. Each team member will play at least one, but probably many VisualAge for Java roles during the course of a project life cycle.

<u>Administrator</u>: At JUM there is one administrator for all of the VisualAge for Java development, our system administrator, Anne. Anne manages the repository files and ensures that the VisualAge for Java Enterprise server process is running and that new users and passwords are included in the configuration.

Project owner: Mark has overall responsibility for all VisualAge for Java Enterprise aspects of the Mr. Bean project. He creates the project in the repository and adds packages to it. For each package he defines the user who will be the package owner and he changes the package ownership to that user. He is responsible for the integrity of the project (ensuring that correct versions of all packages are included and at the right version).

Package owners: As the senior developers on the team, Mark, Uwe, and Janice are package owners. It is their responsibility to ensure that their packages work correctly and to eventually version and potentially release them.

Class owners: Any user who creates a class or interface automatically becomes the owner of that program element. So it is likely that all developers on the team will be class or interface owners in the packages where they are a member of the package group.

Class developers: Any developer who creates a new edition of a class or interface becomes the developer of that program element. All developers on the Mr. Bean project will be class developers.

Notice that the real-world project manager, Peter, does not have any role to play in the VisualAge for Java Enterprise world. This is to be expected because VisualAge for Java is concerned with the project's implementation, and Peter is more likely to be carrying out his managerial role.

Project Life Cycle

In this section we describe in detail some activities that would typically be part of the VisualAge project development phase. You may find it helpful to step through these activities in your own VisualAge for Java environment. Although you do not have the same software elements in your repository or workspaces, you can execute the tasks with any existing packages and classes, or you can choose to create the software elements as you see them in the screen images that are presented.

We assume that you know the details of how to complete a specific task. For example, where we write *Create a new project*, we omit these details:

❑ Go to the Projects view of the Workbench.
❑ Bring up the pop-up menu by clicking the right mouse button once.
❑ Select *Add Project...* to bring up the SmartGuide where you enter a new project name.
❑ Click on **Finish** to have the new project created in your workspace.

We assume that you can execute these basic tasks without this level of detail.

The sample activities presented here are simplistic in terms of the actual programming aspects; in fact, they really do not implement any application function. The Mr. Bean U-Learn 2-Drive School scenario has been created to illustrate the functions and flexibility of the team development environment; the actual class and method contents are not important.

A VisualAge for Java Enterprise role and user name in brackets, as in [Project owner—theUserName], preceding a series of steps indicates that the subsequent steps are performed in that user's workspace.

Sample Activity 1: Creating an Initial Baseline

At the beginning of any project, some startup activities must be undertaken to prepare the environment for the team.

JUM uses projects to manage its customer application deliverables. In this first activity, we prepare a new project for development and create an initial baseline.

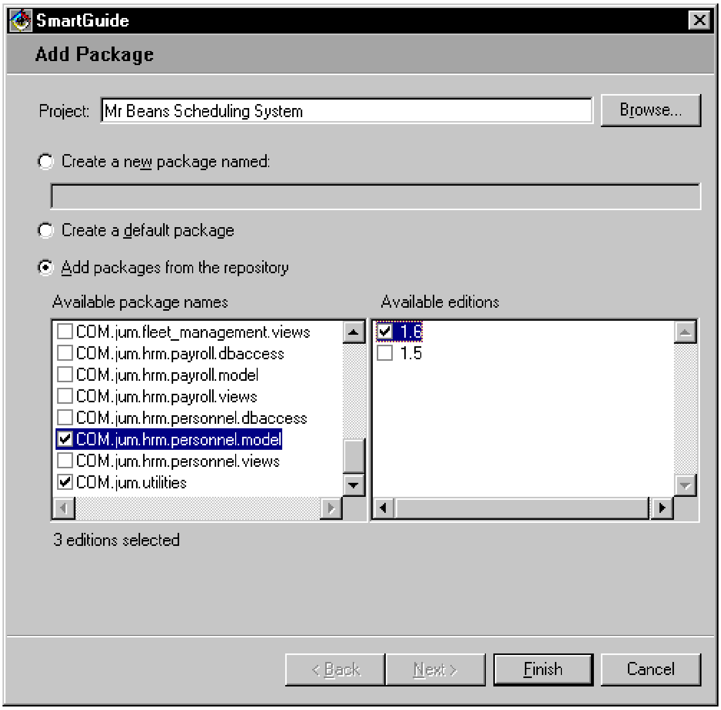
[Project owner—Mark]

1. From the Projects view of the Workbench, create a new project, Mr. Bean's Scheduling System, to contain all packages (new and existing) that make up this development effort.
2. Add a project comment in the comment source pane of the Projects page of the Workbench. It is important to use this section of the project to document the contents, date, purpose, enhancements, fixes, and other important details. Some of the projects shipped with the VisualAge for Java Enterprise product have included these comments, and you may want to use them as a template for your own. See the *IBM Enterprise Access Builder Libraries* and the *IBM Java Examples* projects in the repository. See Figure 19 on page 47 for a view of the repository which includes the project comment.
3. Add existing packages to the project.

   You will probably want to reuse packages previously developed and stored in the repository. In our Mr. Bean example, we add three packages from previous projects:

   - COM.jum.client_management.model 2.1
   - COM.jum.hrm.personnel.model 1.6
   - COM.jum.utilities 3.0

   Figure 20 shows the SmartGuide, where you can add existing packages from the repository to your project.



Project: Mr Beans Scheduling System

○ Create a new package named:

○ Create a default package

● Add packages from the repository

Available package names

- □ COM.jum.fleet_management.views
- □ COM.jum.hrm.payroll.dbaccess
- □ COM.jum.hrm.payroll.model
- □ COM.jum.hrm.payroll.views
- □ COM.jum.hrm.personnel.dbaccess
- ☑ COM.jum.hrm.personnel.model
- □ COM.jum.hrm.personnel.views
- ☑ COM.jum.utilities

Available editions

- ☑ 1.6
- □ 1.5

3 editions selected

Figure 20. Adding a Package from the Repository

4. Add new packages to the project.

   In the Add Package SmartGuide (see Figure 20), select *Create a new package named:* and give a new package name. Your package name should be consistent with your existing naming conventions, but this consistency is not enforced by VisualAge for Java.

5. Add members to the package group.

   The SmartGuide gives you the option of adding new members at the time the new package is created. Alternatively, you can add or delete users at a later stage. In our example, we will perform our development in the three new packages that we add:

   - `COM.jum.operations.scheduling.model`

     The group members for this package are Marc, Janice, Luc, and Uwe.

   - `COM.jum.operations.scheduling.views`

     The group members for this package are Mark and Janice.

   - `COM.jum.operations.scheduling.dbaccess`

     The group members for this package are Mark and Uwe.

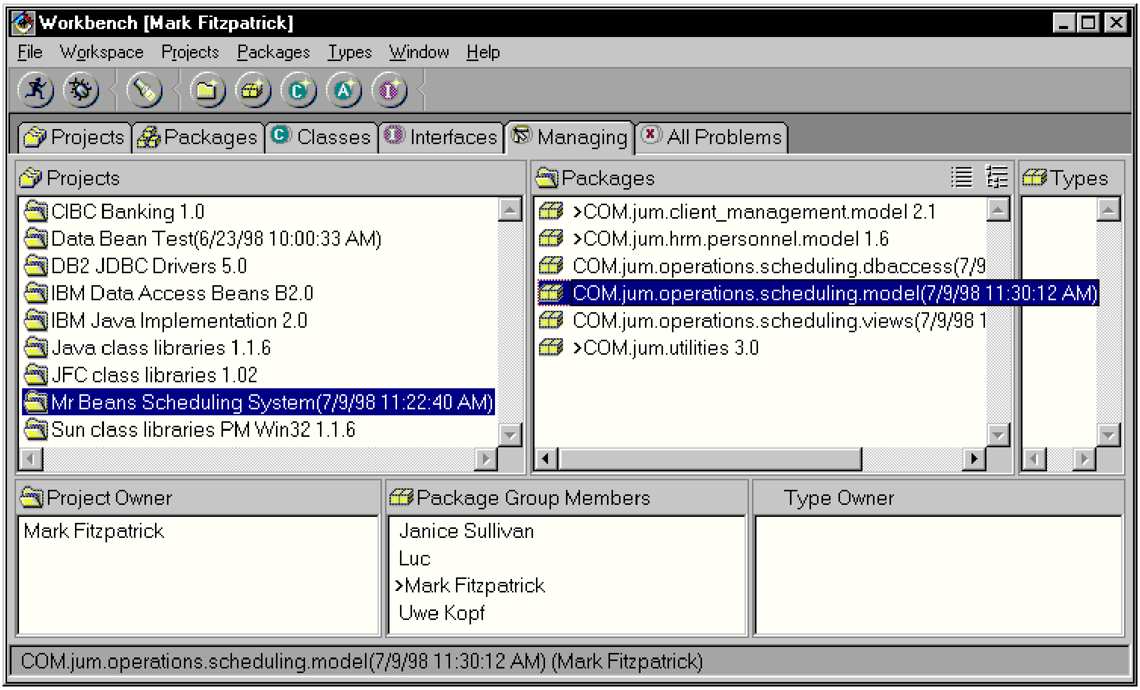
6. Set the package owners.

   By default the user who creates a package is the owner. In the Mr. Bean project, Mark is the owner of all our new packages. He can now change ownership of a package to one of the other group members. Ownership change is done in the Managing view of the Workbench (*Packages > Manage > Change Owner...*).

   We now change ownership to Uwe, for the dbaccess package and Janice, for the views package.

7. Release the existing packages into the project.

   When a new package is created within a project, it is automatically released. This is not the case when you add existing packages from the repository. Notice in Figure 21 that the packages that already existed are prefixed with ">". This mark indicates that the software element has not been released to its containing component.



CIBC Banking 1.0
Data Bean Test(6/23/98 10:00:33 AM)
DB2 JDBC Drivers 5.0
IBM Data Access Beans B2.0
IBM Java Implementation 2.0
Java class libraries 1.1.6
JFC class libraries 1.02
Mr Beans Scheduling System(7/9/98 11:22:40 AM)
Sun class libraries PM Win32 1.1.6

>COM.jum.client_management.model 2.1
>COM.jum.hrm.personnel.model 1.6
COM.jum.operations.scheduling.dbaccess(7/9
COM.jum.operations.scheduling.model(7/9/98 11:30:12 AM)
COM.jum.operations.scheduling.views(7/9/98 1
>COM.jum.utilities 3.0

Mark Fitzpatrick

Janice Sullivan
Luc
>Mark Fitzpatrick
Uwe Kopf

COM.jum.operations.scheduling.model(7/9/98 11:30:12 AM) (Mark Fitzpatrick)

Figure 21. Released and Unreleased Packages in the Mr. Bean Project

Release the packages by selecting *Manage > Release* from the pop-up menu of the Packages pane. By selecting multiple packages, you can release all of them in one step.

8. Each member of the team now does an *Add Project...* from his/her own Workbench and selects the new Mr. Bean project from the repository. This project edition represents an initial baseline for the development of the new application.

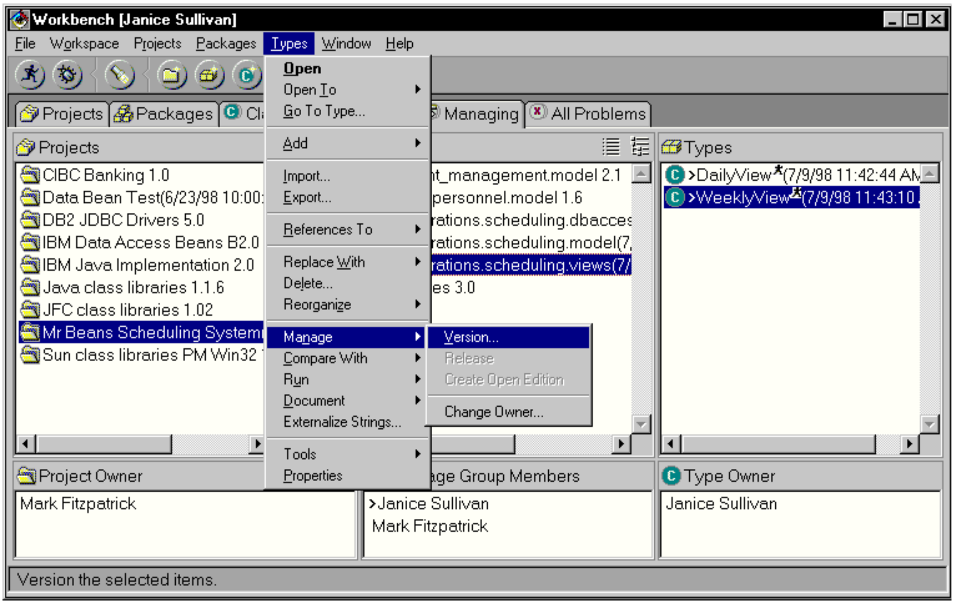
Sample Activity 2: Developing New Classes

The most common activity that occurs in any application development project is the development of new classes and interfaces within your packages.

The package group members are in fact the developers of a package. As a matter of routine, each of those developers creates classes inside the package, using the standard class creation mechanisms and VisualAge for Java tools. When a class is created, the group member who created that class is the owner and class developer. The class developer versions the class edition periodically to perform unit testing or in preparation for releasing it into the package.

[Class developer—Janice]

1. Create the DailyView and WeeklyView classes in the views package. This is where the actual development of the new code takes place.
2. Version the classes, using the versioning dialog (Figure 22).



CIBC Banking 1.0
Data Bean Test(6/23/98 10:00:
DB2 JDBC Drivers 5.0
IBM Data Access Beans B2.0
IBM Java Implementation 2.0
Java class libraries 1.1.6
JFC class libraries 1.02
Mr Beans Scheduling System
Sun class libraries PM Win32

nt_management.model 2.1
personnel.model 1.6
rations.scheduling.dbacces
rations.scheduling.model(7,
rations.scheduling.views(7/
es 3.0

>DailyView(7/9/98 11:42:44 AM
>WeeklyView(7/9/98 11:43:10

Mark Fitzpatrick

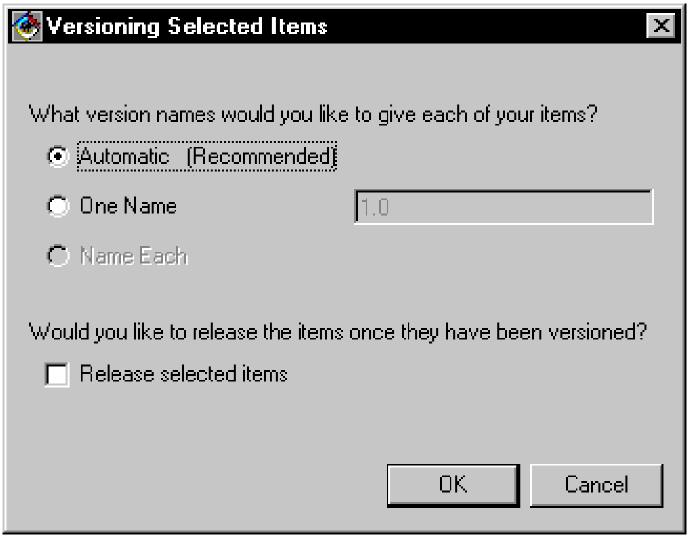
>Janice Sullivan
Mark Fitzpatrick

Janice Sullivan

Version the selected items.

Figure 22. Versioning Classes

In the SmartGuide you decide on the naming convention for the versioned classes. You can also release your classes at the same time (see Figure 23).



Versioning Selected Items

What version names would you like to give each of your items?

● Automatic (Recommended)

○ One Name 1.0

○ Name Each

Would you like to release the items once they have been versioned?

☐ Release selected items

Figure 23. SmartGuide for Versioning

3. Release the classes. If you choose not to release the classes as part of the versioning SmartGuide, you can explicitly release them from the *Manage > Release* menu option.

Sample Activity 3: Modifying Existing Program Elements

Sometimes you have to introduce changes to classes you do not own in order to make them work with your new classes. You do not have to involve the class owner at this stage. Instead you can create a new edition of the class you want to amend and make your changes there.

Recall...	Who owns a class edition?
	Once a developer has created an edition of a class, he or she becomes the class developer and essentially “owns” that edition. No other developer, including the class owner, can make changes in that edition. Only when it comes time to release a class into the containing package does the class owner become involved.

In the scenario that we step through next, we will be working in the COM.jum.utilities package, which is owned by Uwe. For illustrative purposes, we copied a couple of classes into this package from other packages of the standard VisualAge for Java environment. Once copied, these new classes became owned by Uwe.

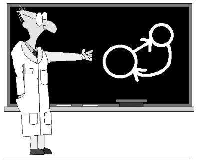
[Class developer—Uwe]

❑ If you want to run through this example in your own workspace, you can copy the classes from the COM.ibm.ivj.examples.vc.mortgageamortizer package into your own package. Select the classes and use the *Reorganize > Copy...* function. Version the COM.jum.utilities package before you continue.

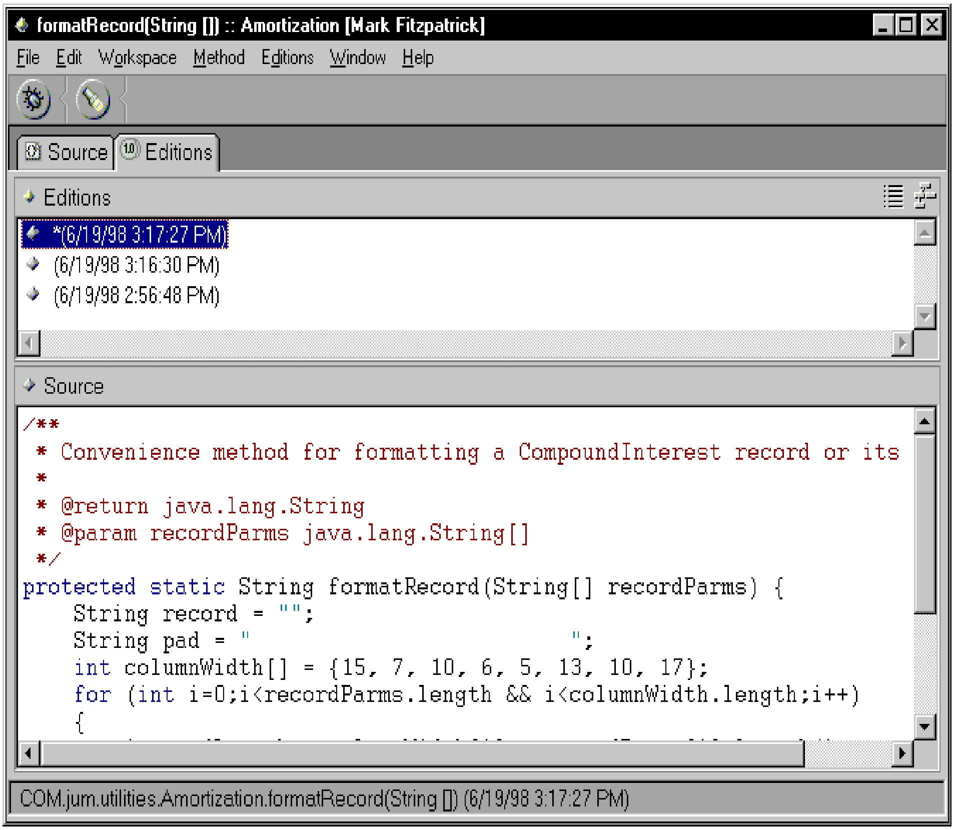
Mark changes the Amortization applet to change the format that is used to display results of the calculations.

[Class developer—Mark]

1. Create a new edition of the Amortization class (*Manage > Create Open Edition*). This causes a scratch edition of the package to be created in Mark's workspace because the package was a version. When opening the properties of this class, we see that Uwe is the owner and Mark is the developer.

Tip	What does the status line display?
	When you select a class or interface in any of the Workbench views, notice that the status line at the bottom shows you some information about that program element: the fully qualified name, the edition name, and the name of the developer. If you select a project or a package in the Workbench, the status line shows what looks like the same information, but note that the user name displayed is the owner of the project or package, not the developer. Opening the properties of any program element displays all available information.

2. Change the formatRecord(String[]) method. We adjust the initialization of the columnWidth array just to introduce a small change for illustrative purposes. On saving the method, a new edition of the method is created. This is not obvious from the Workbench or Class browser because the time stamps of the methods are not shown. However, if you select *Open To > Editions* from the pop-up menu of a selected method, you open a methods browser that lists all editions along with their source (Figure 24).


formatRecord(String []) :: Amortization [Mark Fitzpatrick]
File Edit Workspace Method Editions Window Help
Source
Editions
Editions
Source

*(6/19/98 3:17:27 PM)
(6/19/98 3:16:30 PM)
(6/19/98 2:56:48 PM)

```
/**
 * Convenience method for formatting a CompoundInterest record or its
 *
 * @return java.lang.String
 * @param recordParms java.lang.String[]
 */
protected static String formatRecord(String[] recordParms) {
    String record = "";
    String pad = "                                    ";
    int columnWidth[] = {15, 7, 10, 6, 5, 13, 10, 17};
    for (int i=0;i<recordParms.length && i<columnWidth.length;i++)
    {
```

COM.jum.utilities.Amortization.formatRecord(String []) (6/19/98 3:17:27 PM)

Figure 24. Browsing Method Editions

Every save of a method updates the edition in your workspace and creates a new method edition in the repository. All old method editions remain visible in the repository until the containing package is purged.

3. Version the Amortization class (*Manage > Version*)

   Although the option to release the class is not grayed out in the dialog, do not use it because Mark is not the owner of the Amortization class. Only the class owner can release a class into the containing package.

Sample Activity 4: Consolidating Class Changes for Release

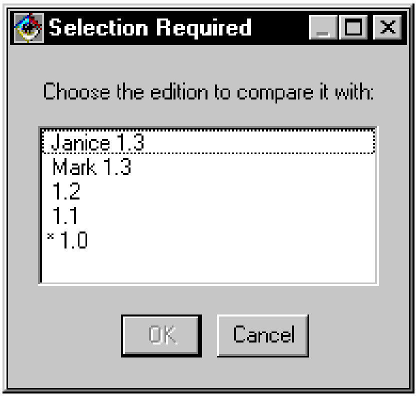
When changes have been made to a class by a developer who does not own the class, the class owner must go through a consolidation step in order to release the class to the package. Recall that the class owner is responsible for the integrity of that class. Therefore the developer must understand the changes and how they affect the class as a whole. It is for this reason that only the owner can release a version so that it is available to other developers. It is also the reason why we recommend that the number of developers of a class be kept to a minimum.

The owner of the Amortization class is Uwe. Janice and Mark have advised him that each of them has made changes to the class, which they have now versioned. Uwe must now reconcile those changes, which are in two separate versions, into a single version that can be released. Uwe has the original version (the one on which both Mark and Janice based their changes) loaded in his workspace. He uses this version as a base to compare with the others.

[Class owner—Uwe]

1. Compare changes.

   With the Amortization class selected in the Workbench, select *Compare With... > Another Edition* to present a list of all the editions of this class (Figure 25).


Selection Required

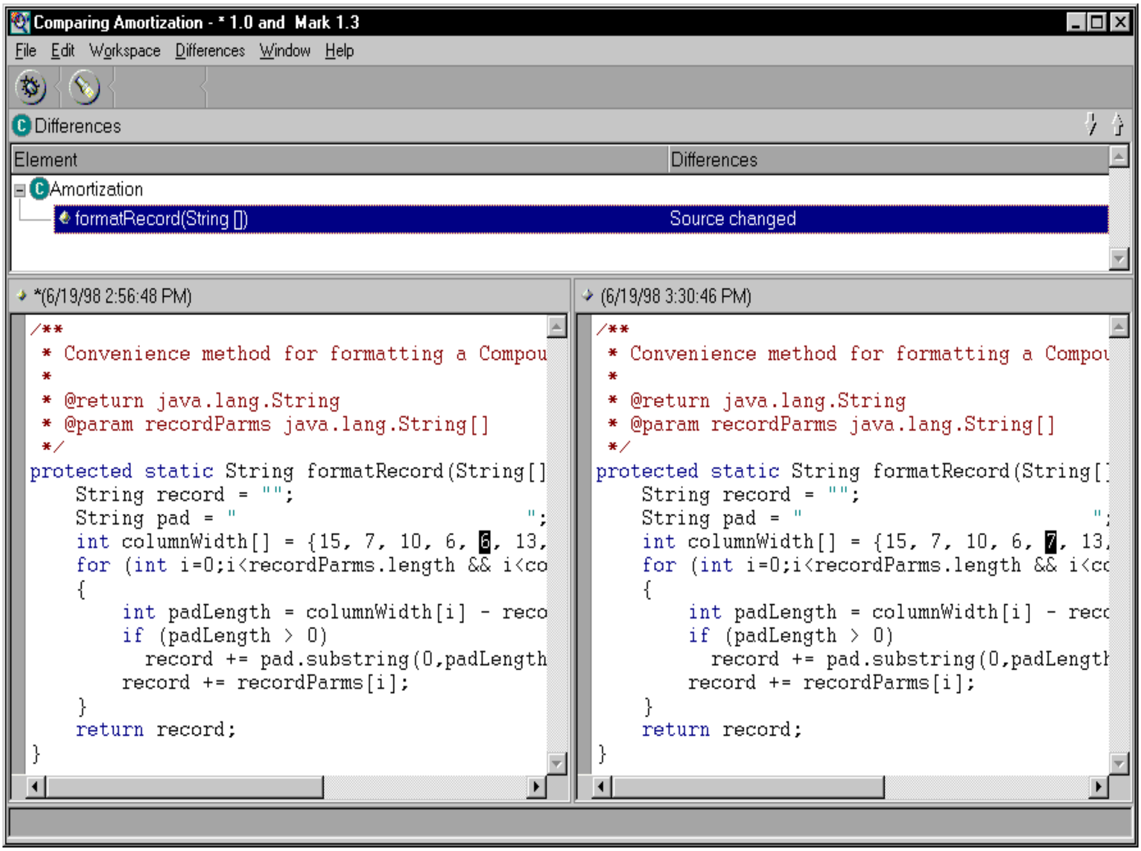
Choose the edition to compare it with:

Janice 1.3
Mark 1.3
1.2
1.1
*1.0

Figure 25. List of Versions of the Amortization Class

2. Select Mark 1.3 as the edition to compare with first.

   The Comparison browser opens showing us the differences between the two editions (Figure 26).



Element	Differences
Amortization	
formatRecord(String [])	Source changed

*(6/19/98 2:56:48 PM)

```
/**
 * Convenience method for formatting a Compou
 *
 * @return java.lang.String
 * @param recordParms java.lang.String[]
 */
protected static String formatRecord(String[]
    String record = "";
    String pad = "                          ";
    int columnWidth[] = {15, 7, 10, 6, 6, 13,
    for (int i=0;i<recordParms.length && i<co
    {
        int padLength = columnWidth[i] - reco
        if (padLength > 0)
          record += pad.substring(0,padLength
        record += recordParms[i];
    }
    return record;
}
```

(6/19/98 3:30:46 PM)

```
/**
 * Convenience method for formatting a Compou
 *
 * @return java.lang.String
 * @param recordParms java.lang.String[]
 */
protected static String formatRecord(String[
    String record = "";
    String pad = "                          ";
    int columnWidth[] = {15, 7, 10, 6, 7, 13,
    for (int i=0;i<recordParms.length && i<c
    {
        int padLength = columnWidth[i] - rec
        if (padLength > 0)
          record += pad.substring(0,padLengt
        record += recordParms[i];
    }
    return record;
}
```

Figure 26. Comparing Differences between Two Class Editions

Inside the Comparison browser, there are two source panes where the differences in the code are highlighted. The left pane shows the version edition that is loaded in the workspace, and the right pane shows the version selected for comparison from the repository. There are two arrows in the top right corner. You can use the arrows to step through the differences one at a time. In our example there is only one difference.

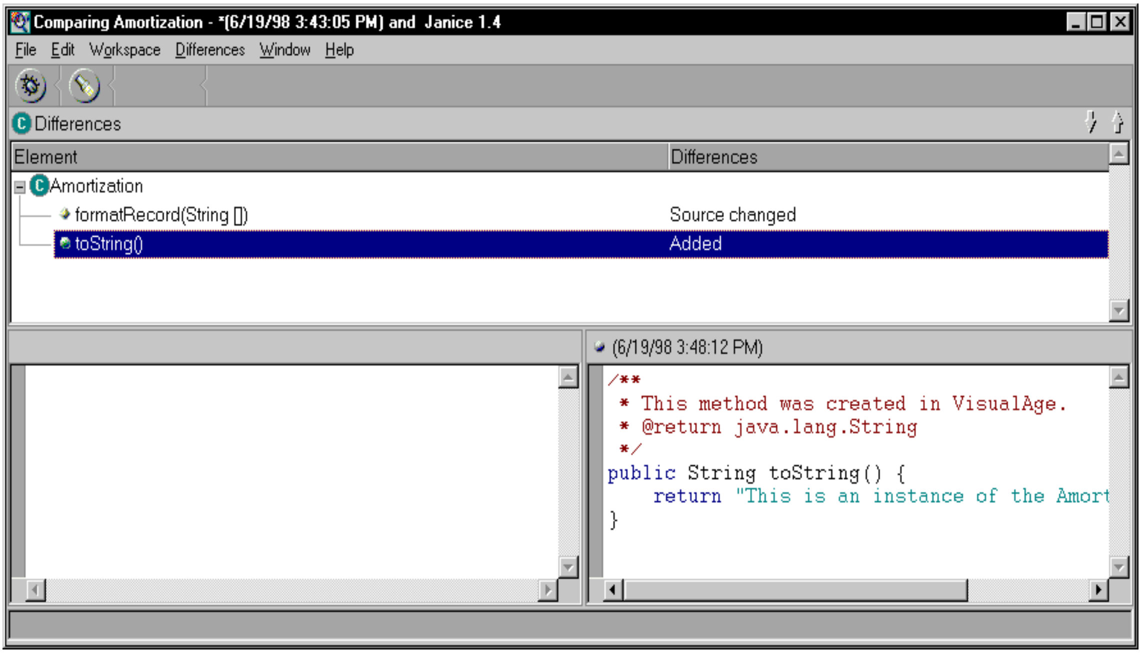
3. Merge the updates.

After reviewing Mark's changes, Uwe decides to load them into his workspace. He positions the cursor in the Differences pane (at the top) and selects *Load Right* from the pop-up menu. This action automatically creates a new edition of the Amortization class and a scratch edition of the package in Uwe's workspace.

Important Info	Merging of changes
	In practice the class owner reviews all changes to understand the impact they may have on one another and on the behavior of the class as a whole. The class owner resolves any potential conflicts before merging the changes into a common version.

4. Repeat this compare and merge process with Janice's version of the Amortization class.

   Figure 27 shows us that there are now two differences between these editions. (Janice has added the toString method in the meantime.)



```
/**
 * This method was created in VisualAge.
 * @return java.lang.String
 */
public String toString() {
    return "This is an instance of the Amort
}
```

Figure 27. Comparing Differences with a Second Version

Ignore the first difference because it refers to the change we previously loaded. Select the *Ignore* option from the pop-up menu when the mouse is positioned on the first change line. This action removes the line from your view, but you can have it shown again (in brackets) by selecting *Show Ignored Items*. The second difference is the new method that Janice implemented in her version. To load this new method, select *Load Right*.

5. Now convert the scratch edition of the COM.jum.utilities package to a new edition (*Manage* > *Create Open Edition*).

   This conversion is necessary because you cannot release a class into a scratch edition of a package.

6. Version and release the Amortization class.

   We can do this in one step because Uwe is the class owner.

7. Release the package.

   When all classes in a package have been versioned and released, the package is ready for versioning and/or release into the project. It is not necessary to create a version before releasing. Releasing the package without prior versioning allows the project owner to build rolling baselines for the team throughout the development cycle. These updated baselines keep developers in step with one another and enable integration testing across package editions.

Attention!	To version or not to version?
	At first glance, it may seem inconsistent that VisualAge for Java Enterprise allows open editions of packages to be released into a project when you know that only versioned classes can be released into their containing package. On closer examination though, this turns out to be a very powerful feature. It allows package owners to include editions that are still under development in new project baseline editions so they can be tested in a comprehensive manner without cluttering the repository with many unnecessary package versions. As a package owner, you should version your package when it is in a logically completed state that you want to preserve. Do not version the package if you want only to include the package edition in a new project baseline.

Sample Activity 5: Creating a Final Baseline for Application Delivery

At the end of your project you will have to deliver an application. At JUM the customer deliverables are organized into VisualAge for Java projects. We created a project for Mr. Bean's Scheduling System at project startup, and it is this project that we will now deliver to Mr. Bean.

As the project owner, Mark is responsible for coordinating the building of this final project version and ensuring its integrity.

[Project owner—Mark]

1. Ensure that all packages contained in Mr. Bean's scheduling system are versioned and released.

   Mark meets with Janice and Uwe as the other package owners within this project and has them update their package comments and release the correct versions of their packages. As a package owner himself, Mark also must release his package into the project.

2. Verify that the comments associated with this current project edition are accurate and up-to-date.

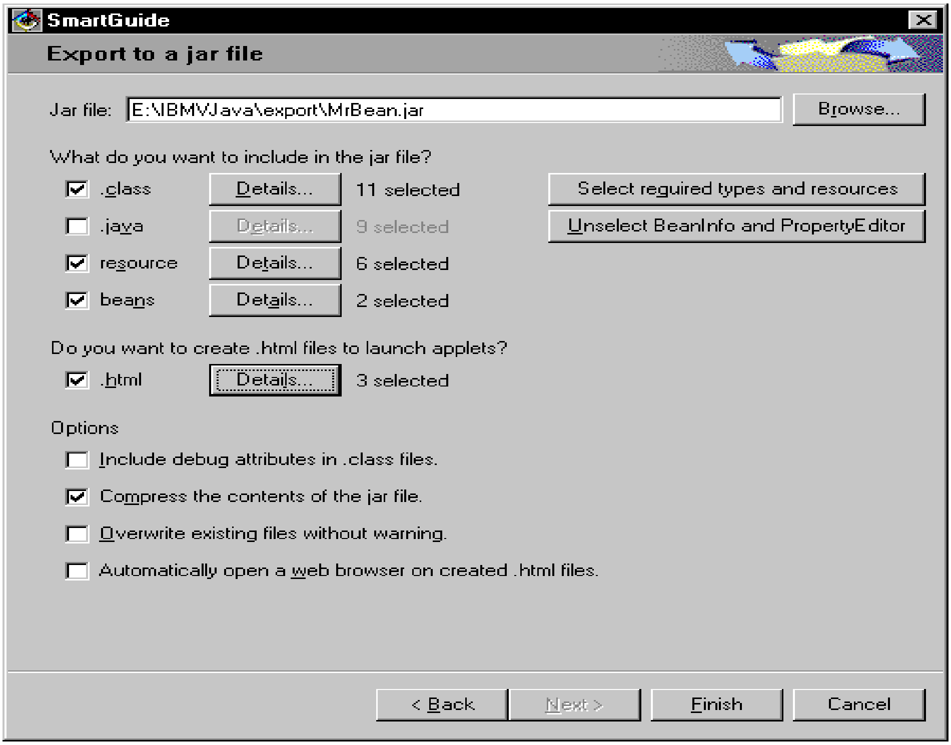
   We recommend including the date in the comment section along with a list of the packages and a description of the project function.

3. Version the project.

4. Export the code from the VisualAge for Java Enterprise environment so that it can be delivered to Mr. Bean.

   There are several export options, but the most common one for this scenario would be to export to a JAR file. Figure 28 shows the VisualAge SmartGuide that quickly guides you through this task.

   Select the classes and resources that you want included in the JAR file. Optionally identify classes that should be Java beans and generate HTML files for the applets. Click on **Finish** to have the JAR file created.



SmartGuide

Export to a jar file

Jar file: E:\IBMVJava\export\MrBean.jar

What do you want to include in the jar file?

☑ .class — 11 selected

□ .java — 9 selected

☑ resource — 6 selected

☑ beans — 2 selected

Do you want to create .html files to launch applets?

☑ .html — 3 selected

Options

□ Include debug attributes in .class files.

☑ Compress the contents of the jar file.

□ Overwrite existing files without warning.

□ Automatically open a web browser on created .html files.

Figure 28. Creating a JAR File of Mr. Bean's Scheduling System

6 Workspace Management

In this chapter we examine various issues concerning the management of an individual user workspace. We describe the Managing page of the Workbench that allows various management operations to be carried out on the standard program elements. We examine how you might organize your projects and packages for maximum flexibility.

We then look at the Management Query facility, which enables you to issue many kinds of search operations over your workspace. Finally we discuss the issue of external resources and suggest how you might manage them to best effect.

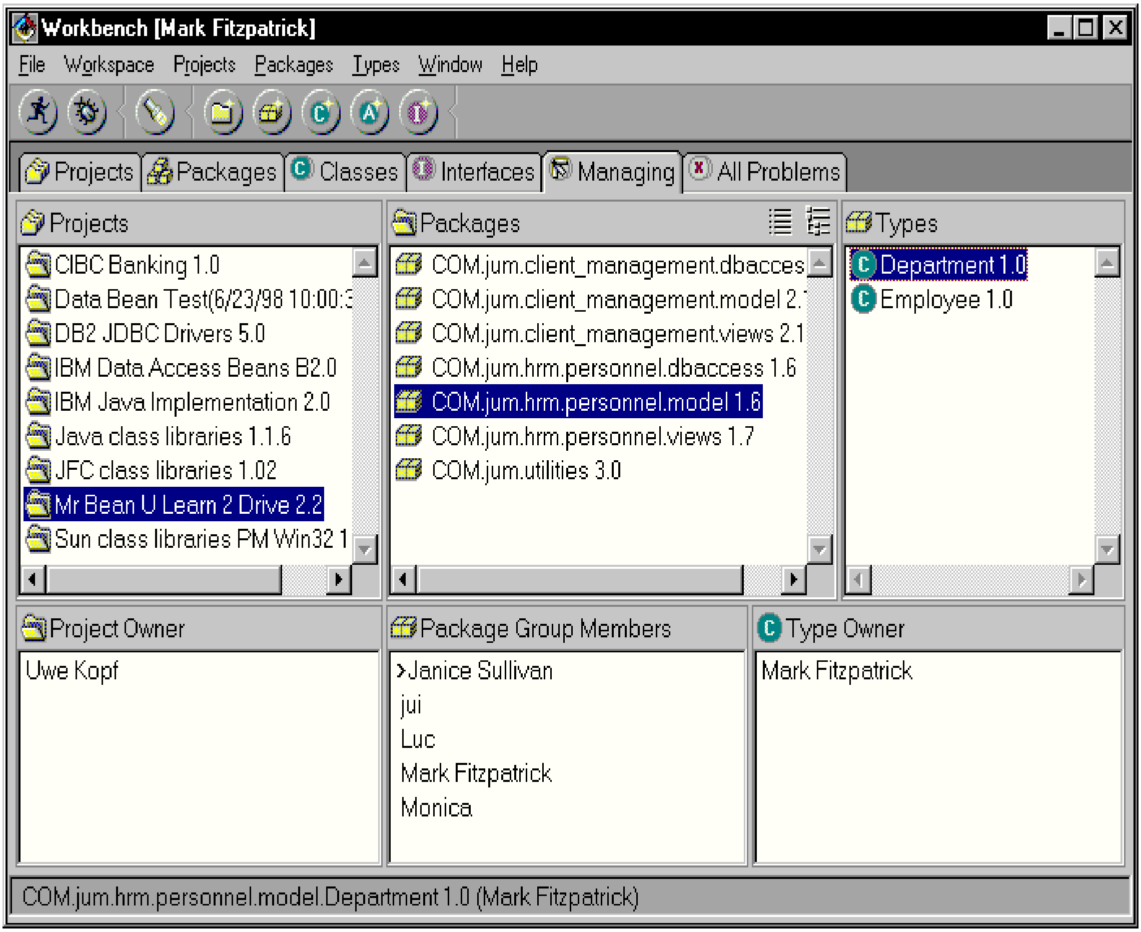
Managing Program Elements

VisualAge for Java Enterprise provides one central place where all user management options can be performed: the Managing page in the Workbench.

The Managing page is a view of all loaded project, package, and type editions and the corresponding ownership information. From the Managing page, an owner of a program element can perform all tasks necessary for version, configuration, ownership, and element management.

All actions can be performed from the menu items or through the pop-up menus in the lower three owner panes or in the upper program element panes. The actions you are allowed to perform on the different program elements depend on the current role you are playing for a certain element. For example, if you are the owner of a package, you can add team members to the package group or transfer ownership to another team member. If you are not the package owner, these options are grayed out.

Figure 29 shows a view of a repository through the Managing page.



Projects

- CIBC Banking 1.0
- Data Bean Test(6/23/98 10:00:3
- DB2 JDBC Drivers 5.0
- IBM Data Access Beans B2.0
- IBM Java Implementation 2.0
- Java class libraries 1.1.6
- JFC class libraries 1.02
- Mr Bean U Learn 2 Drive 2.2
- Sun class libraries PM Win32 1

Packages

- COM.jum.client_management.dbacces
- COM.jum.client_management.model 2.
- COM.jum.client_management.views 2.1
- COM.jum.hrm.personnel.dbaccess 1.6
- COM.jum.hrm.personnel.model 1.6
- COM.jum.hrm.personnel.views 1.7
- COM.jum.utilities 3.0

Types

- Department 1.0
- Employee 1.0

Project Owner

Uwe Kopf

Package Group Members

>Janice Sullivan
jui
Luc
Mark Fitzpatrick
Monica

Type Owner

Mark Fitzpatrick

COM.jum.hrm.personnel.model.Department 1.0 (Mark Fitzpatrick)

Figure 29. Workbench Managing Page

Remember that this page—as with the other pages of the Workbench—allows developers to work with items that are currently loaded in their workspace. The Repository Explorer is used to view other editions of program elements that are not currently loaded in the workspace.

Together with the management query facility described in “Management Query” on page 67, the Managing page serves as a source of all team-related information.

Project Organization

In VisualAge for Java you can use projects in a number of different ways:

- ❑ As a convenient way of grouping packages that have something in common

  VisualAge for Java Enterprise itself delivers several projects with this kind of organization. Consider the *IBM Java Examples* project. It contains a grouping of unrelated packages, each of which contains sample Java code demonstrating various aspects of the product.

- ❑ As a way of grouping together all the elements of a real-world project

  Because loading a project is a single atomic operation, this organization provides a simple mechanism to enable all project members to have a common view of the current state of a project. You are most likely to use this organization for your projects.

- ❑ As a way of organizing packages before making a delivery to a customer

  If a project contains a complete copy of everything that is delivered to a particular customer (or set of customers), it will be easy to replicate a customer environment later on for maintenance purposes.

In practice you can use all three ways to organize your projects, because in VisualAge for Java Enterprise you can specify the same package as part of multiple projects. The only limitation is that only one such project can be loaded into your workspace at any given time.

Package Organization

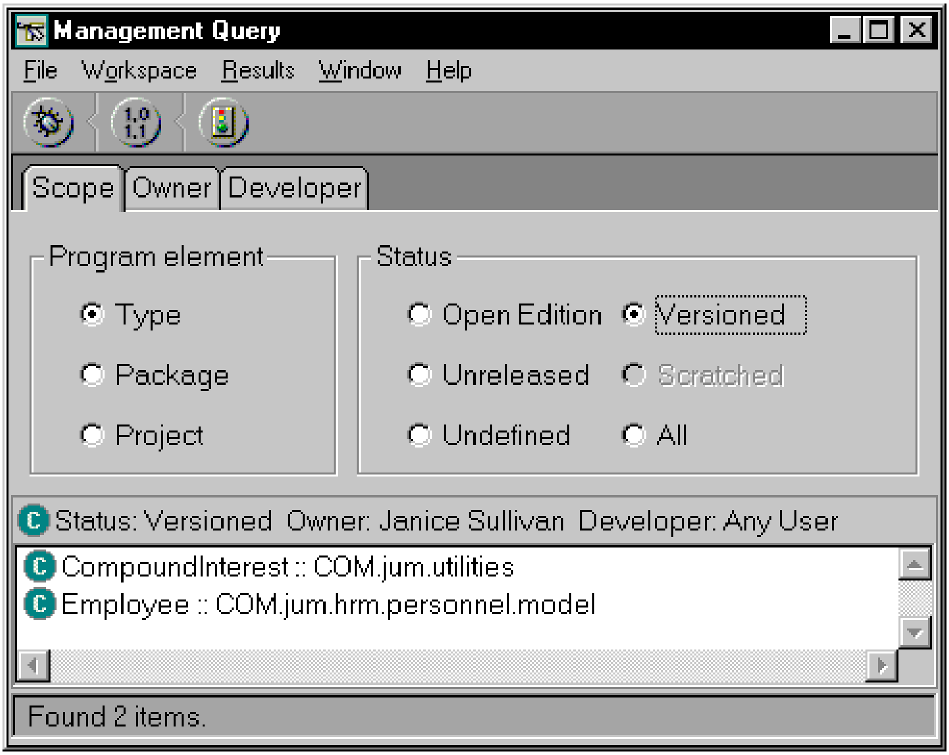
Use packages in VisualAge for Java in exactly the same way as you would in any other Java development environment. Packages provide a way of grouping together a set of classes and interfaces that are related in some way.

In the standard Java Developer’s Kit (JDK), the package naming convention reflects the directory structure in which the classes and interfaces of the package are stored. The standard Java compiler depends on this structure to locate the classes for which it is looking. Because VisualAge for Java is repository-based, rather than file-system-based, the same restrictions do not apply. However, because the code you develop is likely to be eventually exported, VisualAge for Java insists that you name your packages in a conventional way with a dot-separated list of names, each of which represents a valid directory name.

Management Query

VisualAge for Java Enterprise provides a management query facility (Figure 30) for interrogating the status of the program elements in your workspace. This facility enables you to manage your workspace and to regulate its contents. It is an indispensable aid at various stages of the development process when you need to be aware of the status of all of the program elements for which you are responsible.

The management query is invoked through the *Workspace* menu option of any browser. It is organized as a notebook with three pages: Scope, Owner, and Developer.



Program element

● Type
○ Package
○ Project

Status

○ Open Edition ● Versioned
○ Unreleased ○ Scratched
○ Undefined ○ All

Status: Versioned Owner: Janice Sullivan Developer: Any User

CompoundInterest :: COM.jum.utilities
Employee :: COM.jum.hrm.personnel.model

Found 2 items.

Figure 30. Management Query Facility

The management query facility enables you to make queries on types, packages, and projects. For each of these program elements, you can search for elements that are in a particular state, for example, all packages that are open editions.

There are 15 possible query combinations. Each of these queries can be further refined by specifying ownership criteria, and in the case of types, by specifying class or interface developer criteria. On the Scope page of the notebook you can specify the kind of elements you want to view. The Owner and Developer pages provide a list of all users from which you can select a specific user to refine your query.

We examine here a selection of possible queries and discuss a scenario for each that illustrates how that query can be used. In the first example we give details of how to perform the query, but omit the details in subsequent examples because the mechanics are similar and straightforward.

All types owned by Uwe and developed by Mark

During the course of his normal development activities, Mark has to make changes to some classes and interfaces owned by Uwe so that he can progress his own work. When he is satisfied that his changes are complete, he notifies Uwe so that Uwe can appraise the changes and, if appropriate, incorporate them into the official version of the types. Mark uses this query to get a list of all types that have been changed.

To perform this query:

1. Select *Type* and *All* on the Scope page of the query browser.
2. Switch to the Owner page by selecting that tab. Make sure the *Search for any owner* check box is not selected and then select Uwe from the list of users.
3. Switch to the Developer page. Deselect *Search for any developer* and select Mark from the list of users.
4. Click on the *Start Query* button in the tool bar.

Query results are presented in the bottom pane of this browser.

All types owned by Janice and developed by any user

Janice is about to leave the project team, and the classes and interfaces for which she is responsible must be allocated to other team members. You can run this query to get a list of all such types. You can then step through the result list (either as Janice or as the administrator) and change the owner of each type directly from the results pane. As each type has its owner changed, you can explicitly remove it from the list or you can simply rerun the query to get an updated list.

All unreleased types owned by any user and developed by any user

As the owner of a package, you want to monitor progress as the deadline for creating a new package baseline approaches. You can load the latest edition of the package into your workspace and issue this query. This gives you the information about which classes and interfaces need to be released before you can create a new version of your package.

Note that this query—like all others—operates over the entire workspace, and you cannot restrict your search to only the package of interest. However, because fully qualified type names are displayed in the results pane, it should be relatively easy to pick out the types of interest. You can also explicitly remove unwanted types from the list to make it easy to read.

All undefined types owned by any user and developed by any user

Undefined types can exist when a package that is loaded in your workspace has types which are in the repository but are not in your workspace. This situation can arise when there has been a failure while saving your workspace. It can also arise when you load or reload a package in which another user has created a type but has not yet released it.

You can use this query to detect all such types and bring your workspace up-to-date by explicitly replacing the undefined type with another edition of the type.

All scratch editions of packages owned by any user

At regular intervals you can issue this query to show all scratch editions of packages in your workspace. Scratch editions are an indication that some tidying up is required. You may have created a scratch edition of a package you do not own because you made a temporary change to one of the classes in that package. If the change is no longer required, you can remove the scratch edition by replacing it with the current edition. Use the pop-up menu from the Management Query results pane to immediately update the result set to reflect the new status of the package.

Alternatively you may have allowed VisualAge for Java Enterprise to automatically create a scratch edition of a package you own yourself. If the changes you made are of a permanent nature, convert the scratch edition to an open edition as soon as possible.

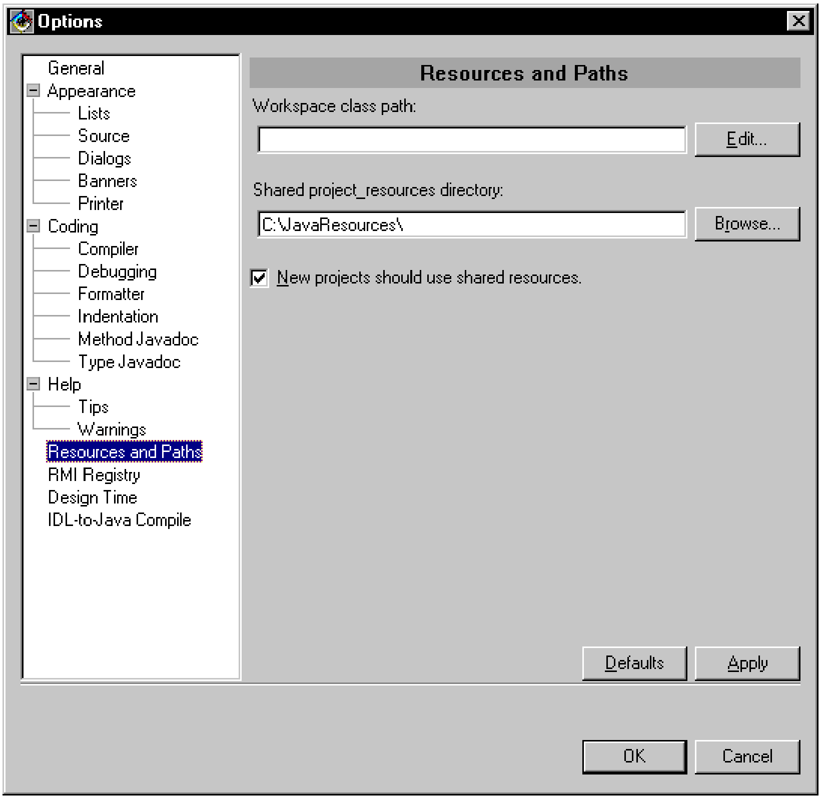
Project Resources

When developing in Java you sometimes need to use external resources that are not part of the language. For example, it is quite common to use images and audio clips in Java applets. VisualAge for Java does not store these external resources in the repository along with your Java source and bytecodes. Instead you must create these resource files explicitly outside the development environment and store them on the standard file system.

VisualAge for Java makes certain assumptions about where resource files are located. For a project called *ProjectX*, it assumes that the resource files will be found in a directory called *IBMVJava\Ide\project_resources\ProjectX* on the client machine. In fact, whenever VisualAge for Java creates a new project, it automatically creates a directory with the same name. VisualAge for Java uses this resource directory in a number of ways:

- ❑ When running an applet from within VisualAge for Java, the code base for the applet is specified as the name of the resource directory. The URL of this directory will be returned by the getCodeBase method of the java.applet.Applet class.
- ❑ When running an applet or an application from within VisualAge for Java, the default CLASSPATH contains the resource directory.
- ❑ When exporting a project to a JAR file, all files in the resource directory for that project can be included in the JAR file, and the JAR file can be compressed.
- ❑ Skeleton HTML files for all applets are optionally generated.

An alternative to the default resource directory is to explicitly tell VisualAge for Java in the workspace options (*Window > Options*) that you want to use shared resources and indicate the path to the location where those shared resources can be found (Figure 31).



Workspace class path:

Shared project_resources directory: C:\JavaResources\

☑ New projects should use shared resources.

Figure 31. Specifying a Shared Resource Directory

This path should identify a shared file location. You must manually create a directory under this path for each project whose resources are to be shared. Even if this option is set, VisualAge for Java still automatically creates the default resource directory. This shared resource location is used by VisualAge for Java when exporting a project in JAR format. In fact it includes all resource files from both the shared and the default resource directories. The shared resource directory is *not* used to establish the code base when running an applet from within VisualAge for Java.

Resource files must be managed carefully to avoid problems when VisualAge for Java needs to locate them. Issues that must be considered include:

- If many developers are working on a single project and each developer manages resources independently, problems will occur when you attempt to consolidate the work of separate developers. For example, system testers will be unable to locate resources when testing a a package that has many class owners. More fundamentally, a class owner may be unable to test a class if it uses another class that has associated resource files and is owned by another developer.

❑ When carrying out certain kinds of export operations, all resource files for a project must be accessible to VisualAge for Java or they will not be included in the export.

❑ By default, resource files are located by the project in which a package exists. If you move the package to another project, VisualAge for Java will not be able to locate the resource files.

In addition to the requirement that VisualAge for Java be able to locate resource files, it is also important that your applets and applications can find their resource files in a consistent way.

Managing Resource Files

Here we offer some suggestions for managing your resource files. Many options are available to you, and you should consider the unique circumstances of your own environment before deciding which solution will work best for you. The first scenario makes use of VisualAge for Java's shared resources facility. The second scenario manages resources by package.

Using Shared Resources

In this scenario, you want to manage your resources using a shared resources directory.

❑ Create a directory on a shared file system to act as the shared resource path.

❑ Set the VisualAge for Java option to use shared resources as detailed above.

❑ In the shared resource directory, create a subdirectory for each project that contains resource files to be managed.

❑ It is the project owner's responsibility to maintain the project resource directories. This is normally done in conjunction with the owner of the class that uses the resources. It is also the class owner's responsibility to copy the required resources into the local project resource directory as created by VisualAge for Java. Some operating systems (UNIX, for example) allow you to create a soft link between the master copy and your local copy. This is the preferred mechanism, but a hard copy is acceptable.

❑ All developers and testers are responsible for copying the required resource files into their personal project resource directory.

❑ When it is time to deliver a project through an export operation, the required resource files are already in place.

Managing Resources by Package

In this scenario, you want to manage your resources by package as it likely that your packages will be moved among projects for organizational reasons.

- ❑ Establish a directory on a shared file system to store the master copy of all resource files.
- ❑ In the directory create a subdirectory for each *package* that contains resource files to be managed. Use the full name of the package as your directory name to keep the hierarchy shallow and facilitate retrieval. If you have a large number of packages, you can group them in arbitrary subdirectories. By organizing your resource files by package instead of by project, you avoid problems associated with moving packages among projects.
- ❑ It is the package owner's responsibility to maintain the package resource directories. This is normally done in conjunction with the owner of the class that uses the resources. It is also the class owner's responsibility to copy the required resources into the project resource directory as created by VisualAge for Java.
- ❑ All developers and testers are responsible for copying the required resource files into their personal project resource directory.
- ❑ It is the responsibility of the project owner to copy all resource files of all packages in the project into the project resource directory before exporting the project as a JAR file.

In both scenarios class developers should access resource files in their code relative to a common root directory. For applets, this should be the applet code base. This organization fits in with the code base established by VisualAge for Java for testing applets. For applications, there is much greater freedom, and you will probably pass a parameter to your program indicating where resource files are kept.

7 Repository Management

In this chapter we discuss the role of the administrator and the different tasks associated with that role. We also discuss which steps have to be performed to clean up the repository, and we provide different scenarios for repository backup.

The Role of the Administrator

When you install VisualAge for Java Enterprise, one user is predefined, the administrator. This user plays an important team role and cannot be deleted from the VisualAge for Java Enterprise repository. The first user of a new installation is the administrator and has all of the rights necessary to set up the development environment on the server.

The administrator's role is primarily one of managing the repository (or repositories) and includes five major tasks:

❑ Set up the VisualAge for Java Enterprise team environment on the server (see "Installation of the Team Environment" on page 6).

❑ Grant users access to the repository. This task consists of two parts:

- Creating users in the repository
- Defining user IDs and passwords for EMSRV, if required (see "Accessing the VisualAge for Java Enterprise Server" on page 9)

❑ Maintain existing users

❑ Clean up the repository when necessary

❑ Back up the repository regularly

We cover the user-related tasks in Chapter 2, "Installation and Setup" on page 5. In this chapter we focus on the repository-related tasks.

The administrator must be familiar with the VisualAge for Java Enterprise environment and the server machine environment. The administrator has to know about the project organization: which developers are part of the project team with what responsibilities. However, the administrator has no need to know the details of the projects that are developed with VisualAge for Java Enterprise.

As an administrator you may also be a member of the development team itself. In this case you are playing two or more roles, depending on the size of the development team and/or the complexity of the development project. For example, the administrator may also be a package owner and a class developer at the same time.

You must clearly distinguish between the tasks you perform as an administrator and those you perform as a package owner or class developer. You will have two users defined for you: the administrator user, and the user under which you do development. You must switch between these two users according to the task you are performing. The administrator user should never author any code on any project!

How do I become an administrator?

❑ When you are the user performing the initial installation of VisualAge for Java Enterprise, you become the administrator by default.

❑ By changing the workspace owner to administrator. If password checking is enabled, you have to provide the valid password for the administrator.

What can I do as an administrator?

❑ Add new users to the repository

❑ Change or delete existing users

❑ Compact the repository

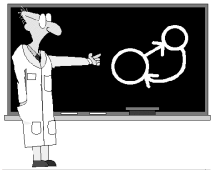
❑ Change ownership of an existing project

❑ Purge open editions and versions of a project

❑ Purge open editions and versions of a package

Besides the actions listed above, you can perform other tasks that do not depend on the administrator role. For example, you can become a class developer and then do all the things a class developer can do. But, as mentioned before, do not perform developer tasks as the administrator user.

Important Info	Implicit creation of users
	There is one exception to the rule that only the administrator is allowed to create users. When you import a package, all users in the package group associated with the package are implicitly created if they do not exist.

Tip	Restoring deleted users
	When you accidently delete a user that owns program elements, the user still exists in the repository and is identified by a unique name. The connections from all program elements the user owned to the deleted user still hold. You can restore the user by creating a new user, using the same unique name of the user you deleted. Deleted users are removed from the repository after the repository is compacted.

Repository Cleanup

Over time the repository grows as a result of the ongoing work of the developers. Each developer creates new editions of different program elements. In fact each save of a method creates a new open edition of the method that is stored in the repository twice: as source code and as compiled code.

Once the repository has been cleaned up, it contains no obsolete or unwanted editions of program elements.

There are mainly two reasons for cleaning up a repository:

- The repository grows to a size that leads to a decrease in performance. The upper limit size depends on the platform and the file system where EMSRV is running. For Windows NT, the maximum size is 2 GB for a partition that uses a file allocation table (FAT) and 16 GB for a partition that uses the Windows NT file system (NTFS). A repository size of 250 MB to 300 MB is quite common.
- If developers tend to version classes or packages too often by versioning packages that are not baselines, it is useful to clean up the repository. Cleanup provides better control over elements in the repository and results in a shorter list of program elements.

Cleaning up the repository is basically a team effort. Even though only the administrator has authority to begin the compact process, the developers and the administrator must work closely together to avoid problems that might result in time-consuming activities to recover work that has been accidently lost.

There are four steps to achieving a reorganized repository that is free from unwanted and obsolete program elements. These steps must be performed in order, and different team members may be responsible for them as shown in Figure 32.

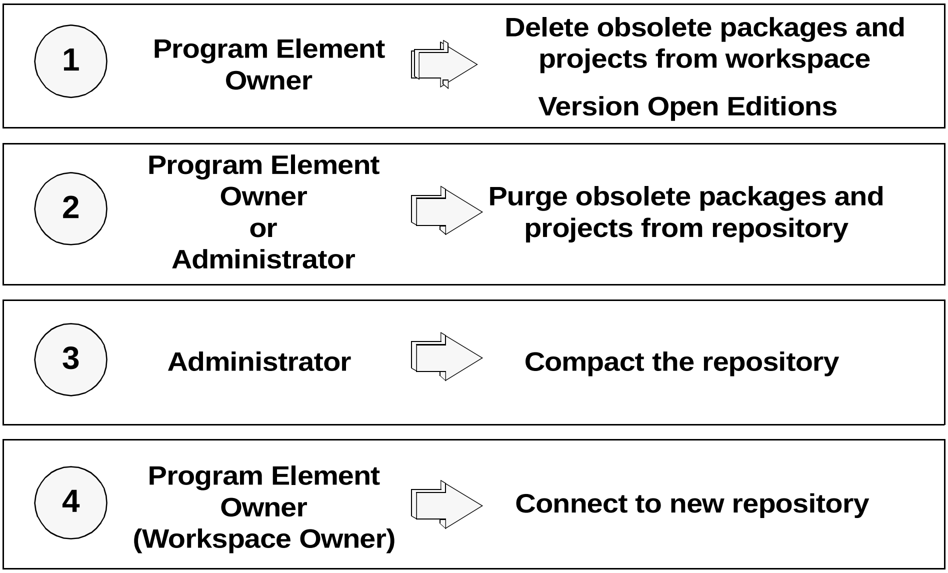

1
Program Element Owner
Delete obsolete packages and projects from workspace
Version Open Editions
2
Program Element Owner or Administrator
Purge obsolete packages and projects from repository
3
Administrator
Compact the repository
4
Program Element Owner (Workspace Owner)
Connect to new repository

Figure 32. Four Steps to Cleaning up the Repository

1. Delete obsolete packages and projects from the workspace and version open editions

   Each developer who owns a program element that has become obsolete performs this step. Deleting a program element unloads it from the workspace. (You do this not only for purging purposes but also to keep your workspace in good shape.) Open editions should be versioned because compaction of the repository copies only versions to the new repository. This first step is performed at any time during the development cycle, not only just before purging of the program element.

2. Purge obsolete packages and projects from the repository

   This step is carried out by the package or project owner or by the administrator, depending on the size and organization of the development team. It is important to ensure that no other users require the program elements you are purging. Although purged elements can be restored, regard this step as an action you do only to obsolete elements.

3. Compact the repository

   The administrator performs this step. Compacting must be scheduled in advance and cannot be done during development time. Therefore compacting should be thought of as a batch job. The administrator must ensure that no other user is working with the repository that is about to be compacted.

4. Connect to new repository

   This step is typically done implicitly by each developer when starting VisualAge for Java Enterprise after the compaction because the compacted repository is usually renamed to the name of the old repository.

Attention!	Keep your old repository
	Do not delete your old repository after you have compacted it and all developers have switched over to the new repository. Murphy's law tells us that a developer will realize something was left out as soon as the old repository is deleted! Store the old repository in a safe place for a predetermined and agreed on time frame, so that team members have a grace period for recovering old items they still need.

Deleting and Versioning Program Elements

Deleting program elements from the workspace is a routine activity during development. Program elements are deleted to minimize the size of the workspace (remember that the workspace is loaded into memory) or because they are obsolete. Deleted program elements are still in the repository. All developers must delete program elements before they can be purged.

Open editions of program elements should be versioned because compacting of the repository copies only versions and not open editions.

Purging Program Elements

Purging program elements *logically* removes them from the repository. You purge only those elements that you and your team no longer need. After a program element is purged, you do not see its name in the Repository Explorer.

Because purging is a logical delete, it does not free up the disk space occupied by the program element. However, you can restore the program element. If you accidentally purge a program element, immediately attempt to restore it.

For large teams we suggest that you not rely on recovery of purged program elements. Look at the purge process as a process that removes your program element from your environment, because you are not sure when the administrator will compact the repository.

If program elements are purged with any intent other than permanently removing them from the repository, the purge function is being used incorrectly.

Which program elements can be purged?

- ❑ Single open editions and single versions of packages
- ❑ Single open editions and single versions of projects
- ❑ A complete package
- ❑ A complete project

When you purge a complete project or package, you are purging all existing editions, not just a single edition of that element. When you purge the last existing edition of a package or project, the complete package or project is purged.

Who can purge program elements?

- ❑ The owner of the open edition or version of a package or project
- ❑ The administrator

Program elements must be deleted from your workspace before you can purge them. Purging program elements can be done only from the *Repository Explorer* window. You have to choose the appropriate notebook page for purging projects or packages. Even if you see all package editions when you are on the Projects page, you have to switch to the Packages page to delete a package edition.

Figure 33 shows the purging operation with two pop-up menus. The pop-up menu that is displayed depends on whether you click the right mouse button on the package name (to purge all editions) or the edition name (to purge a single edition).

Notes on purging program elements:

- ❑ Make sure other users do not need your program element.
- ❑ Make sure no other user has your program element loaded in the workspace. You have to ask the other developers, because VisualAge for Java Enterprise provides no query for that.
- ❑ You do not get a warning if you are deleting a package or project that is actually loaded in another user’s workspace! When the other user starts to work with the purged element, he or she is notified that the element is missing, as shown in Figure 36 on page 85.
- ❑ You get an error if you are deleting a package or project that is actually loaded in your workspace.

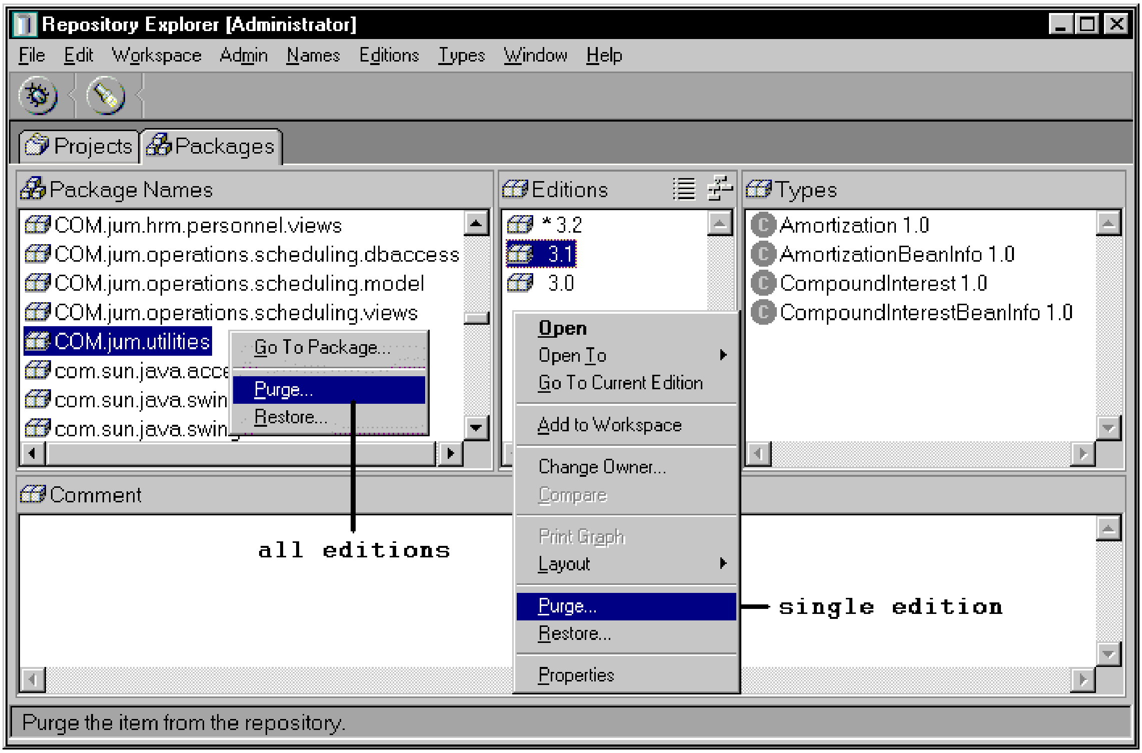


Figure 33. Purging Program Elements from the Repository

Purging Packages

Packages are structures that contain executable code. If you purge a package or a certain edition of a package, all editions of the contained program elements are also purged. Because packages are the smallest unit for purging pieces of code, there is no other way to remove certain class or method editions from the repository. After compacting, the purged package and the contained program elements are no longer available in the new repository.

Purging packages is the only way to reduce the size of the repository considerably. Although purging packages itself will not free up any disk space, it is the necessary prerequisite for the compacting step (see Figure 34).

Before you purge a package or a package edition, ensure that no other users are using it.

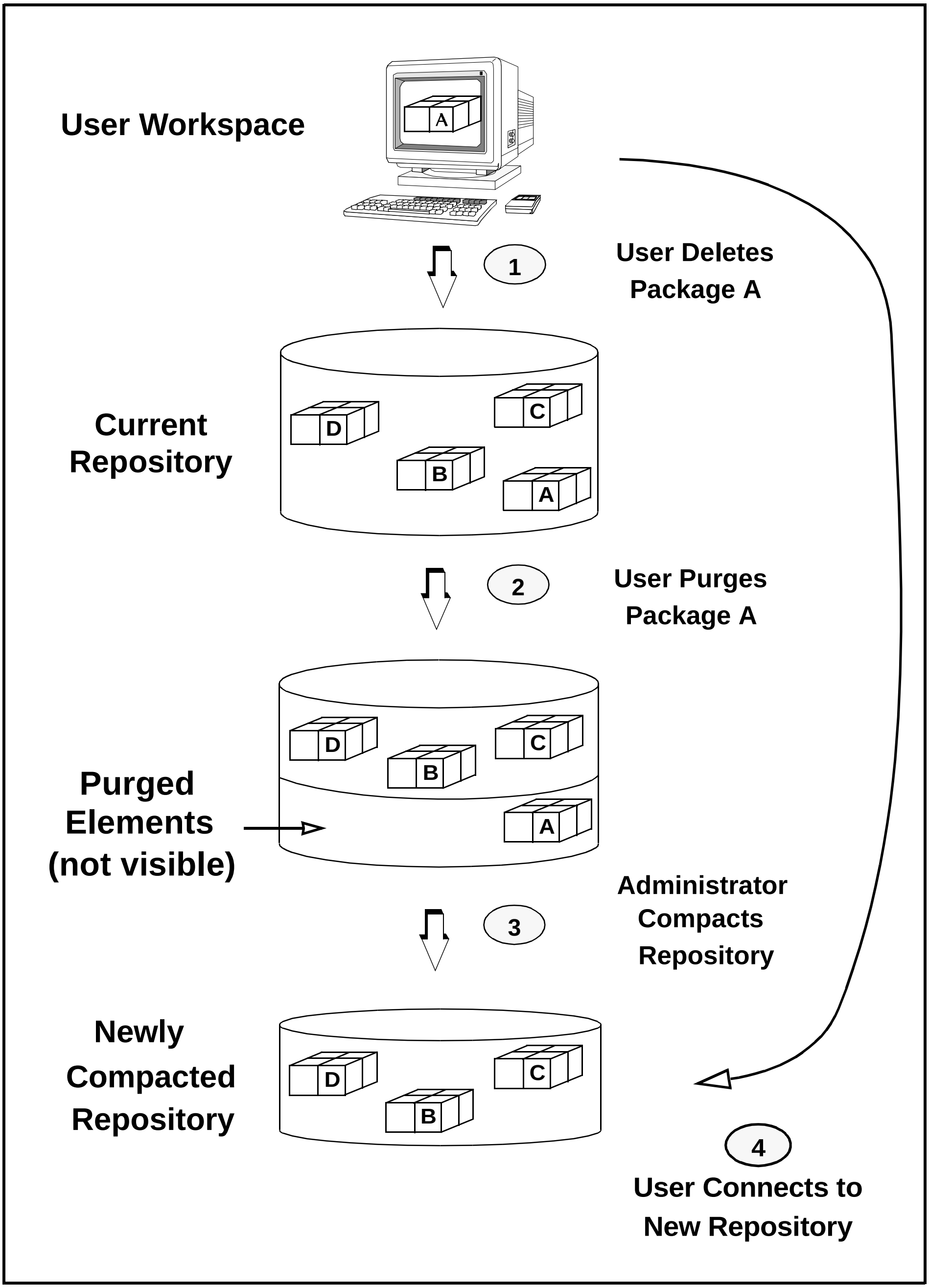

User Workspace
A
1
User Deletes Package A
Current Repository
D
C
B
A
2
User Purges Package A
Purged Elements (not visible)
D
C
B
A
3
Administrator Compacts Repository
Newly Compacted Repository
D
C
B
4
User Connects to New Repository

Figure 34. Repository Cleanup When Purging Packages

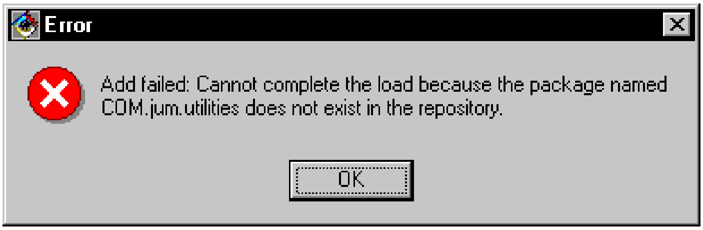
Consider the following problem and solution:

Problem: Uwe is owner of the *Mr. Bean U-Learn 2-Drive* project, and Mark is owner of the COM.jum.utilities Version 3.1 package. Uwe has added several packages to his project, one of which is the COM.jum.utilities Version 3.1 owned by Mark. Mark has decided to purge this version of his package, for some reason, but has not notified the team of his purge.

Now consider two cases:

- ❑ Uwe has loaded the *Mr. Bean U-Learn 2-Drive* project in his workspace. He will not run into trouble as long as he does not touch the purged package. He will be able to change all other packages and even version and delete his project.
- ❑ Uwe has not loaded the *Mr. Bean U-Learn 2-Drive* project in his workspace. He therefore will not run into trouble at this point.

Now Uwe—or any other user—wants to add the *Mr. Bean U-Learn 2-Drive* project into the workspace. Because the COM.jum.utilities Version 3.1 package is part of the project edition but does not exist in the repository (which in fact means it is not visible in the Repository Explorer), VisualAge for Java Enterprise does not load the project. Because adding a project in this case is an atomic operation, even the other existing packages are not added to the workspace. Instead Uwe gets the message shown in Figure 35.



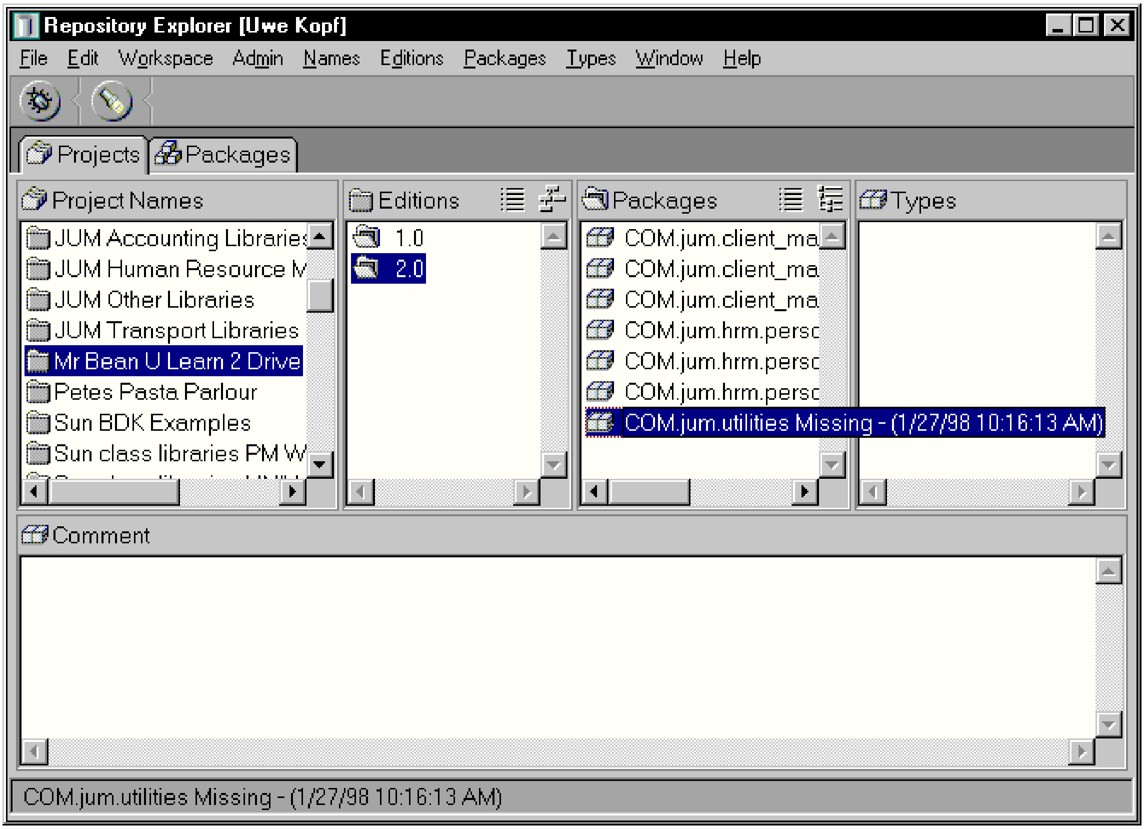
Add failed: Cannot complete the load because the package named COM.jum.utilities does not exist in the repository.

Figure 35. Error Message When Load Failed

Solution: Be aware that the phrase *does not exist in the repository* can have two different meanings:

- ❑ The element is not in the repository at all. Therefore, because Mark purged the package, the repository must have been compacted. Uwe will have to look for a backup of the old repository and then follow the procedure described in “Troubleshooting If Restoring Does Not Work” on page 88.
- ❑ The element still resides in the repository but is purged and therefore not visible in the Repository Explorer. In this case, Uwe can restore the package and then load the *Mr. Bean U-Learn 2-Drive* project.

Remember, you are not notified when a package that is currently loaded in your workspace has been purged by another user until you perform a repository operation on it. If you open the Repository Explorer on the project that contains the package, you get an information message (Figure 36).



COM.jum.utilities Missing - (1/27/98 10:16:13 AM)

Figure 36. Information Message When a Package Is Missing

All you have to do is restore the package as described in “Restoring Program Elements” on page 88, and you can work with it again.

Attention!	Do not try to add *missing* elements
	If you find a *missing* package when browsing your project in the Repository Explorer, you still have the pop-up option *Add to Workspace*. We strongly recommend not clicking on the *Add to Workspace* option. It causes unpredictable results and your workspace may become corrupted!

Purging Projects

Purging projects is different from purging packages. Projects are structures that organize code. If you purge a project or a certain edition of a project, you logically delete the information about how packages are organized.

From a repository perspective, a project does not contain packages, but packages are part of one or more projects. Each project is a configuration of packages. When you purge a project therefore, you delete the information that certain packages are part of the project configuration. In other words, you are tidying up the repository rather than removing code elements.

Purging a project does not purge any of the packages that are part of the project.

You may want to purge a project or a project edition for two reasons:

❑ You do not need any of the packages that are part of the project and therefore they do not have to be copied to the new compacted repository. In this case you must:

   1. Purge all packages that are part of the project or project edition

   2. Purge the project or project edition

❑ You want to unlink all packages from the project because you no longer need the project. In this case you only have to purge the project or project edition.

Therefore if the repository contains some purged projects but no purged packages, compacting the repository will not result in a considerable reduction of the required disk space for the new repository because only configuration information, not code, is removed from the new repository (see Figure 37).

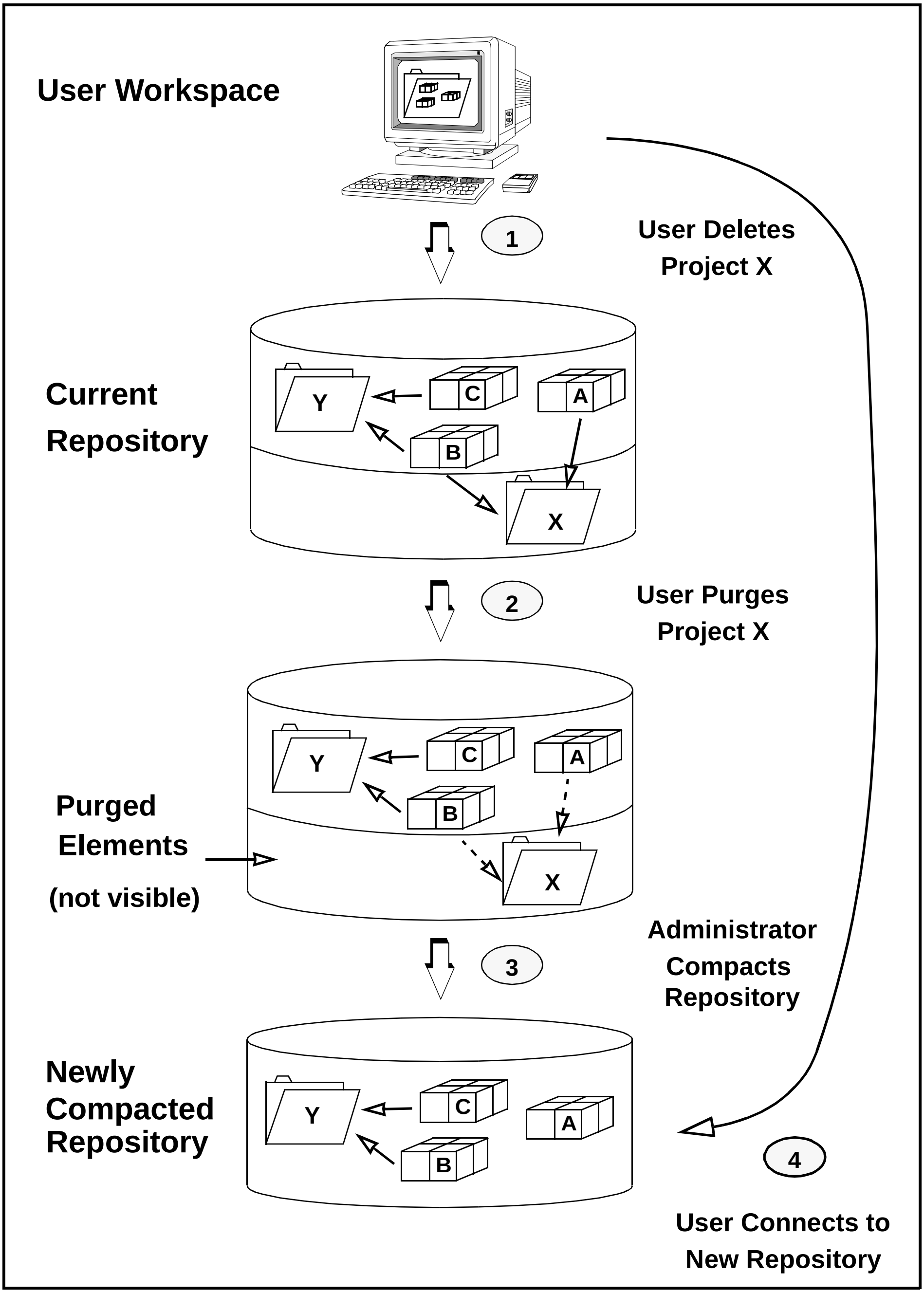


Figure 37. Repository Cleanup When Purging Projects

Restoring Program Elements

You can restore program elements that have been purged from the repository before it is compacted. After you restore them, they are visible in the Repository Explorer and can be added to the workspace.

Which program elements can be restored?

❑ All previously purged program elements as long the repository has not been compacted and you are connected to the new repository.

Who can restore program elements?

❑ Every user can restore any purged program element.

When you restore a program element that you do not own, you will not get ownership. Restoring only makes a program element visible again in the Repository Explorer and removes the indicator that it will not be copied in the new repository when the administrator compacts the current repository.

You can only restore from the Repository Explorer window. Remember that you have to choose the appropriate notebook page for restoring either projects or packages.

Troubleshooting If Restoring Does Not Work

Purge only those program elements that are obsolete. Do not use the purging mechanism as a facility to hide projects or packages in the repository temporarily. If you accidentally purge a project or a package, try to restore it immediately. If you do not act immediately, there is no guarantee that you can restore your work later on through a series of simple mouse clicks.

Consider the following problem and solution:

<u>Problem:</u> You are a developer on a large team. You have purged a package accidentally and you have not restored it immediately afterward. At the end of the day, the administrator compacts the repository. When you come back the next day and start VisualAge for Java Enterprise, you will get connected to the new repository. You then want to restore the package that you accidentally deleted, but it is no longer in the repository.

<u>Solution:</u> Ask your administrator to give you access to the “old” repository. Connect to the old repository either by changing your ide.ini file or by using the *Change Repository* option from the Repository Explorer. Then restore the purged package. You then can export the package into another repository, connect to the new repository, and import the package from the other repository.

Compacting the Repository

Compacting a repository copies all versioned projects and packages (including all versioned classes) to a new repository. Compacting therefore has two aspects:

- Tidying up the repository
- Reducing the size of the repository

Only the administrator can compact the repository.

After the repository is compacted, you have two repositories: the unchanged original repository, and the new, compacted repository that does not contain any open editions and purged elements. In comparison to exporting versioned projects or packages, compacting copies all versions of a program element to the new repository if they have not been purged. Exporting copies only the released version of contained program elements and acts only on elements currently loaded in the workspace. Figure 38 gives an overview of the compacting operation.

Important Info	Compacting may remove class versions
☞	Versioned classes are removed from the repository during compacting if they are contained only in an open edition of a package! This is due to the fact that classes have to be contained in a package. If the package is an open edition, it is removed with all of the contained classes, whether they are versioned (and released) or not. So versioning a class is not enough to keep it in the repository when the repository is compacted.

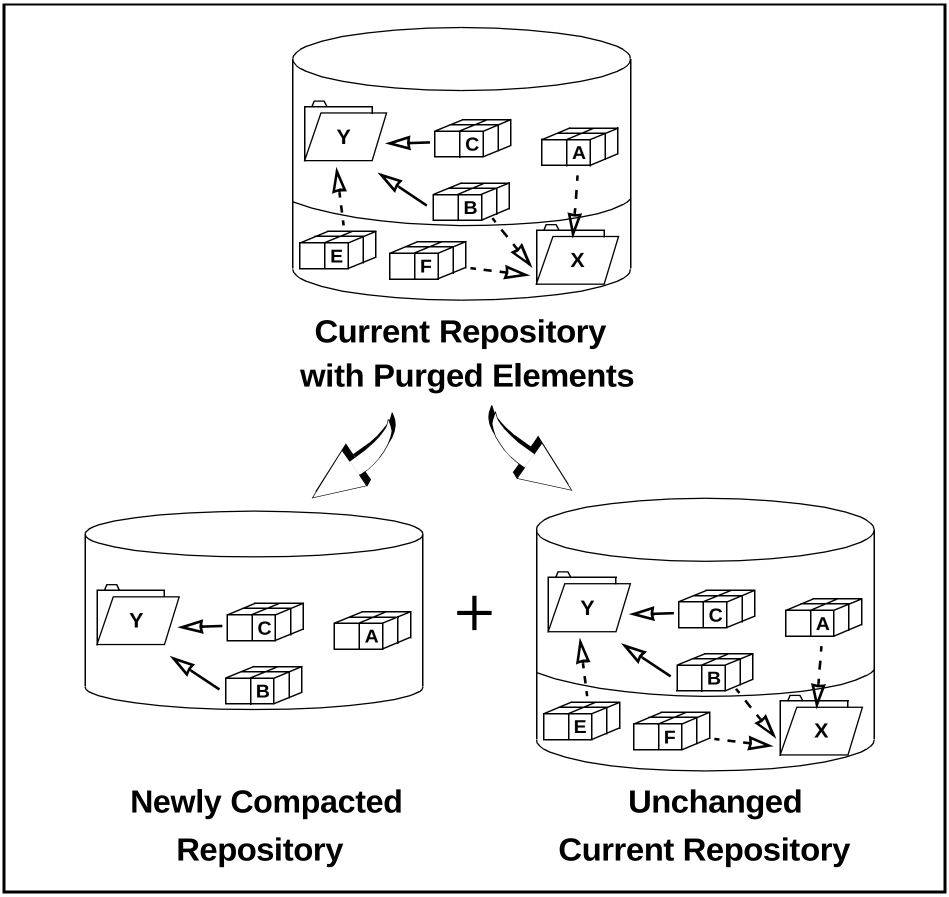


Figure 38. Compacting a Repository

When to compact a repository depends on the actual size of the repository and the resulting decrease in performance on the server machine. Compacting does not have to be performed at predefined times; rather it is driven by the development cycle of purging packages.

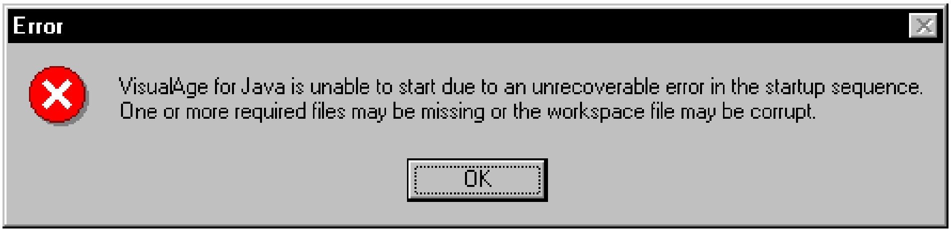
To avoid problems, plan ahead and discuss the procedure with the team developers. Besides the actual compacting operation, other considerations apply. For example, it is absolutely necessary to ensure that no user other than the administrator has access to the repository during compacting. If another user works on the repository during compaction, the repository may become corrupted!

Table 1 lists the 10 steps to compact a repository in a professional and secure way.

Table 1. Administrator's Checklist for Compacting a Repository

Step	Description
1	Communicate to the team that the repository will be compacted and each package and project owner should purge all obsolete projects and packages. Ask the owners and developers to shut down when they are done purging.
2	Check that no team member is connected to the repository. Use the EMADMIN LIST command (see Appendix C, "EMADMIN" on page 133). No connection should be reported.
3	Stop EMSRV (see Appendix B, "EMSRV" on page 127).
4	Make a backup copy of the current repository (see "Repository Backup" on page 92).
5	Restart EMSRV with the maximum user option set to 1 to ensure that no other user can connect to the repository during compaction.
6	Restart VisualAge for Java.
7	From the Repository Explorer window, start compacting the repository from the *Admin* menu. Do not use the default name for the new repository because you cannot compact a repository into itself. The Log window displays the progress during the operation.
8	After you have terminated the compaction, shut down VisualAge for Java and stop EMSRV.
9	Rename the original repository to an archive name. Rename the new repository to the original name. Delete the backup copy from step 4.
10	Restart EMSRV with standard options.

If another user tries to connect to the repository while the administrator holds the only valid connection to it, VisualAge for Java displays the error message shown in Figure 39.



VisualAge for Java is unable to start due to an unrecoverable error in the startup sequence. One or more required files may be missing or the workspace file may be corrupt.

Figure 39. Error Message When Connecting to a Repository during Compaction

Connect to New Repository

Connecting to the new repository does not require any developer action if the compacted repository was renamed to the name of the old repository.

When a developer starts VisualAge for Java Enterprise, a connection to the compacted repository is performed automatically.

Repository Backup

The repository is the only place where all work of all developers on the team is saved. It is an extremely powerful concept for sharing work and for recovery. If one of the team member’s workspace becomes corrupted, it can be easily recovered from the repository. However, if for some reason the repository itself becomes corrupted, all work done by all users on the team is lost! Therefore the administrator has to take care that the work in the repository is backed up regularly. In addition to backing up the repository, the administrator has to back up all shared resource files at the same time (see “Project Resources” on page 70).

Team size, amount of development activity, and the critical nature of your projects are some of the factors to consider when deciding on a backup schedule for your team. Certainly an active VisualAge for Java Enterprise development team should be backing up the repository on a daily basis. You will also have to decide how long you will save the old repositories.

There are different ways of making backup copies on the server. We provide two principle scenarios, one using the EMADMIN copy command, and one using native operating system commands. These scenarios apply for a Windows NT server machine; they may be slightly different on other server platforms.

Scenario 1: Using EMADMIN Copy Command

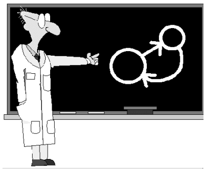
EMADMIN provides a copy command to copy a repository (see Appendix C, “EMADMIN” on page 133). EMADMIN checks that the file you are about to copy is a VisualAge for Java repository. The EMADMIN copy command also locks the repository while it is copied. A user connected to the repository has to wait until the copy has finished before he or she can continue working.

Table 2 shows the steps for the administrator to follow to safely back up the repository with the EMADMIN copy command.

Table 2. Repository Backup Using EMADMIN

Step	Description
1	Communicate to the team that the repository will be backed up. This is a warning because VisualAge for Java does not allow any updates during the backup operation.
2	At a command prompt type the EMADMIN command to copy the repository. No other user will be able to update the repository during the copy process.

Note: If another user tries to update the repository during a backup process, VisualAge for Java displays an hour glass and waits for the backup to finish.

Tip	Locking out other users during backup
	You can stop EMSERV and restart it with the maximum user option set to 2. This prevents any other users from connecting to the repository while the backup is running. This precaution is optional because the EMADMIN copy command locks the repository for updates.

Scenario 2: Using Operating System Commands

One way of ensuring that no user can access the repository while the administrator copies it is to shut down EMSRV. All VisualAge for Java Enterprise client environments should be set up to connect to the repository through EMRSV (which we strongly suggest), and the server must be set up such that no direct client access to the repository is possible.

Table 3 shows the steps to back up the repository by using operating system commands.

Table 3. Repository Backup Using Operating System Commands

Step	Description
1	Communicate to the team that the repository will be backed up and each team member should shut down VisualAge for Java. Close your own workspace if you have VisualAge for Java running.
2	Check that no team member is connected to the repository. Use the EMADMIN LIST command (see Appendix C, “EMADMIN” on page 133). No connection should be reported.
3	Stop EMSRV (see Appendix B, “EMSRV” on page 127).
4	Make a backup copy of the repository, using the copy command of the operating system.
5	Restart EMSRV with standard options.

8 Workspace and Repository Configuration Options

In this chapter we discuss the different options for managing your client developer workspace file and the connections that you can make to different repositories.

We also discuss an approach that uses multiple workspaces for connecting to multiple repositories.

The Basics of Connecting to a Repository

First, let us recall that a VisualAge for Java Enterprise client cannot run without a live connection to a repository. A repository file is always required; VisualAge for Java fails to start if a valid repository file is not found at startup.

When VisualAge for Java is started, the ide.exe file reads the ide.ini file that specifies the location of the repository with which the workspace should attempt to connect. For each program element in the workspace, VisualAge for Java maintains pointers to the permanent representation of that element in the repository. Thus VisualAge for Java can quickly and easily access the source of each element as you use it in the course of normal development activities.

If the workspace has not been previously connected to the repository with which it is currently attempting to communicate, the cache of pointer information must be reestablished. The term *recache* is used to describe this activity. This situation occurs during the initial startup after installation and after switching to a new repository with the *Change Repository* option.

Whenever your workspace contains packages that are not in the repository to which you are connecting, VisualAge for Java cannot reestablish pointers to the classes and interfaces in those packages. This will generate many log messages but will not cause any problems until you attempt to access those classes and interfaces from the Workbench. You will then be advised, quite rightly, that the source for the program element could not be found. You can avoid this situation by making sure that your workspace contains only packages that also exist in the repository you want to connect to, *before* you make the connection. However, this solution may be unrealistic, and it is best to understand the nature of the log messages. If you are aware of the situation and understand the implications, you may be quite comfortable proceeding to connect to a new repository, knowing that the new work you intend to do has no bearing on those elements you know do not exist in the target repository.

The most typical installation and working environment for VisualAge for Java Enterprise is a configuration that has the repository installed on a file server owned and managed at a department level within a development organization. Multiple developer workstations will be installed with the client portion of VisualAge for Java Enterprise (this is of course all the running code!). Much of the day-to-day development work will take place in this configuration in the office environment. There will, however, be occasions when work needs to be done offsite—most likely involving a developer working at home.

It is not feasible to expect that a developer will copy the central department repository to take home. How then can VisualAge for Java Enterprise accommodate this requirement?

There are three main ways:

- ❑ Work is conducted remotely while the developer is connected to the main repository
- ❑ Work is conducted on a single machine (a laptop, for example) that can be connected to the main repository or a local repository
- ❑ Work is conducted on two separate client machines and two separate repositories

Remote Connection to a Repository

Remote connection to a repository is the simplest case and is essentially an extension of the standard way of working with VisualAge for Java Enterprise Provided a fast enough TCP/IP connection is available, a developer can connect to a remote repository and work as though connected through a LAN. All that is required is the IP address of the team server and the name of the repository.

In our experience, a minimum line speed of 28,800 bps is required to connect to a remote repository. Although reconnecting your workspace to the new repository will take some time, once you are connected, access to the repository will be faster. In practice, the nature of the work being carried out will determine the minimum response time that is acceptable. A slower connection might be acceptable for low volume maintenance work, while a faster connection is required for heavy development activities.

Connecting to Two Repositories

A developer uses a single machine to connect to two separate repositories. As an example, you might use a laptop computer in a docking station in the office and connect to the main repository and work at home using the laptop and a local repository (see Figure 40).

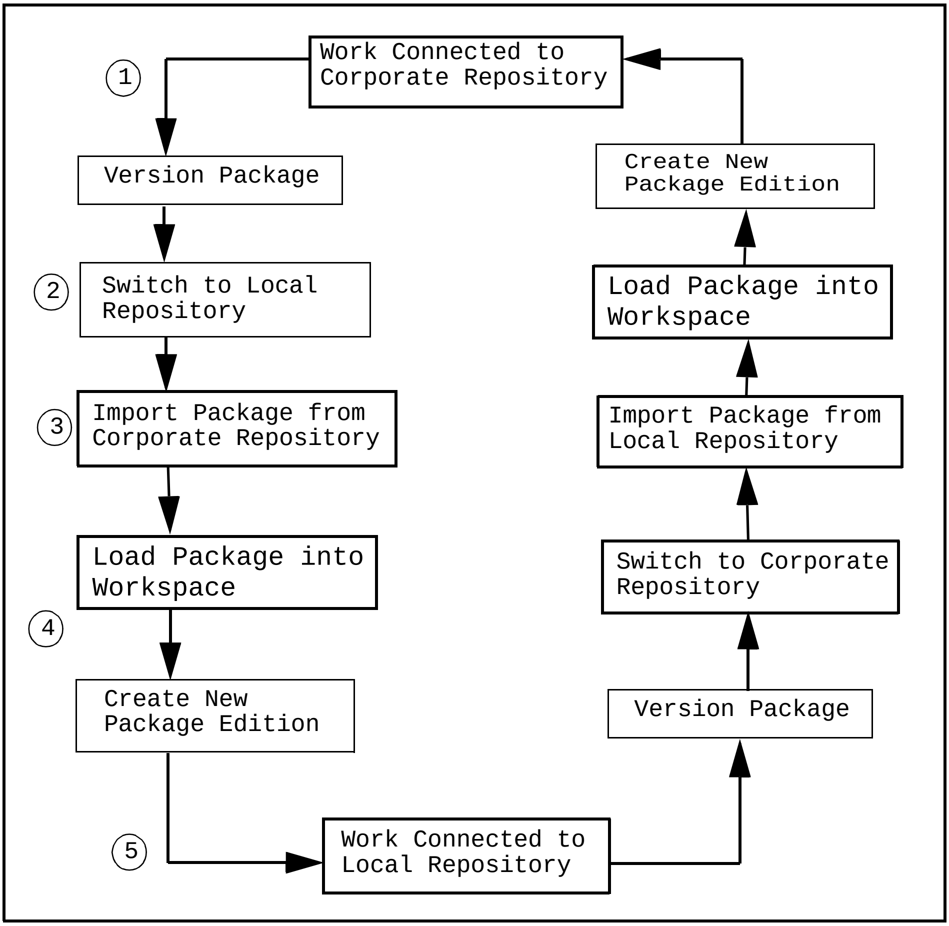

Work Connected to Corporate Repository
1
Version Package
2
Switch to Local Repository
3
Import Package from Corporate Repository
Load Package into Workspace
4
Create New Package Edition
5
Work Connected to Local Repository
Version Package
Switch to Corporate Repository
Import Package from Local Repository
Load Package into Workspace
Create New Package Edition

Figure 40. Moving Your Work between Repositories

Follow the steps shown in Figure 40 and detailed below, to move your work between two repositories.

1. While working in the standard corporate repository, you decide it is time to go home and you would like to continue working at home the following day. Create a new version of your package, remembering to first version and release all classes in the package.

2. Switch to your local repository.

   You can switch to your local repository in one of two ways. Use option A below if you are using Windows 95 as your client or if you prefer not to run the EMSRV process. Use option B if you are running EMSRV on your workstation that you are transporting home for work offline.

A. It is not possible to switch between a repository that was not accessed through EMSRV and one that is accessed through EMSRV. If you do not want to run EMSRV on your laptop or you are running Windows 95 where EMSRV is not supported, you must change the ide.ini file to establish a direct connection to the local repository. It is best to maintain two separate .ini files (for example, *homeide.ini* and *workide.ini*) and switch between them.

   Shut down VisualAge for Java, copy *ide.ini* to *workide.ini*, rename your *homeide.ini* to *ide.ini* and restart VisualAge for Java.

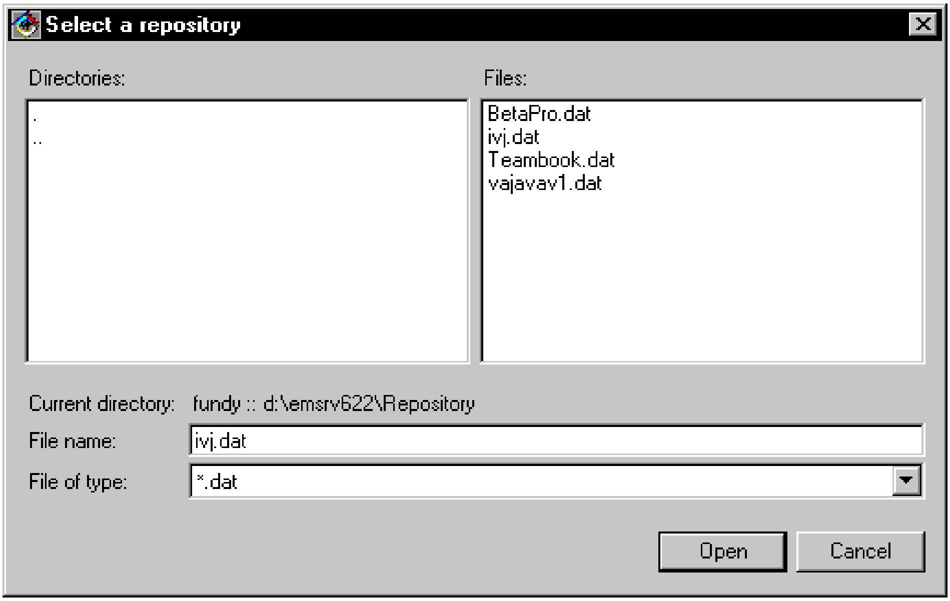
B. Use the *Change Repository* option of the Repository Explorer to connect to your local repository. Enter your local machine host name or IP address and specify the file name of your local repository.

You may notice that your log file contains numerous messages that were generated during the change operation as a result of the recaching of pointers to program elements that are in your workspace but are not in the repository to which you are switching. These messages can be safely ignored.

3. Now you can use the VisualAge for Java import facility to import the version of the package you have just created in the corporate repository. In the import SmartGuide, specify that you want to import a package and select the name and version to import.
4. Add the newly imported package version to your workspace.
5. Create a new edition of the package and continue working connected to your local repository.

Important Info	Importing or exporting?
	Importing the package from the source repository into the target repository is just one way of moving a package between repositories. It is equally valid to export the package from the source to the target repository.

To connect your workspace to a different repository, select *Change Repository...* from the *Admin* menu option of the Repository Explorer. You are prompted for a host name or IP address of a server to which to connect. By default you are connected to the current team server. Then you are presented with a dialog that enables you to traverse the directory structure of the current drive of the file server to select another repository. Notice in Figure 41 that there may be a multiple repository files that are managed by the single EMSRV process to which you are connected.



.
..

BetaPro.dat
ivj.dat
Teambook.dat
vajavav1.dat

Current directory: fundy :: d:\emsrv622\Repository

File name: ivj.dat

File of type: *.dat

Figure 41. Selection Dialog to Connect to Another Repository

You can also connect to a repository on another drive on the server, but you have to explicitly enter the fully qualified path and file name in the *File name* field.

Switching between Two Clients and Two Repositories

A developer has a machine in the office connected to the corporate repository for normal development activities but also uses a home PC when working offline. The workspace on the home PC is connected to a local repository. The issue is one of moving and synchronizing work between the two environments.

In this situation it is best to structure the team so that each package is the responsibility of a single developer. This structure facilitates the import/export procedure and minimizes the effort required to reconcile divergent packages. A suggested process is shown in Figure 42. Before such a process can be followed it is important to synchronize the corporate and home repositories. The home repository is likely to be much smaller than the corporate one, but all relevant packages from the corporate repository must be loaded into the home repository. When new package baselines are established in the corporate environment, they must also be loaded in the home environment.

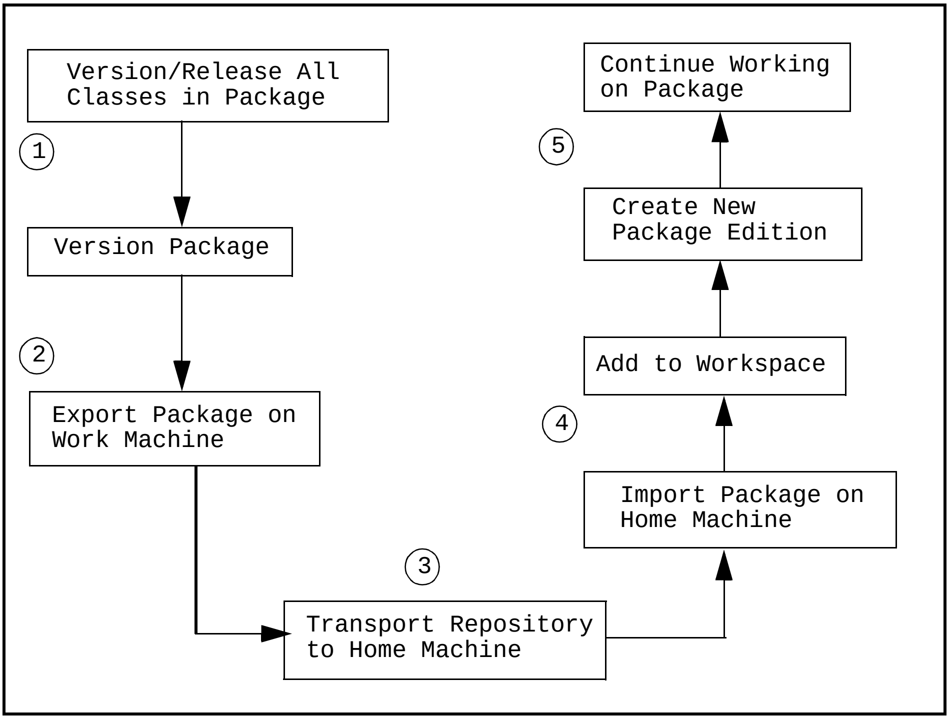

Version/Release All Classes in Package
1
Version Package
2
Export Package on Work Machine
3
Transport Repository to Home Machine
Import Package on Home Machine
4
Add to Workspace
Create New Package Edition
5
Continue Working on Package

Figure 42. Copying a Package across Repositories

1. Before you can carry out a repository style export on a package, the package must be a version.
2. Export the package using the repository export. Essentially you are creating a new repository that contains only the package you are exporting.
3. To easily transport your temporary repository, it must fit on a suitable medium such as a 1.44 MB diskette. In practice we found that a package containing about 20000 lines of Java source code fits on a single floppy disk (remember that the repository contains the Java bytecodes as well). If your temporary repository is greater than 1.44 MB, you may be able to use a compression program to fit it on a floppy disk. We found that the popular Windows utility, WinZip, can achieve compression ratios of about 80% on repository files. Using WinZip, you can export the entire set of Java 1.1 packages to a single floppy disk. If this still does not satisfy your requirements, you will have to find an alternative way of transporting your repository (file transfer, for example) or use a different style of export (.java files, for example, as discussed in the box below).

4. After importing the package into the home repository, load it into the workspace to make it available.
5. Create a new edition of your package and continue working on it.

The whole process is repeated in reverse order when you want to move your package back to your work machine.

Warning!	Potential complications
	Quite often you will find that you have to make changes to classes in other packages as part of your work at home. When this happens you need to replicate those changes in your corporate environment so that the normal reconciliation process can take place. To replicate, export the changed classes as .java files and later import them into your corporate environment. This approach allows the class owners to see the changes and, if appropriate, merge them into their packages. You also have to export your classes as .java (and optionally .class) files if your package is too big for easy transportation. You must exercise caution when using this mechanism to ensure that all changed classes are included in your export. Otherwise your home and corporate repositories will get out of step. We recommend versioning your classes before carrying out the export operation even though with VisualAge for Java you can export a class that is an open edition.

Multiple Workspaces

Our discussion has assumed that each developer has his or her own personal workspace that is constantly changing: Projects and packages are added; classes and interfaces are created; program elements are deleted; the workspace is switched between repositories. This is a perfectly normal scenario and is likely to be the way most work is carried out. However, there are situations where it makes sense to have multiple workspaces and to switch among them in appropriate circumstances. We explore some of those situations here and describe the mechanism of switching workspaces.

Workspaces can be highly dynamic. You can tailor the contents of your workspace by adding and deleting projects and packages. But these operations take time and can be prone to human error if the wrong project or package is accidentally loaded. Sometimes it is useful to establish *reference* *workspaces,* which contain a desired configuration of project and package versions.

A developer can establish reference workspaces by simply making a copy of a configured workspace and saving it with a different but appropriate name. It has to be renamed to *ide.icx* in order to start VisualAge for Java Enterprise in the usual way. Figure 43 shows a process to store and rename a workspace.

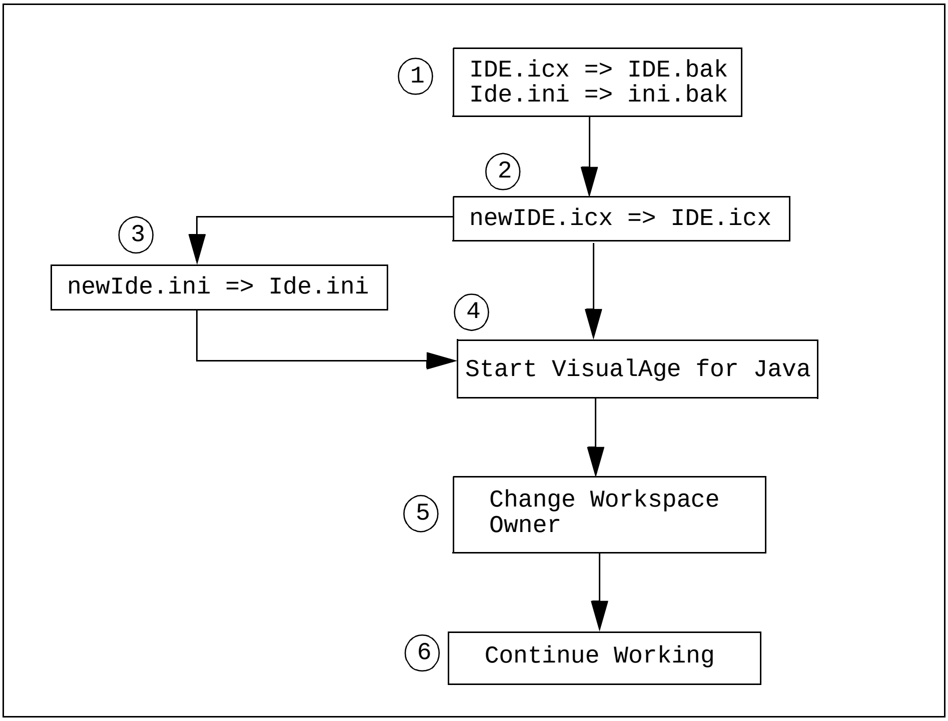


Figure 43. Copying Workspaces

The procedure here is quite straightforward:

1. Take a copy of your workspace file and your .ini file in case they are needed for recovery. These files are in the *x:\IBMVJava\Ide\program* directory where x: is the directory in which the developer installed the VisualAge for Java client. This step can be omitted if you take a regular workspace backup.
2. Copy the reference workspace file to your local workspace file.
3. If you are connecting to a different repository, take a copy of the reference .ini file. If you do not do this, you have to edit your own .ini file to change the repository details.

4. Start VisualAge for Java Enterprise.
5. The reference workspace has probably been saved with someone else (usually the administrator) as the workspace owner. You must change the workspace owner to yourself.
6. Continue working with your new workspace.

Let us consider some reasons why you and your team may want to maintain multiple workspace files:

❑ Before embarking on a new project, you want to ensure that all developers start off with a consistent view of the environment. You construct a workspace with the new project and required packages created and initialized. You also add any other projects and packages that may be needed.

❑ You could construct a workspace that reflects the delivered state of software for a particular customer. You can then quickly and easily replicate that customer's environment.

❑ It is also useful to have a reference workspace that developers can revert to in situations where their personal workspace has become corrupted.

❑ You need to connect to multiple repositories on a regular basis. The process of changing the repository to which a workspace is connected can be time consuming, especially if the new repository is located remotely with a slow connection. The workspace has to be fully synchronized with the new repository before the change can be completed. A valid alternative is to have a special workspace file that you use for connecting with the remote repository. When you need to access the repository, close your existing VisualAge for Java session, copy the other workspace, restart VisualAge for Java, and continue working with the new workspace.

Remember that making use of multiple workspaces in this way is for convenience only. The repository is where everything is stored, and it is always possible to reconstruct any workspace given sufficient time and knowledge about what is to be cached in the workspace.

A Program Element State Transitions

In this appendix we document the allowable state transitions for projects, packages, and classes. In Chapter 3, “Understanding the Basics” on page 13, we state that software elements can exist in three states—as scratch editions, open editions, and versioned editions or, more simply, versions. We also describe the notion of *releasing,* where a program element is made available to its containing program element. While releasing does not actually change the state of a program element, it can sometimes be useful to think about it in that way. For example, releasing is an action on a class that changes the subsequent actions that can be applied to that class. We therefore introduce the notion of a released class and, for packages, a released version and a released open edition.

So far our discussion has assumed that we are talking about program elements that are present in the workspace. After all, it is only in the workspace that elements can be changed. However, program elements must be created at some time and can exist only in the repository after they have been removed from the workspace. Program elements can also be removed completely (admittedly with some difficulty) from the repository. Therefore we introduce the states of *not existing* for program elements before their creation or after their removal, and *repository only* for elements that do not exist in the workspace.

We define a transition as the change of a program element from one state to another. This definition is somewhat loose because the nature of such a transition can vary. For example, when we talk about moving from an open edition to a version, the program element remains the same but its name changes. When we talk about moving from a version to an open edition, however, we are actually creating a new instance of the program element. Nevertheless, in all cases, the semantics of the transition should be clear. For each transition we document the situation in which the transition might occur, the prerequisites that must be satisfied before it can occur, and the mechanism by which it occurs.

Project State Transitions

Figure 44 shows the 5 states in which projects can exist and the 10 valid transitions among them.

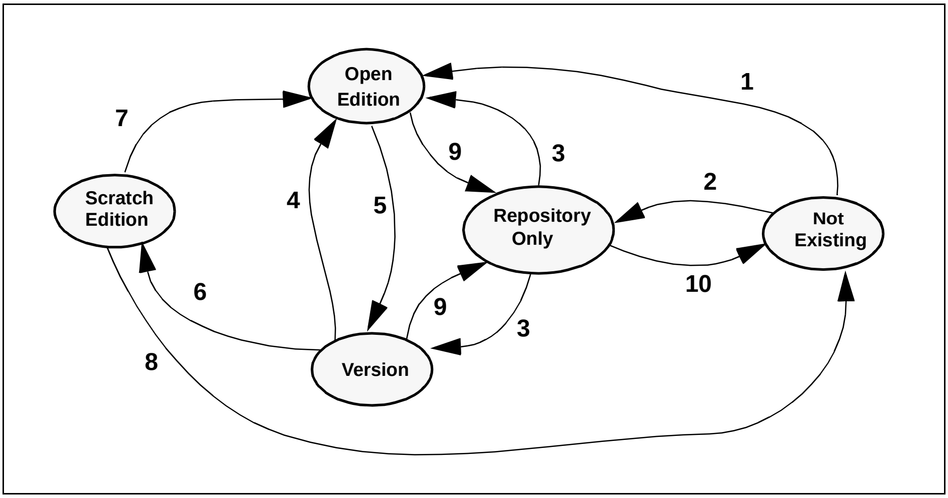


Figure 44. Valid State Transitions for Projects

These transitions are listed below and described in the subsequent sections:

1. Not Existing -> Open Edition
2. Not Existing -> Repository Only
3. Repository Only -> Version/Open Edition
4. Version -> Open Edition
5. Open Edition -> Version
6. Version -> Scratch Edition
7. Scratch Edition -> Open Edition
8. Scratch Edition -> Not Existing
9. Version/Open Edition -> Repository Only
10. Repository Only -> Not Existing

Not Existing -> Open Edition (1)

Situation

This is how brand new projects are created.

Prerequisites

- ❑ Any registered team user can create a new project. The user becomes the owner of the project.

Mechanism

- ❑ From the Project view of the Workbench, bring up a SmartGuide by selecting *Selected > Add > Project...* The same SmartGuide can be invoked by selecting *Projects > Add > Project...* from the Managing view of the Workbench.
- ❑ Select the *Create a new project named:* radio button and type the name of your new project. If a project of the same name already exists in your workspace, you are prompted with a message at the bottom of the SmartGuide and you will not be able to click the **Finish** button. Otherwise, click on the **Finish** button, and the new project is created in your workspace and in the repository. If the project already exists in the repository, you will be asked if you want to create a new edition of that project. You cannot have more than one project with the same name in the repository.

Not Existing -> Repository Only (2)

Situation

A project is created in the repository as a result of an import operation.

Prerequisites

- ❑ Any registered user can import a project into the repository. The administrator becomes the owner of such projects.
- ❑ The project to be imported must be a version in the other repository.

Mechanism

- ❑ Select *File > Import* from the Workbench to invoke the SmartGuide.
- ❑ Complete projects can be imported from another repository by selecting the *Repository* radio button.
- ❑ The SmartGuide asks you to provide the IP address and file name of the other repository.

❑ When you have selected a valid repository, you are presented with a list of projects (and packages) in the other repository, and you can select the projects and versions you want to import.

Repository Only -> Version/Open Edition (3)

Situation

You must bring a project into the workspace before you can work on any of its packages. Project versions and open editions are handled in the same way.

Prerequisites

❑ Any registered user can load a project into the workspace. The load will not be successful if the project to be loaded contains a package that already exists in the workspace as part of another project. It is a rule in VisualAge for Java Enterprise that the same package can exist in multiple projects in the repository, but not in the workspace. The entire load is an atomic operation, and the operation will either succeed completely or not load any portion of the project.

Mechanism

❑ Invoke the Add Project SmartGuide by selecting *Projects > Add > Project...* from the Managing view of the Workbench or *Selected > Add > Project...* from the Project view.

❑ Check *Add projects from the repository* and select the projects and the editions you want to load. Click on the **Finish** button, and the selected project editions are loaded into your workspace.

Alternatively:

❑ Go to the Repository Explorer and highlight the project and edition you want to load.

❑ Select *Editions > Add to Workspace* (or use the pop-up menu in the Editions pane), and the selected project edition is loaded into your workspace. A project that already exists in the workspace can be replaced by another version/open edition from the repository.

❑ Select *Projects > Replace With > Another Edition...*, and you are asked to select from the list of editions in the repository. You can quickly revert to the previous edition by selecting *Projects > Replace With > Previous Edition* or, if you are currently working with an open edition and it has changed in the repository, you can bring your workspace up-to-date by selecting *Projects > Replace With > Current Edition*.

Version -> Open Edition (4)

Situation

You should create a new edition of a project when you want to make a change to the project. Remember that a project version is immutable—if you want to change it in any way such as creating new package editions or even changing the comment associated with the project, you must first create a new edition.

Changing the owner of a project, however, does not force you to create a new edition.

Prerequisites

❑ Only the project owner is allowed to create a new edition of the project.

Mechanism

❑ Select *Projects > Manage > Create Open Edition* (or *Selected > Manage > Create Open Edition*) to create a new project edition. The edition is named by appending the current data and time in parentheses to the project name.

If you attempt to add a new package to a project version, the VisualAge for Java SmartGuide asks you if you want it to create a new edition of the project.

(There is actually a way for anyone to create a new edition of a project. First delete the project from your workspace. Then go into the Add Project SmartGuide and create a new project of the same name. The SmartGuide tells you that a version already exists in the repository and asks you if you want to create a new edition. If you answer *yes*, a new edition of the project is created with you as the owner.)

Open Edition -> Version (5)

Situation

You create a new project version from an open edition when you want to establish a baseline for your entire project. Once versioned, the project is immutable.

Prerequisites

❑ Only the owner of a project can create a new version of the project.

❑ All packages in the project must be versioned and must be released (*released versions* in our terminology).

Mechanism

- ❑ Select *Projects* > *Manage* > *Version* from the Managing view of the Workbench.

Version -> Scratch Edition (6)

Situation

A scratch edition of the project is automatically created whenever an open edition of an existing package is created in a project version.

Prerequisites

- ❑ None. A scratch edition is created for any user. Note that only package owners are allowed to create an open edition of a package.

Mechanism

- ❑ None. Creating a scratch edition of the project happens automatically.

Scratch Edition -> Open Edition (7)

Situation

Because VisualAge for Java can automatically create scratch editions of a project, it may be necessary to convert your scratch edition to an open edition. You do this conversion if you want to eventually release package elements into the project and thus make them available to other users.

Prerequisites

- ❑ Only the project owner can convert a scratch edition to an open edition.

Mechanism

- ❑ Select *Projects* > *Manage* > *Create Open Edition* from the Managing view of the Workbench.

Scratch Edition -> Not Existing (8)

Situation

If you do not want to retain the changes you made to the project that caused the scratch edition to be made, you can completely remove the scratch edition.

You can remove the scratch edition in two ways. As project owner you can delete the project and thus remove the project from the workspace.

Alternatively you can replace the scratch edition with another edition from the repository and thus remove the scratch edition from the workspace. Because the scratch edition existed only in the workplace, it is completely deleted.

Note that nothing ever really disappears from VisualAge for Java! If you created new package editions in your project scratch edition, they will still be in the repository after the project scratch edition is deleted. However, you have to search for the new package editions in the packages list because they do not belong to any project.

Prerequisites

- ❑ Only the project owner can explicitly delete a project scratch edition.
- ❑ Anyone can replace a scratch edition with another edition from the repository.

Mechanism

- ❑ To delete a project, select *Projects* > *Delete...* from the Managing view of the Workbench, and the project is completely removed from the workspace. However, all previous project versions and open editions still exist in the repository and can be reloaded at any time. Because the scratch edition existed only in the workspace, it disappears completely.
- ❑ To replace the scratch edition with another edition, select *Projects* > *Replace With* from the Managing view of the Workbench. You have a choice of three submenus:
  - *Current Edition*: Load the edition that has the same name as the edition in your workspace. If the current edition is not a version, its contents may have changed since you last loaded it.
  - *Previous Edition*: Load the version of the project that was created chronologically before the edition currently in your workspace.
  - *Another Edition...*: This option presents you with a list of all project editions in the repository. You are invited to select the edition to load.

Version/Open Edition -> Repository Only (9)

Situation

VisualAge for Java provides menu options for deleting program elements—however, you only delete in the context of your personal workspace. The elements themselves remain in the repository. You use this option simply to clear up your workspace. Having a smaller workspace reduces your startup and shutdown times. So if you are no longer working on a particular project, you should remove it from your workspace.

You may also have to remove a project from your workspace if it contains a package that is also contained in another project that you want to load. Only one copy of a package can be in your workspace at a time.

Prerequisites

❑ None. Any user can delete a project version or open edition from the workspace.

Mechanism

❑ Select *Projects* > *Delete* from the Managing view of the Workbench.

Repository Only -> Not Existing (10)

Situation

Eventually you want to remove old projects from your repository. Otherwise the repository continues to grow.

Prerequisites

❑ Only the project owner or the administrator can purge projects or project editions. A project edition cannot be purged if it is loaded in your workspace. (Check that there are no asterisks beside any of the project editions in the Repository Explorer.) Only the administrator can compact the repository.

Mechanism

❑ Completely removing projects is a two-stage process:

- First you must purge the project and/or project editions. To purge the entire project, in the Repository Explorer highlight the project and select *Names* > *Purge*. To purge one or more editions, highlight the required editions and select *Editions* > *Purge*. If all editions of a project are purged, the project is also purged.
- The second stage involves compacting the repository. See “Compacting the Repository” on page 89 for more information.

Package State Transitions

Figure 45 shows the 7 states for packages and the 14 transitions.

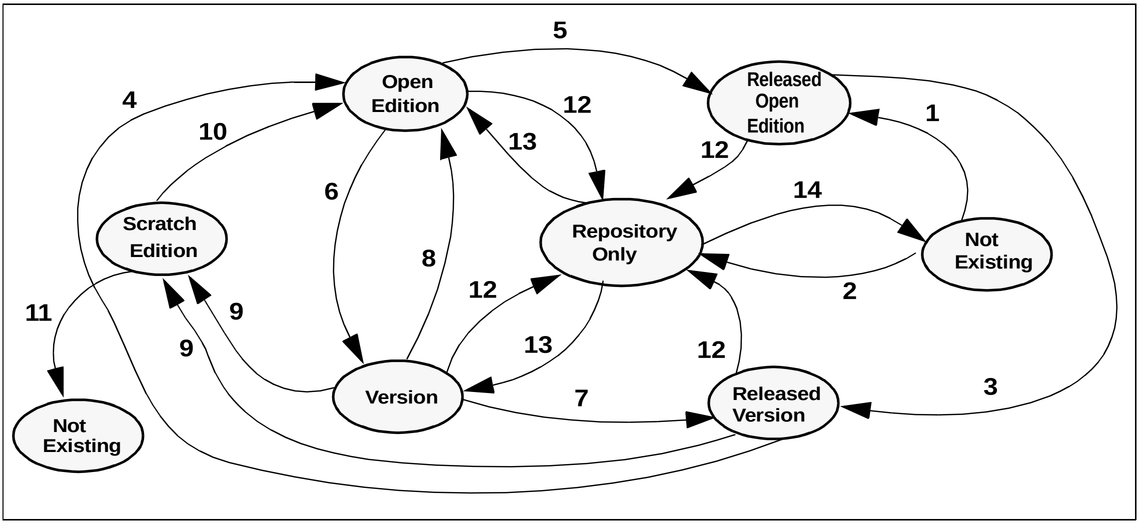


Figure 45. Valid State Transitions for Packages

These transitions are listed below and described in the subsequent sections:

1. Not Existing -> Released Open Edition
2. Not Existing -> Repository Only
3. Released Open Edition -> Released Version
4. Released Version -> Open Edition
5. Open Edition -> Released Open Edition
6. Open Edition -> Version
7. Version -> Released Version
8. Version -> Open Edition
9. Version/Released Version -> Scratch Edition
10. Scratch Edition -> Open Edition
11. Scratch Edition -> Not Existing
12. (Released) Open Edition/(Released) Version -> Repository Only
13. Repository Only -> Open Edition/Version
14. Repository Only -> Not Existing

Not Existing -> Released Open Edition (1)

Situation

There are two situations where this transition can occur: creating a brand new package and importing elements (.class files, .java files, jar files) from outside VisualAge for Java into a package that does not exist.

A package must always be created within the context of a project. You typically create a new package when a major new work application is being initiated. You import an existing package when you want to copy external code into VisualAge for Java. A major difference between importing directly from another repository and importing files is that in the former case you import directly into the repository, and in the latter case you import into a package in your workspace.

Prerequisites

- ❑ The user must be the project owner.
- ❑ The project must be an open edition.
- ❑ For package creation, a package of the same name must not exist in any project in the user's workspace.
- ❑ For package import, if a package of the same name already exists in a project currently loaded in the workspace, the new classes will be added to it. This applies even if the package is in a project other than the one specified on import. In this situation, all prerequisites for creating classes in a package apply.

Mechanism

- ❑ Creating a new package:
  - Select *Packages > Add > Package...* from the Managing view of the Workbench. This brings up the Add Package SmartGuide.
  - Select the *Create a new package named:* radio button and type your new package name. You may also add new package group members at this stage if you want. Package group members can create new classes and release them into your package.
- ❑ Importing from files:
  - Select *File > Import...* from the Workbench. This brings up the Import SmartGuide.
  - Select the project to import to and the type of import (class files, java files, JAR file, or entire directory)
  - The next screen in the SmartGuide allows you to select the files or directory to import.

Not Existing -> Repository Only (2)

Situation

This transition occurs when you import a package from another repository into your VisualAge for Java repository.

Prerequisites

❑ Any registered user can import a package into the repository. The administrator becomes the owner of such packages.

❑ The package you want to import must be a version in the other repository.

Mechanism

❑ A package can be imported as part of a project as detailed in the corresponding transition for projects.

❑ A package can also be imported independently:

- Select *File* > *Import...* from the Workbench. This brings up the Import SmartGuide.
- Select the type of import (Repository).
- The next page in the SmartGuide asks you to provide the IP address or host name of the server and the name of the repository file from which to import.
- When you have selected a valid repository, you are presented with a list of packages (and projects) in the repository. Select the package and package edition you want to import.

Released Open Edition -> Released Version (3)

Situation

You should convert a released open edition to a released version when you want to make a new package baseline. Once versioned, the package cannot be changed.

Prerequisites

❑ You must be the package owner.

❑ All classes in the package must be released (which also implies that they must be versioned).

Mechanism

- ❑ Select *Packages* > *Manage* > *Release* from the Managing view of the Workbench.

Released Version -> Open Edition (4)

Situation

You can create an open edition of a package from a released version when you want to make changes to the package.

Prerequisites

- ❑ You must be the package owner.
- ❑ The owning project must be an open edition. If not, a scratch edition of the project will be created. You do not need to be the project owner to create a new package edition.

Mechanism

- ❑ Select *Packages* > *Manage* > *Create Open Edition* from the Managing view of the Workbench.

Open Edition -> Released Open Edition (5)

Situation

This transition allows you to release a package to its project before you have versioned the package. This is a very useful facility when you want to create a rolling baseline from which all developers can synchronize their workspaces. When a developer loads the project, only classes that have been released to the package are visible. As other classes are released, the developer can reload the project to update the workspace. This facility avoids cluttering the repository with package versions that are created with the sole purpose of establishing a baseline.

Prerequisites

- ❑ You must be the project owner or package owner.

Mechanism

- ❑ Select *Packages* > *Manage* > *Release* from the Managing view of the Workbench.

Open Edition -> Version (6)

Situation

You should create a package version from an open edition when you want to freeze the contents of the package before releasing it into its project.

Prerequisites

❑ You must be the package owner.

❑ All classes in the package must be released (which also implies that they must be versioned).

Mechanism

❑ Select *Packages* > *Manage* > *Version* from the Managing view of the Workbench.

Version -> Released Version (7)

Situation

You can release a version of your package when you want to make it available to the enclosing project.

Prerequisites

❑ You must be the project owner or package owner.

Mechanism

❑ Select *Packages* > *Manage* > *Release* from the Managing view of the Workspace.

Version -> Open Edition (8)

Situation

You can create an open edition of a package from a version when you want to make changes to the package.

Prerequisites

❑ You must be the package owner.

❑ By definition, the owning project will already be an open (or scratch) edition.

Mechanism

❑ Select *Packages* > *Manage* > *Create Open Edition* from the Managing view of the Workspace.

Version/Released Version -> Scratch Edition (9)

Situation

This transition occurs automatically whenever a new edition of an existing class or interface is created in a versioned package.

Prerequisites

❑ None. A scratch edition will be created for any user, even if that user is a member of the package group.

Mechanism

❑ None. The transition happens automatically.

Scratch Edition -> Open Edition (10)

Situation

The scratch package edition exists because you created new classes or interfaces in a versioned package. You have to convert the scratch edition to an open edition to make the new classes or interfaces available to other users.

Prerequisites

❑ You must be the package owner.

❑ The project must be an open edition. If not, VisualAge for Java creates a scratch project edition.

Mechanism

❑ Select *Packages* > *Manage* > *Create Open Edition* from the Managing view of the Workspace.

Scratch Edition -> Not Existing (11)

Situation

You can remove the package scratch edition if you do not want to retain the changes you made to the package that caused the scratch edition to be created.

You can remove the scratch edition in two ways. As package owner, you can delete the package and thus remove the package from the workspace and disassociate it from its project. However, this is unlikely to be the behavior you want. The simplest approach is to replace the scratch edition with another edition from the repository. As the scratch edition is not stored in the repository, the replace deletes the scratch edition.

As with scratch project editions, removing a scratch package edition does not completely remove any of its class editions. These are still stored in the repository. To see them you must use a class browser and examine editions of the class.

Prerequisites

- ❑ To explicitly delete a scratch edition, you must be the package owner.
- ❑ Anyone can replace a scratch edition with another edition from the repository.

Mechanism

- ❑ To delete a package select *Packages* > *Delete...* from the Managing view of the Workbench.
- ❑ To replace with another edition select *Packages* > *Replace With* from the Managing view of the Workbench. You have a choice of four submenus:
  - *Released Edition*: Load the edition that has been released to the project.
  - *Current Edition*: Load the edition that has the same name as the edition in your workspace. If that edition is not a version, its contents may have changed since you last loaded it.
  - *Previous Edition*: Load the version of the package that was created chronologically before the edition currently in your workspace.
  - *Another Edition...*: This option presents you with a list of all package editions in the repository. You are invited to select the edition to load.

(Released) Open Edition/(Released) Version -> Repository Only (12)

Situation

You can delete a package if you have the correct privileges. Deleting a package has two effects:

- The package is no longer loaded into your workspace. However, it still exists in the repository and can be accessed through the Repository Explorer.
- The association between the project and the package is removed, even in the repository. In the repository, a package can be part of zero, one, or many projects. In a workspace it must belong to a single project.

Prerequisites

❑ You must be the package owner, project owner, or administrator.

❑ The project must be an open edition.

Mechanism

❑ Select *Packages* > *Delete...* from the Managing view of the Workbench.

Repository Only -> Open Edition/Version (13)

Situation

A package can be loaded from the repository into a specific project in the workspace. Typically this will be an infrequent operation carried out when a project is being set up and defined. When package editions are loaded in this way, they retain their original package owner and package group members.

Prerequisites

❑ You must be the owner of the project or the administrator.

❑ The project must be an open edition.

❑ A package of the same name must not exist in the same or any other project in the workspace.

Mechanism

❑ From the Managing view of the Workbench select *Packages* > *Add* > *Package*. This brings up the Add Package SmartGuide.

❑ Select the *Add packages from the repository* radio button and a list of available packages and editions will be displayed.

❑ Select any number of package editions and click on the **Finish** button.

Repository Only -> Not Existing (14)

Situation

Packages can be purged from the repository if they are no longer required. You can also purge individual package editions. Even when purged, packages can be restored, provided that the repository has not been compacted.

Prerequisites

- ❑ Only the package owner or the administrator can purge packages or package editions.
- ❑ The package must not be currently loaded into the workspace.
- ❑ Only the administrator can compact the repository.

Mechanism

- ❑ Completely removing a package from the repository is a two-stage process:
  - First you must purge the package and/or package editions. In the Packages view of the Repository Explorer, select *Names* > *Purge...* to purge a complete package or select *Editions* > *Purge...* to purge a single edition.
  - The second stage involves compacting the repository. See “Compacting the Repository” on page 89 for more information.

Class State Transitions

Figure 46 shows the 5 states in which classes and interfaces can exist and the 7 valid transitions among them.

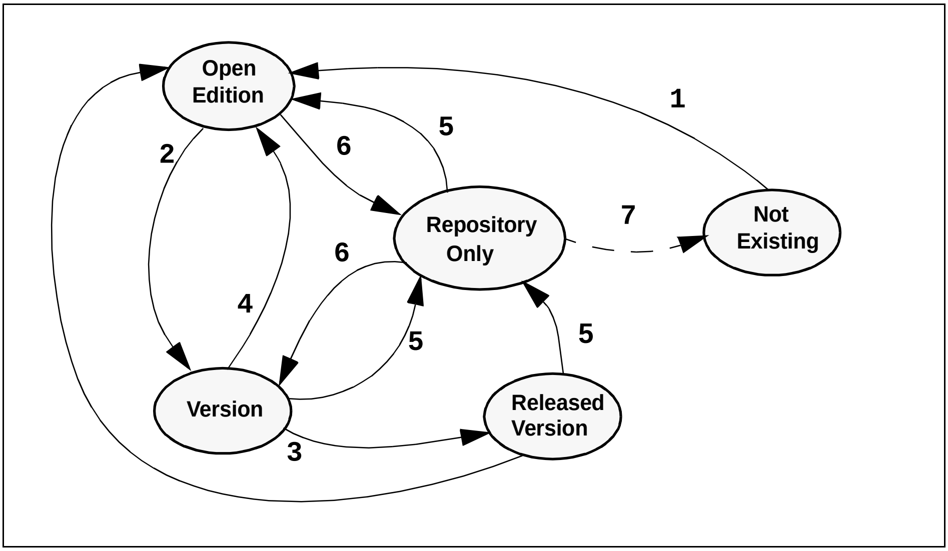


Figure 46. Valid State Transitions for Classes

These transitions are listed below and described in the subsequent sections:

1. Not Existing -> Open Edition
2. Open Edition -> Version
3. Version -> Released Version
4. Version/Released Version -> Open Edition
5. Version/Open Edition/Released Version -> Repository Only
6. Repository Only -> Version/Open Edition
7. Repository Only -> Not Existing

Not Existing -> Open Edition (1)

Situation

This is how brand new classes and interfaces are created.

Prerequisites

❑ The package in which the class is being created must be an open edition. If it is not, the SmartGuide will change the package to an open edition for you automatically. You must be the package owner.

❑ If the package is already an open edition, you must be a package group member to create a new class.

Mechanism

❑ Classes are created using the Create Type SmartGuide. One way to create a new class is through the *Types* > *Add* > *Class/Interface* menu option in the Managing view of the Workbench. The creator of the class becomes the class owner and the class developer of the new edition.

Open Edition -> Version (2)

Situation

You create a new version of a class from an open edition when you want to freeze the definition of the class. Versioned classes cannot be changed.

Prerequisites

❑ Only the class developer can version a class.

Mechanism

❑ Select *Types* > *Manage* > *Version...* from the Managing view of the Workbench. You can version multiple classes at a time and apply the same or individual version numbers to each class.

Version -> Released Version (3)

Situation

You release a class to its enclosing package when you want to make it available to other users who load that edition of the package.

Prerequisites

❑ Only the class owner can release a class.

❑ The enclosing package must be an open edition.

Mechanism

❑ Select *Types* > *Manage* > *Release* from the Managing view of the Workbench

Version/Released Version -> Open Edition (4)

Situation

If you want to make changes to a versioned class, you must first create an open edition of the class.

Prerequisites

❑ Anyone can create an open edition of an existing class. If the package to which the class belongs is not an open edition, a scratch edition is created automatically.

Mechanism

❑ Select *Types* > *Manage* > *Create Open Edition* from the Managing view of the Workbench.

Version/Open Edition/Released Version -> Repository Only (5)

Situation

You can delete a class when it is no longer required. Deleting a class removes it from the enclosing package and from the workspace. The class remains in the repository and is still implicitly associated with the same package. You can load the class from the repository back into any edition of the same package at a later stage.

Prerequisites

❑ Only the class owner can delete a class.

❑ The enclosing package must be an open edition.

Mechanism

❑ Select *Types* > *Delete...* from the Managing view of the Workbench.

Repository Only -> Version/Open Edition (6)

Situation

This is how you retrieve a previously deleted class from the repository.

Prerequisites

- ❑ The enclosing package must be an open edition. VisualAge for Java does not automatically create a scratch edition for you.
- ❑ You must be a group member of the enclosing package. You automatically become the new owner of the class you load in this way. The class developer remains the same, so when you load an open edition that has another user as its developer, you cannot change the class.

Mechanism

- ❑ Select *Types* > *Add* > *Class/Interface...* > *Type from Repository...* from the Managing view of the Workbench. You will be presented with a list of classes and interfaces that belong to the currently selected package in the repository.
- ❑ Select a class and an edition of the class to load.

Repository Only -> Not Existing (7)

Situation

This transition is shown as a broken line because it is not possible to explicitly remove classes from the repository. Classes are purged when the package they belong to is purged.

B EMSRV

In this appendix we provide details for running the EMSRV program on your VisualAge for Java server machine. Examples are also provided showing different uses on the different operating platforms.

EMSRV Startup Options

Tables 4–6 describe the parameters that can be used to manage the operation of EMSRV. Not all options are available on all platforms.

The key to the Platform column is as follows:

all	all platforms
NetWare	Novell NetWare
NT	Windows NT
OS/2	OS/2 Warp
UNIX	IBM AIX, Sun Solaris, and HP-UX

Table 4. EMSRV Startup Options (Part 1)

Parameter	Platform	Description
-A<0,1>	all	The file system requires locks. The default value is 0, indicating that the file system does not require locks.
-a <seconds>	UNIX	Sets the number of seconds before a connection is deemed inactive. The default is 360.
-b <kbytes>	all	Sets the low-volume threshold warning in kilobytes. The default is 10,000 KB. If the available disk space is less than the low-volume threshold value, EMSRV logs warning messages to the log file.
-f	UNIX	Sets EMSRV to run in the foreground
-h	all	Displays the help text listing valid options
-i<q,t>	UNIX	Ignores signals: q = ignore SIGQUIT; t = ignore SIGTERM. By default either of these signals causes EMSRV to terminate.
-install	NT	Installs EMSRV as a service
-lc	all	Logs messages to the console. By default messages are not written to the console.
-lf <name>	OS/2, NT	Writes the log file to file <name>. By default messages are logged to the emsrv.log file. Specify a valid path for which the EMSRV account has sufficient rights.
-lp <scnds>	UNIX	Sets the maximum number of seconds to wait for a lock. The default is 15 seconds.
-ls	UNIX	Logs messages to stdout instead of a log file. EMSRV must be run in the foreground (using -f).

Table 5. EMSRV Startup Options (Part 2)

Parameter	Platform	Description
-lt <scnds>	UNIX	Sets the maximum number of seconds to hold a lock
-M<n>	all	Specifies the maximum number of connections that can be established with EMSRV. The default is 256.
-n	UNIX	Turns off statistics gathering
-P <port #>	all	Specifies the port number that the EMSRV process uses. The default is 4800.
-p <passwd>	OS/2, NT, NetWare	Specifies the password for the EMSRV user account in order to restrict repository file access. This password is used for EMADMIN functions such as shutting down EMSRV remotely. In a Windows NT environment you must specify -p with no parameter if the user account has no password.
-R<0,1>	all	The file system releases locks on file close. The default setting is 1, which indicates that the file system releases locks when a file is closed.
-r	UNIX	Rejects users who are not in the passwd.dat file
-rd	OS/2, NT	Disables password checking of clients. This is the default setting.
-remove	NT	Removes the EMSRV service from the registry
-rn	NT, NetWare	Rejects users who do not supply a valid user name and password. EMSRV validates using the native user profiles on the server. The default grants access to all users without a user name and password.
-rp	OS/2, NT, NetWare	Rejects users who are not in the passwd.dat file. The default grants access to all users without checking this password file.
-s<0,1,2>	all	Sets the reporting level to the specified severity level: 0 = log all operations; 1 = log warnings and error messages; 2 = log errors only. On PC platforms the default is 2. On UNIX the default is 1.
-t	all	Protects existing libraries from truncation. By default, libraries can be created over existing ones (truncated to 0).
-u <user>	NT, NetWare	Specifies the EMSRV account name to be used

Table 6. EMSRV Startup Options (Part 3)

Parameter	Platform	Description
-v	UNIX	Verifies password using system authorization
-W<path>	OS/2, NT, NetWare	Specifies the EMSRV work (repository) directory. The directory <path> must be a valid path for which the EMSRV account has read/write access.
-w	all	Specifies that EMSRV should track locks for each connection
-xd	UNIX	Specifies valid devices for the database
-xn	UNIX	Allows databases to be opened on nonnative devices

Using EMSRV

In this section we provide some examples of using EMSRV. We present the examples by platform to highlight the differences.

EMSRV on OS/2

On OS/2, EMSRV does not require a user logon account to start. A password can be provided to prevent unauthorized shutdown. The following examples are run from the OS/2 command line in the directory where EMSRV is installed:

❑ `emsrv -p rainy -W c:\VajSrv\Repository -lc`

This command starts EMSRV with a work directory *c:\VajSrv\Repository*, and messages are logged to the console. The directory must already have been created. To shut down EMSRV, the password *rainy* has to be included as a parameter of the EMADMIN command. Refer to Appendix C, “EMADMIN” on page 133 for details on using EMADMIN.

❑ `emsrv -p rainy -lf c:\logs\vaemsrv1.log -rp -M40`

This command starts EMSRV, allowing a maximum of 40 concurrent client connections to the repository file. EMADMIN counts as one connection. The log file to be used for messages is *c:\logs\vaemsrv1.log*. The work directory is the directory where EMSRV is installed, for example, *c:\VajSrv*. Users are required to log on to VisualAge for Java, using a valid user name and password that is in the *passwd.dat* file. This file must be located in the work directory.

EMSRV on Windows NT

In a Windows NT environment, a user account must be defined to run EMSRV. This user is referred to as the EMSRV account and must be granted the Advanced User Right “Act as part of the operating system.” The user must also be a member of the administrators group and have read-write access to the directory where EMSRV is installed.

The program can be started in two ways; from the command line, or as a Windows NT service.

Start EMSRV from the Command Line

To following example is run from the command line in the directory where EMSRV is installed:

```
emsrv -u EMSRVUSER -p rainy -W d:\VajSrv\Repository -s1 -rp
```

This command starts EMSRV with user `EMSRVUSER`, password rainy, and work directory *d:\VajSrv\Repository*. Warning messages are logged along with error messages to *emsrv.log* in the work directory. Users are required to enter a password each time they start VisualAge for Java or attempt to change the workspace owner. EMSRV validates the user name and password on the server machine.

Install and Start EMSRV As an NT Service

From the directory where EMSRV is installed, enter `emsrv -install` at a command prompt. This installs EMSRV into the Windows NT registry.

Startup parameters can be entered as `emsrv -install <parms>`:

```
emsrv -install -u EMSRVUSER -p rainy -rp
```

To start the EMSRV service, use one of the following methods:

❑ From the Control Panel:

- Open the Services window.
- Select the EMSRV process.
- Select Automatic for the startup type if suitable parameters were entered with the `emsrv -install <parms>` command.
- Optionally add startup parameters, for example: `-s1 -M40 -lc`

  To enter a working directory with the -W parameter, include an extra backslash for each backslash in the path.
- Click on **Start** and EMSRV starts as a service.

- ❑ Using the Service Controller utility (sc.exe). This utility is available for Windows NT 4.0 as part of the Resource Kit.
  - Enter `sc start -u EMSRVUSER -p rainy` from the command line.
  - Query the service status, using `sc query EMSRV`.
  - Stop the service, using `sc stop EMSRV`.

When running as a service, EMSRV does not have to interact with the desktop, and any error messages about startup or low disk space are written to the Event Log.

EMSRV on UNIX

On UNIX platforms, the EMSRV process has the file access permissions of the UNIX account that starts it. The account name that is used to start EMSRV is referred to as the EMSRV account. The EMSRV account must have sufficient rights to read, write, and create files in the directory where the repository exists, and read and execute access on all parent directories.

Before running EMSRV you must determine whether your UNIX system is using shadow passwords or not. If your system is using shadow passwords, use emsrv.shadow. Otherwise, use EMSRV.

To run EMSRV, change to the directory where EMSRV is installed and start the program. For example:

- ❑ Start EMSRV with lock tracking turned on and with logging of messages to the console as well as the log file named *logfile*.

```
emsrv -w -lc -lf logfile
or
emsrv.shadow -w -lc -lf logfile
```

- ❑ Start EMSRV with native password verification, ignoring SIGQUIT signals, and logging only error messages.

```
emsrv -v -iq -s2
or
emsrv.shadow -v -iq -s2
```

To shut down EMSRV on UNIX, the account password is required.

C EMADMIN

In this appendix we describe the details of the EMADMIN utility and provide examples of its use.

The EMADMIN command utility enables you to communicate with EMSRV from a workstation. EMADMIN runs on Windows 95, Windows NT 4.0, OS/2 Warp 4.0, AIX 4.2.1, HP-UX 10.20, and Sun Solaris 2.6.

EMADMIN Installation

The EMADMIN utility is provided in the directory on the server where EMSRV was installed, for example, *C:\VajSrv*. Copy the *emadmin.exe* file to the workstation where you want to run this utility. You can install it on the EMSRV server, or on a separate system where the administrator is working. Do not install it on client developer workstations.

Using EMADMIN

The EMADMIN command has the following syntax:

```
emadmin [command] [host] [command modifier] [option]
```

Each command allows a host name or IP address to be specified if EMADMIN is not run on the same machine as EMSRV. Additionally, each command accepts the -P option to specify a port number other than the default 4800.

EMADMIN Commands

The EMADMIN program requires a valid command as a parameter. Running just EMADMIN with no command line parameters lists the valid commands and their options. The commands are:

Command	Description
bench	Run bench tests between the client TCP stack and the server stack. These tests can help investigate performance problems.
copy	Copy a VisualAge for Java repository on the server.
list	Display the current EMSRV connection list, or information about a specific connection.
opts	Display current EMSRV options.
stat	Display EMSRV statistics.
stop	Shut down EMSRV remotely or kill an active connection.

We describe each command below and provide examples.

EMADMIN Bench

The bench command enables IBM to determine whether there are performance problems between the client and server TCP stacks. It may take up to 5 minutes to run, depending on the machine.

The command runs a series of read tests on a temporary repository to determine the performance for different client buffer sizes. Delete this temporary repository after the test has been completed.

You may be asked to perform this test and submit the log files to IBM technical support for analysis if they suspect a performance problem between the client and the server.

Command modifiers are listed in Table 7.

Table 7. Command Modifiers for EMADMIN Bench

Command Modifier	Description
-t <tname>	Specifies the name of the test library to create. The default name is testlib.dat.
-l <lname>	Specifies the name of the log file to which to write results. If the file exists, it will be overwritten. By default, a log file, embench.log, is created.
-s <size>	Specifies the size of the sweep. The size must be between 4096 and 65536.

Example:

```
emadmin bench 9.1.150.42 -t mytest.dat -l mytest.log -s 10000
```

This command runs a bench test between the client where the command is executed and the server at 9.1.150.42. The bench sweep size is 10 KB and the files created are as named. They are created in the current directory where EMADMIN is installed.

EMADMIN Copy

The copy command enables you to make a copy of a valid VisualAge for Java repository. You can copy within a single EMSRV process or between different EMSRVs running on different machines. The command locks the repository to prevent it from being changed while the copy is in progress.

The syntax of the command is:

```
emadmin copy <source> <dest> [command modifiers]
```

where <source> is a valid VisualAge for Java repository, and <dest> is a file specification for the copy. The destination repository file must be accessible to the EMSRV account. The format is: ip_address:filename.

Command modifiers are listed in Table 8.

Table 8. Command Modifiers for EMADMIN Copy

Command Modifier	Description
-o	Specifies that the destination file may be overwritten without prompting
-p <pwd>	Specifies the password of the user who started EMSRV
-q	Indicates a quiet copy without prompting about potential problems from low disk space

Example:

```
emadmin copy 9.1.15.2:d:\repos\ivj.dat 9.1.15.2:d:\repos\backup.dat -p mypw
```

This command copies ivj.dat to backup.dat on 9.1.15.2. The repository will be locked to prevent any access from clients while the copy takes place. The copy command can be run in a batch file with the password supplied as part of the command line, which, along with the quiet option, ensures that no prompting occurs during the batch process.

Sample output of a local copy operation run on the server:

```
D:\VAJSRV>emadmin copy ivj.dat ivjcopy4.dat
...
Enter the password of the user who started EMSRV : *******

Copying localhost:ivj.dat => localhost:ivjcopy4.dat

Locking entire source file
[0:52233776]                          <=== running counter during copy operation
Unlocking source

Repository localhost:ivj.dat copied to localhost:ivjcopy4.dat.
```

EMADMIN List

The list command lists all current connections to an EMSRV. The command modifiers allow requesting more information about connections.

Use this command to verify that all clients are disconnected before stopping EMSRV.

Command modifiers are listed in Table 9.

Table 9. Command Modifiers for EMADMIN List

Command Modifier	Description
-s <cnum>	Displays the statistics for the connection specified by the connection number
-l <cnum>	Displays the active locks for the connection specified by the connection number

Example:

```
emadmin list 9.1.150.42
```

This command lists all currently active connections. Each connection has an associated ID, which you can use as the connection number in subsequent list commands, for example:

```
emadmin list 9.1.150.42 -s5
```

This command displays statistics for connection ID 5.

Sample output:

```
D:\VAJSRV>emadmin list 9.1.150.42

EMADMIN 6.22
Copyright (C) IBM Corporation 1989-1998
Server Type     : EMSRV
Server Version  : EMSRV 6.22 for Windows NT Apr  4 1998 15:36:56 (AEST)
==============================================================================
EMSRV Connection list for: 9.1.150.42

                    Active  Last
 ID IP Address      Locks   Request  Repository
------------------------------------------------------------------------------
  0 9.1.150.41      0      09:06:34  ivj.dat
  2 9.1.150.66      0      09:04:42  ivj.dat
------------------------------------------------------------------------------
==============================================================================
```

EMADMIN Opts

The opts command enables you to display the current options on an EMSRV process. The command modifier is listed in Table 10.

Table 10. Command Modifier for EMADMIN Opts

Command Modifier	Description
-s <level>	Specifies a new logging level for EMSRV and enables you to change the logging level without having to access the machine where EMSRV is running. Valid levels are: 0 Log all operations and messages 1 Log warnings and error 2 Log only errors

Example:

```
emadmin opts 9.1.150.42
```

This command displays the current options on the EMSRV running on the specified host machine.

Sample output:

```
EMSRV Options for: 9.1.150.42
------------------------------------------------------------------------------
EMSRV 6.22 for Windows NT Apr  4 1998 15:36:56 (AEST) Options

Maximum number of concurrent connections = [256]
Working directory = [d:\emsrv622\repository]
Password checking = [Disabled]

Logging level = [Error]
Log file name = [emsrv.log]

Allow connection to truncate libraries = [true]
Track EMSRV statistics = [true]
Track process file locking statistics = [false]
Process activity timeout value = [360] sec.
Sleep on lock value = [1000] msec.
Free disk space warning threshold = [10000] KBytes
Restrict libraries to local filesystems = [false]
------------------------------------------------------------------------------
```

EMADMIN Stat

The stat command displays statistics for an EMSRV process. There are no command modifiers for the stat command.

Example:

```
   emadmin stat 9.1.150.42
```

This command displays statistics for EMSRV running on the specified host machine. Sample output is provided here to show the information produced by the command:

```
   EMADMIN 6.22
   Copyright (C) IBM Corporation 1989-1998
   Server Type     : EMSRV
   Server Version  : EMSRV 6.22 for Windows NT Apr  4 1998 15:36:56 (AEST)
   ==============================================================================
   EMSRV Statistics for: localhost
   ------------------------------------------------------------------------------
   EMSRV 6.22 for Windows NT Apr  4 1998 15:36:56 (AEST)

   Total Connects:                     33  Total Disconnects:                  30
   Total Opens:                        42  Total Closes:                       38
   Active Locks                         0  Unexpected Connection Closes:        0
   Total Locks:                      4696  Total Unlocks:                    4696
   Total Reads:                     48051  Total Writes:                     7270
   Total Reads Failed On Lock:          0  Total Locks Failed On Lock:         25
   Times Lock Limit Hit:                0
   Total Requests Serviced:         68114  Requests in last interval:           0
   Largest Packet Sent:             32780  Largest Packet Received:         32784

   Server Has Been Alive For: 2 Days 0 Hours 34 Minutes 13 Seconds
   Server Working Directory : f:\VajSrv\repository
   ------------------------------------------------------------------------------
   ==============================================================================
```

EMADMIN Stop

The stop command is used to close a connection to EMSRV. It can be used to terminate a connection that can no longer communicate with the client.

The command modifiers are listed in Table 11.

Table 11. Command Modifiers for EMADMIN Stop

Command Modifier	Description
-k <cnum>	Specifies the connection number to terminate
-p <pwd>	Specifies the password of the user who started EMSRV

Example:

```
emadmin stop 9.1.150.42 -k5
```

This command terminates connection number 5. Use the list command to determine which connection number you want to terminate.

The EMSRV process is stopped completely if no connection number is provided.

Sample output:

```
F:\VAJSRV>emadmin stop 9.1.150.42

EMADMIN 6.22
Copyright (C) IBM Corporation 1989-1998
Server Type     : EMSRV
Server Version  : EMSRV 6.22 for Windows NT Apr  4 1998 15:36:56 (AEST)
Enter the password of the user who started EMSRV : *****

Server has been scheduled to stop.
```

D Migration

In this appendix we cover migration considerations when you move from a stand-alone VisualAge for Java Professional or VisualAge for Java Enterprise Version 1.0 environment to the VisualAge for Java Enterprise Version 2 environment.

Overview

The team enhancements to VisualAge for Java Enterprise Version 2 bring new and powerful capabilities to Java developers. We have long known that enterprise application development, using Java or any other language, is not done by a single developer. Now we have a robust set of tools that enable many developers to concurrently access a shared repository.

We know that managing multiple users working at the same time on program elements stored in a common repository brings along the requirement for functions like version control and change management. VisualAge for Java Enterprise team delivers these constructs by building on the basis that we already know and are comfortable with from Version 1.0.

Probably the most prevalent change that we see in VisualAge for Java Enterprise team support is the idea of ownership. In the Version 1.0 single developer environment, we had no ownership imposed on our program elements because we did not have to concern ourselves with other developers accessing them. Now we can think of all elements as having a single owner. When we export our packages and projects to a team repository, we see that they have a single user object associated with them—the administrator.

Migration from a stand-alone environment can be simply stated as the process of importing your work from a single user repository and then adding the appropriate team and ownership information so you can work with your program elements in a team environment.

In the remainder of this appendix, we detail the steps to take when moving your work products from a single user to a team repository. As with most migrations, there are several strategies. We present the options, and your project environment and policies will help you decide which strategy is best for you. The process is independent of whether you are migrating from VisualAge for Java Enterprise Version 1.0 before the availability of the team support, or VisualAge for Java Professional, which is a stand-alone environment.

The General Process

Figure 47 shows in a high-level diagram how individual developers can migrate their work from a stand-alone repository to a multiuser repository.

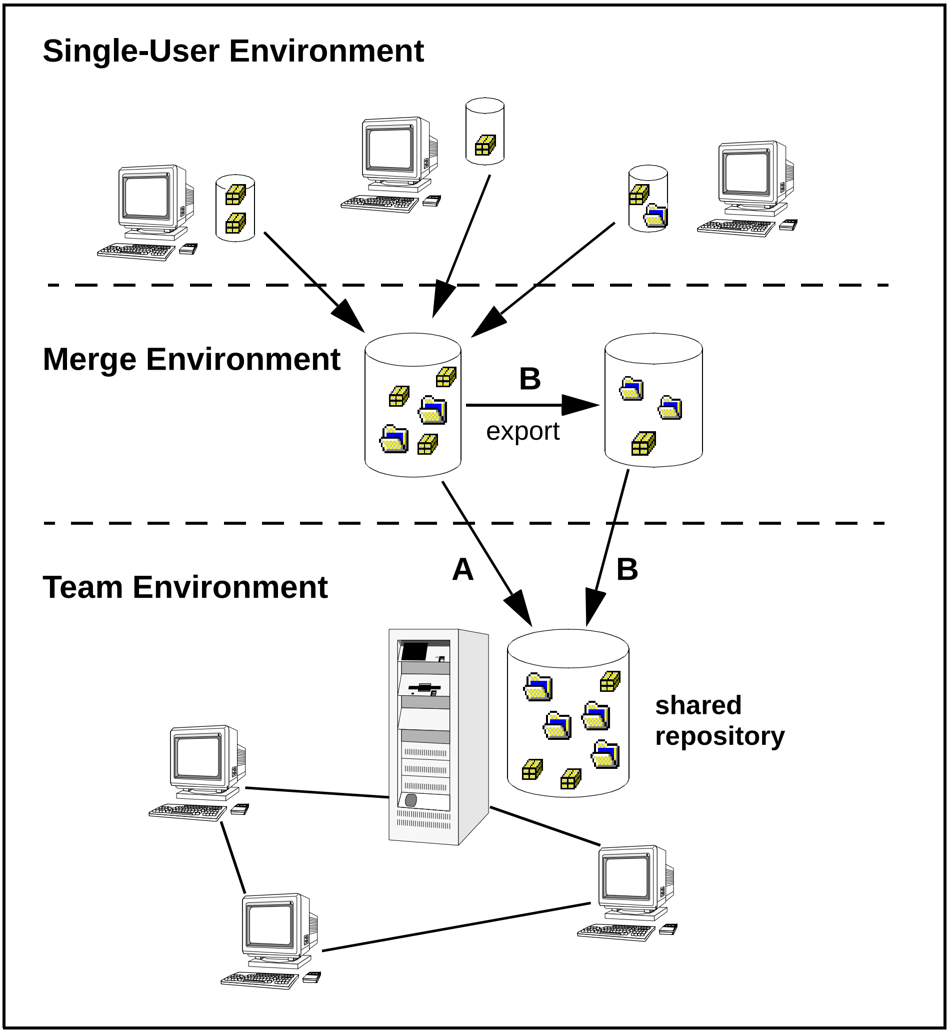


Figure 47. Migration from Single-User to Team Environment

You should merge in the way you merged your team work before creating a code baseline. Therefore the merging environment does not necessarily reside on a separate machine. It could be either any appropriate developer workstation or any other workstation you usually use for merging your team work. This scenario typically applies to large teams, where the package and project owners are responsible for the integrity of the program elements, and the administrator is not necessarily involved in the development.

In Figure 47, A and B are indications of alternatives that are available depending on whether or not you want to create an extra repository containing only the program elements you want to migrate. If you create a separate repository to import from, you have an additional backup copy of all of your work done with VisualAge for Java Enterprise Version 1.0.

Detailed migration steps are provided in Table 12.

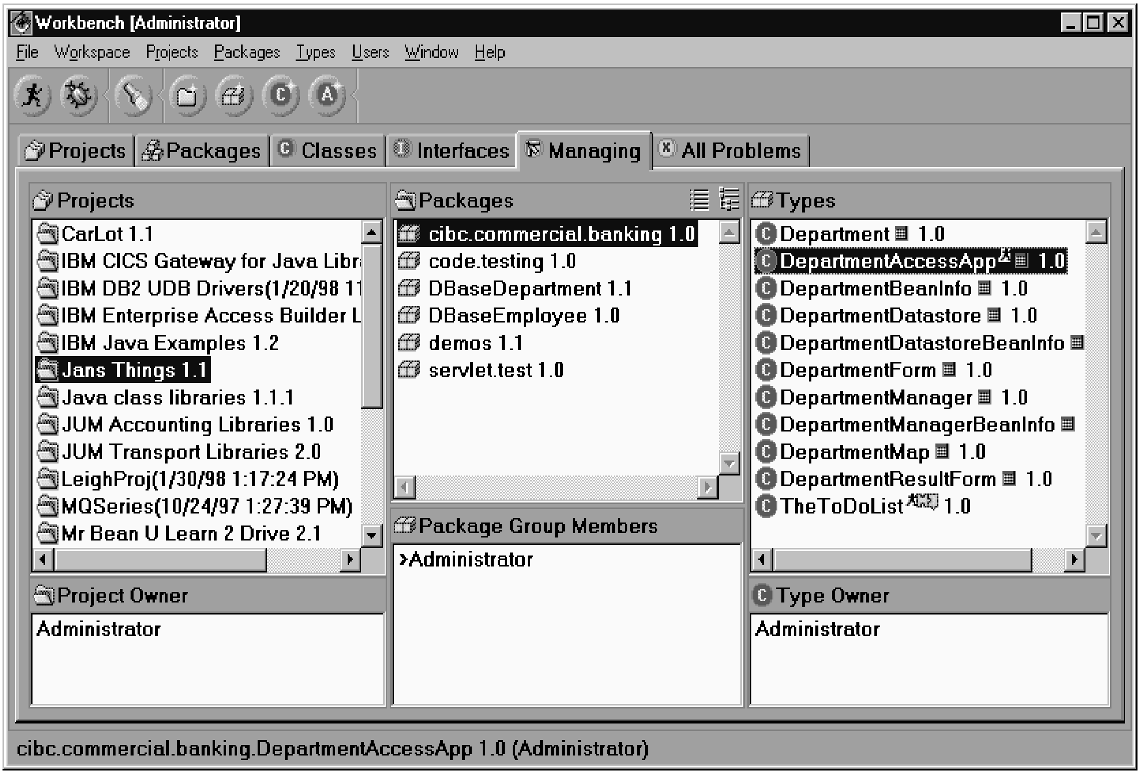
Table 12. Migration Steps from VisualAge for Java Enterprise Version 1.0

Step	Description
1	Version and release all program elements you want to migrate into the team environment. Merge all work of existing repositories into one repository, using the process you have already established in your team. Make a backup copy of the repository.
2	Copy your Version 1.0 repository to a another directory or another drive to ensure that it will not be deleted accidentally when you install the new version of VisualAge for Java Enterprise. It must be a directory that you can access afterwards from the newly installed VisualAge for Java Enterprise version.
3	Uninstall Version 1.0 of VisualAge for Java Enterprise and install the new team-enabled version. Note: Uninstall is necessary only when installing the new version on the same machine.
4	Start the new version of VisualAge for Java Enterprise. Set up the team environment and define all users who will work with the new repository. Note: You can start VisualAge for Java Enterprise with or without EMSRV running. This does not affect the next steps.
5	Use the Import option to connect to your old Version 1.0 repository and import all projects and packages into your new repository.
6	All imported program elements are owned by the administrator by default. Add the appropriate user names to the different package groups. Assign ownership to projects and packages and types.
7	If necessary, repeat steps 5 and 6 for other repositories your team used before.

Some steps may vary depending on the machine on which the new version is installed. Steps 1 and 2 will be different if you decide to export your work from the old repository into another repository and import from that repository.

Whether you export from a stand-alone repository into a multiuser repository, or import into your team repository from a stand-alone repository, the resulting ownership of your program elements is the same. They will be owned by the administrator.

Figure 48 shows a project after it was imported from a VisualAge for Java Enterprise Version 1.0 repository. We are browsing this project from the *Managing* pane of the team-enabled version. Note the ownership of all program elements as shown in the bottom three panes of this browser. Also notice that the version information is retained with the elements as they are migrated.


Workbench [Administrator]
File Workspace Projects Packages Types Users Window Help
Projects Packages Classes Interfaces Managing All Problems

Projects

- CarLot 1.1
- IBM CICS Gateway for Java Libr
- IBM DB2 UDB Drivers(1/20/98 11
- IBM Enterprise Access Builder L
- IBM Java Examples 1.2
- Jans Things 1.1
- Java class libraries 1.1.1
- JUM Accounting Libraries 1.0
- JUM Transport Libraries 2.0
- LeighProj(1/30/98 1:17:24 PM)
- MQSeries(10/24/97 1:27:39 PM)
- Mr Bean U Learn 2 Drive 2.1

Packages

- cibc.commercial.banking 1.0
- code.testing 1.0
- DBaseDepartment 1.1
- DBaseEmployee 1.0
- demos 1.1
- servlet.test 1.0

Types

- Department 1.0
- DepartmentAccessApp 1.0
- DepartmentBeanInfo 1.0
- DepartmentDatastore 1.0
- DepartmentDatastoreBeanInfo
- DepartmentForm 1.0
- DepartmentManager 1.0
- DepartmentManagerBeanInfo
- DepartmentMap 1.0
- DepartmentResultForm 1.0
- TheToDoList 1.0

Project Owner

Administrator

Package Group Members

>Administrator

Type Owner

Administrator

cibc.commercial.banking.DepartmentAccessApp 1.0 (Administrator)

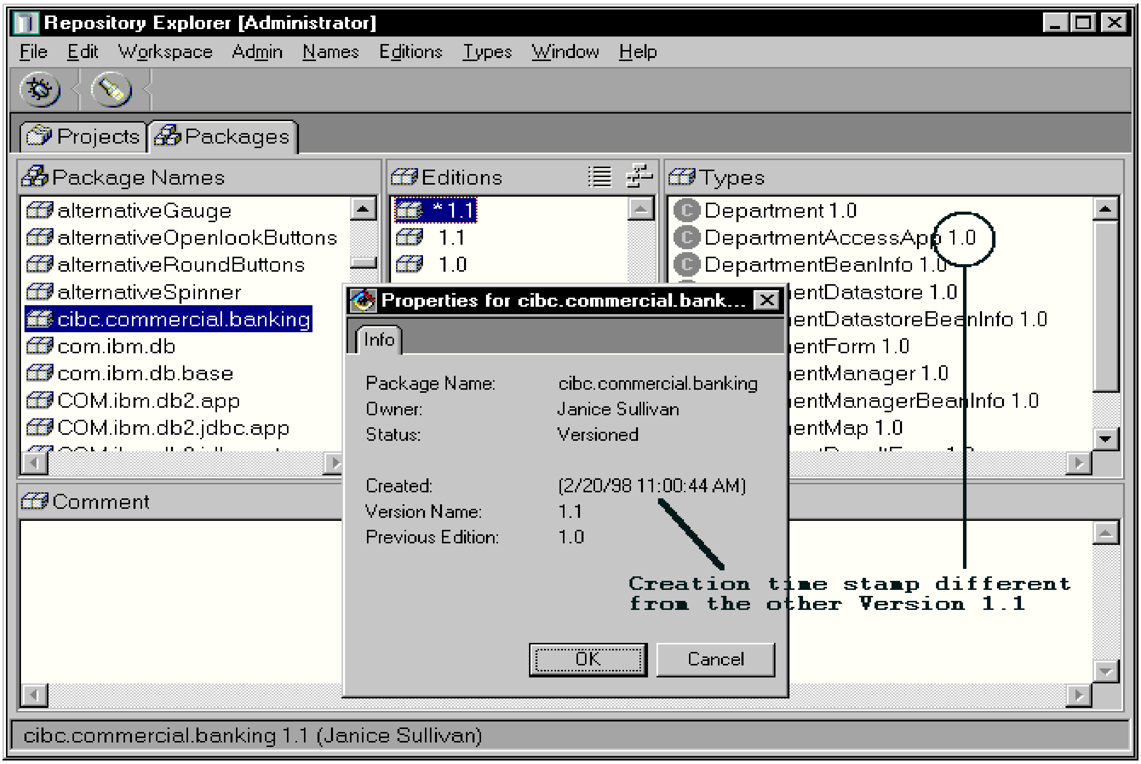
Figure 48. Project Imported from a Version 1.0 Repository

Considerations

If you work in a small team, you may consider importing from each of the developer workspaces separately and consolidating all work directly in the new team repository. This is a valid procedure if the administrator knows which edition of which program elements make up the new baseline.

You may find when you import projects or packages from different developer workspaces that they have the same version name. As long as the creation dates of these versions are different, the VisualAge for Java Enterprise server loads multiple versions with the same name into the repository without overriding existing code. As a result you can have two (or more) versions of a program element with the same name but different creation dates. Each of these program elements is handled as a different edition.

Figure 49 and Figure 50 show us a package that first existed in the repository as Version 1.1 and then was imported again having the same version name but different class versions.

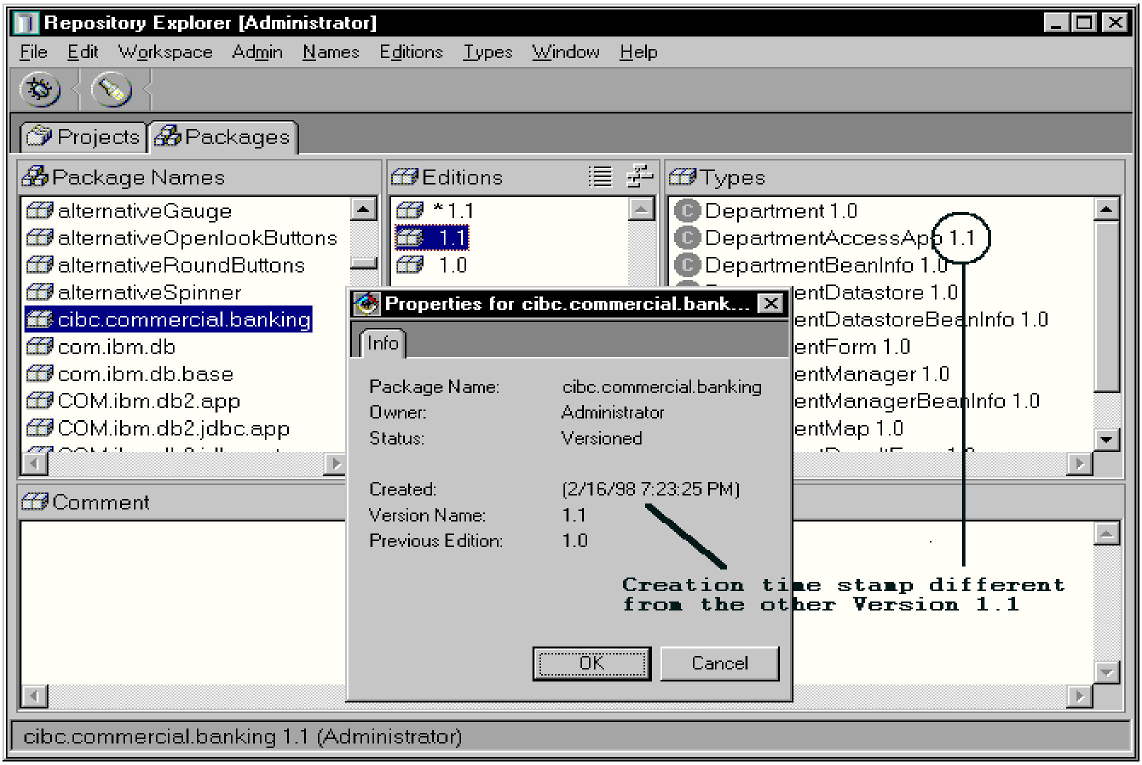


Package Name: cibc.commercial.banking
Owner: Janice Sullivan
Status: Versioned

Created: (2/20/98 11:00:44 AM)
Version Name: 1.1
Previous Edition: 1.0

Creation time stamp different from the other Version 1.1

Figure 49. Package Versions with the Same Name



Package Name: cibc.commercial.banking
Owner: Administrator
Status: Versioned

Created: (2/16/98 7:23:25 PM)
Version Name: 1.1
Previous Edition: 1.0

Creation time stamp different from the other Version 1.1

Figure 50. Package Version 1.1 Imported with the Same Version Name

When you notice two versions of a package with the same version name, open the properties of each of the versions, as we have done here, and you will see that the creation time stamp is different. Of course only one of these packages can exist in any workspace at any one time.

E Special Notices

This publication is intended to help VisualAge for Java developers work in a team using the team support provided by VisualAge for Java Enterprise. The information in this publication is not intended as the specification of any programming interfaces that are provided by VisualAge for Java Enterprise. See the PUBLICATIONS section of the IBM Programming Announcement for VisualAge for Java for more information about what publications are considered to be product documentation.

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, 500 Columbus Avenue, Thornwood, NY 10594 USA.

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 information about non-IBM ("vendor") products in this manual has been supplied by the vendor and IBM assumes no responsibility for its accuracy or completeness. 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.

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

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

AIX	IBM
OS/2	VisualAge

The following terms are trademarks of other companies:

C-bus is a trademark of Corollary, Inc.

Java and HotJava are trademarks of Sun Microsystems, Incorporated.

Microsoft, Windows, Windows NT, and the Windows 95 logo are trademarks or registered trademarks of Microsoft Corporation.

PC Direct is a trademark of Ziff Communications Company and is used by IBM Corporation under license.

Pentium, MMX, ProShare, LANDesk, and ActionMedia are trademarks or registered trademarks of Intel Corporation in the U.S. and other countries.

UNIX is a registered trademark in the United States and other countries licensed exclusively through X/Open Company Limited.

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

F Related Publications

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

International Technical Support Organization Publications

For information about ordering these ITSO publications, see "How To Get ITSO Redbooks" on page 157.

- ❑ *Programming with VisualAge for Java*, published by Prentice Hall, ISBN 0-13-911371-1, 1998 (IBM order number SR23-8478)
- ❑ *Application Development with VisualAge for Java Enterprise*, SG24-5081
- ❑ *Creating Java Applications with NetRexx*, SG24-2216
- ❑ *Unlimited Enterprise Access with Java and VisualAge Generator*, SG24-5246
- ❑ *VisualAge Generator Client/Server Communications*, SG24-4237
- ❑ *VisualAge Generator Version 3.0 System Development Guide*, SG24-4230
- ❑ *From Client/Server to Network Computing, A Migration to Java*, SG24-2247
- ❑ *CBConnector Overview*, SG24-2022
- ❑ *CBConnector Cookbook Volume 1*, SG24-2033
- ❑ *Connecting the Enterprise to the Internet with MQSeries and VisualAge for Java*, SG24-2144
- ❑ *Factoring JavaBeans in the Enterprise*, SG24-5051
- ❑ *JavaBeans by Example*: *Cooking with Beans in the Enterprise*, SG24-2035, published by Prentice Hall, 1997
- ❑ *Java Network Security*, SG24-2109, published by Prentice Hall, 1998
- ❑ *Building AS/400 Applications with Java*, SG24-2163
- ❑ *Accessing the AS/400 System with Java*, SG24-2152
- ❑ *World Wide Web Programming: VisualAge for C++ and Smalltalk*, SG24-4734, published by Prentice Hall, 1997
- ❑ *Programming with VisualAge for C++ for Windows*, SG24-4782, published by Prentice Hall, 1997
- ❑ *Object-Oriented Application Development with VisualAge for C++ for OS/2*, SG24-2593, published by Prentice Hall, 1996
- ❑ *VisualAge and Transaction Processing in a Client/Server Environment*, GG24-4487, published by Prentice Hall, 1995

Redbooks on CD-ROMs

Redbooks are also available on CD-ROMs. **Order a subscription** and receive updates 2-4 times a year at significant savings.

CD-ROM Title	**Subscription Number**	**Collection Kit Number**
System/390 Redbooks Collection	SBOF-7201	SK2T-2177
Networking and Systems Management Redbooks Collection	SBOF-7370	SK2T-6022
Transaction Processing and Data Management Redbook	SBOF-7240	SK2T-8038
Lotus Redbooks Collection	SBOF-6899	SK2T-8039
Tivoli Redbooks Collection	SBOF-6898	SK2T-8044
AS/400 Redbooks Collection	SBOF-7270	SK2T-2849
RS/6000 Redbooks Collection (HTML, BkMgr)	SBOF-7230	SK2T-8040
RS/6000 Redbooks Collection (PostScript)	SBOF-7205	SK2T-8041
RS/6000 Redbooks Collection (PDF Format)	SBOF-8700	SK2T-8043
Application Development Redbooks Collection	SBOF-7290	SK2T-8037

Other Publications

This publication is also relevant as a further information source:

❑ *Developing JavaBeans Using VisualAge for Java*, Dale Nilsson and Peter Jakab, published by John Wiley, ISBN 0-471-29788-7

How To Get ITSO Redbooks

This section explains how both customers and IBM employees can find out about ITSO redbooks, CD-ROMs, workshops, and residencies. A form for ordering books and CD-ROMs is also provided.

This information was current at the time of publication, but is continually subject to change. The latest information may be found at `http://www.redbooks.ibm.com`.

How IBM Employees Can Get ITSO Redbooks

Employees may request ITSO deliverables (redbooks, BookManager BOOKs, and CD-ROMs) and information about redbooks, workshops, and residencies in the following ways:

- **PUBORDER** – to order hardcopies in United States
- **GOPHER link to the Internet** – type `GOPHER WTSCPOK.ITSO.IBM.COM`
- **Tools disks**

  To get LIST3820s of redbooks, type one of the following commands:

  ```
  TOOLS SENDTO EHONE4 TOOLS2 REDPRINT GET SG24xxxx PACKAGE
  TOOLS SENDTO CANVM2 TOOLS REDPRINT GET SG24xxxx PACKAGE (Canadian users only)
  ```

  To get lists of redbooks:

  ```
  TOOLS SENDTO USDIST MKTTOOLS MKTTOOLS GET ITSOCAT TXT
  ```

  To register for information on workshops, residencies, and redbooks:

  ```
  TOOLS SENDTO WTSCPOK TOOLS ZDISK GET ITSOREGI 1996
  ```

  For a list of product area specialists in the ITSO:

  ```
  TOOLS SENDTO WTSCPOK TOOLS ZDISK GET ORGCARD PACKAGE
  ```

- **Redbooks Web Site on the World Wide Web**

  `http://w3.itso.ibm.com/redbooks`

- **IBM Direct Publications Catalog on the World Wide Web**

  `http://www.elink.ibmlink.ibm.com/pbl/pbl`

  IBM employees may obtain LIST3820s of redbooks from this page.

- **REDBOOKS category on INEWS**
- **Online** – send orders to: USIB6FPL at IBMMAIL or DKIBMBSH at IBMMAIL
- **Internet Listserver**

  With an Internet E-mail address, anyone can subscribe to an IBM Announcement Listserver. To initiate the service, send an E-mail note to `announce@webster.ibmlink.ibm.com` with the keyword `subscribe` in the body of the note (leave the subject line blank). A category form and detailed instructions will be sent to you.

How Customers Can Get ITSO Redbooks

Customers may request ITSO deliverables (redbooks, BookManager BOOKs, and CD-ROMs) and information about redbooks, workshops, and residencies in the following ways:

- **Online Orders** (Do not send credit card information over the Internet) – send orders to:

	IBMMAIL	**Internet**
In United States	usib6fpl at ibmmail	usib6fpl@ibmmail.com
In Canada	caibmbkz at ibmmail	lmannix@vnet.ibm.com
Outside North America	dkibmbsh at ibmmail	bookshop@dk.ibm.com

- **Telephone orders**

United States (toll free)	1-800-879-2755
Canada (toll free)	1-800-IBM-4YOU
Outside North America	(long distance charges apply)
(+45) 4810-1320 - Danish	(+45) 4810-1020 - German
(+45) 4810-1420 - Dutch	(+45) 4810-1620 - Italian
(+45) 4810-1540 - English	(+45) 4810-1270 - Norwegian
(+45) 4810-1670 - Finnish	(+45) 4810-1120 - Spanish
(+45) 4810-1220 - French	(+45) 4810-1170 - Swedish

- **Mail Orders** – send orders to:

IBM Publications
Publications Customer Support
P.O. Box 29570
Raleigh, NC 27626-0570
USA

IBM Publications
144-4th Avenue, S.W.
Calgary, Alberta T2P 3N5
Canada

IBM Direct Services
Sortemosevej 21
DK-3450 Allerød
Denmark

- **Fax** – send orders to:

United States (toll free)	1-800-445-9269
Canada	1-800-267-4455
Outside North America	(+45) 48 14 2207 (long distance charge)

- **1-800-IBM-4FAX (United States)** or **(+1) 408 256 5422 (Outside USA)** – ask for:

Index # 4421 Abstracts of new redbooks
Index # 4422 IBM redbooks
Index # 4420 Redbooks for last six months

- **Direct Services** – send note to softwareshop@vnet.ibm.com
- **On the World Wide Web**

Redbooks Web Site	http://www.redbooks.ibm.com
IBM Direct Publications Catalog	http://www.elink.ibmlink.ibm.com/pbl/pbl

- **Internet Listserver**

With an Internet E-mail address, anyone can subscribe to an IBM Announcement Listserver. To initiate the service, send an E-mail note to announce@webster.ibmlink.ibm.com with the keyword subscribe in the body of the note (leave the subject line blank).

IBM Redbook Order Form

Please send me the following:

Title	Order	Quantit

First name Last name

Company

Address

City Postal code Country

Telephone number Telefax number VAT number

□ Invoice to customer number

□ Credit card number

Credit card expiration date Card issued to Signature

We accept American Express, Diners, Eurocard, Master Card, and Visa. Payment by credit card not available in all countries. Signature mandatory for credit card payment.

Index

ITSO Redbook Evaluation

VisualAge for Java Enterprise Version 2 Team Support
SG24-5245-00

Your feedback is very important to help us maintain the quality of ITSO redbooks. **Please complete this questionnaire and return it using one of the following methods:**

- Use the online evaluation form found at http://www.redbooks.ibm.com
- Fax this form to: USA International Access Code + 1 914 432 8264
- Send your comments in an Internet note to redbook@us.ibm.com

Please rate your overall satisfaction with this book using the scale:
(1 = very good, 2 = good, 3 = average, 4 = poor, 5 = very poor)

Overall Satisfaction __________

Please answer the following questions:

Was this redbook published in time for your needs? Yes___ No___

If no, please explain:

____

____

____

____

____

What other redbooks would you like to see published?

____

____

____

Comments/Suggestions: (THANK YOU FOR YOUR FEEDBACK!)

____

____

____

____

____

VisualAge for Java Enterprise Version 2 Team Support **SG24-5245-00**

.A.