IBM VisualAge® for Java™, Version 3.5

Getting Started

Note!

Before using this information and the product it supports, be sure to read the general information under Notices.

Edition Notice

This edition applies to Version 3.5 of IBM VisualAge for Java and to all subsequent releases and modifications until otherwise indicated in new editions.

Contents

Chapter 1. Preface

About this book

This document provides you with information about the following topics:

- Developing Java applications in a stand-alone and team environment
- Optimizing and debugging your Java applications for specific platforms
- Developing Java applications that access existing data and applications
- Creating server-oriented applications and applications that are available through the Internet

This book will familiarize you with the VisualAge for Java components and the tasks that you can perform with them. You will learn how to develop an application using the Visual Composition Editor. We also guide you through the steps involved in preparing to work in a team environment.

Java fundaments are not covered in this book, and a basic knowledge of Java is necessary to to understand this book. You can find Java tutorials in the Education section of IBM's Developer Works Web site at http://www.ibm.com/developer/java/ and in the Documentation/Training section of the Sun Web site: http://java.sun.com/

Who should read this book

This document is aimed at developers who want to perform the following tasks:

- Develop Java applications in an integrated visual environment
- Work in a team environment or manage a project in a team environment
- Want to optimize application performance of their Java applications for one or more specific operating system
- Access existing enterprise applications and data outside their Java applications
- Develop server-oriented, multi-tier applications

Information from this book can also be found in the Getting Started section of the online help.

Related information

For more detailed information and tutorials and samples for components, refer to the online help that is available from the **Help** menu of any window in VisualAge for Java. For information about the various search options available, click “Search options” which is next to the search field.

The online help is organized into several categories, all of which are directly accessible both from the home page of the help and from any of the content pages:

- **Topics** lets you view the help topics organized by type of information (concepts, tasks, reference materials, samples):
- **Components** contains the same information as **Topics**, organized by product features

- **PDF Documents.** Some of the online help has been grouped into PDF documents which you can view and print using Adobe Acrobat Reader (available from http://www.adobe.com/)
- **Web Resources** contains links to Java-related resources that are available on the Internet.

README files and release notes
Release notes are component-specific and contain information about component prerequisites, limitations and known problems. They also contain general hints and tips about the component. Please refer to the README file if you want information about general product limitations and known problems.

Installation and Migration guide
Information on how to install VisualAge for Java and migrate from a previous version of VisualAge for Java can be found in the Installation and Migration guide, which is on the product CD directory. Please refer to the README file for the exact location of the guide.

How to find updates to software and documentation
Updates can be found on the the VisualAge Developer Domain (VADD) Web site http://www.ibm.com/software/vadd. This Web site offers information about tools, education, hints, and tips, along with easy access to support and product updates for VisualAge for Java.

As well, you can access VisualAge for Java **scenario-based documentation** from this web site. The scenario documentation describes how two or more Visual for Java features or products can used together to develop a Web-based, end-to-end, client/server application that meets a business need; for example: using ET/390 to develop a client application that can access CICS® over IIOP.

The product home page for VisualAge for Java is at http://www.ibm.com/software/ad/vajava

PDF index

The PDF files listed below have been prepared directly from the HTML online help documentation for VisualAge for Java, Version 3.5. For this reason, the PDF documents resemble a collection of online help topics rather than a printed book. These PDF documents are provided to help you print a collection of online help topics.

You can view or print the following PDF files using the Adobe Acrobat Reader, which is available from http://www.adobe.com.

Getting started
This book (getstart.pdf) provides a tutorial on the VisualAge for Java IDE and Visual Composition Editor. It is the same book that is packaged with the VisualAge for Java software.

Developing programs

- Integrated Development Environment basics (ide.pdf) is a VisualAge for Java IDE primer.
- Visual Composition (jvbpdf.pdf) provides instructions on how to do visual programming with the beans shipped in VisualAge for Java.
- Distributed Debugger (debugwks.pdf) describes how to debug programs on Windows®.

Working in a team environment

- [ENTERPRISE] Team Programming (team.pdf) decribes information to develop and manage Java programs in a team environment.
- Interface to external version control systems (scci.pdf) describes how to connect to source code in an external source code management system.

Accessing relational data

- Data Access Beans (dataacc.pdf) describes how to access relational data through beans that support SQL.
- SQLJ Tool (sqlj.pdf) describes how to access databases using embedded SQL in your Java source code.
- DB2® Stored Procedure Builder (stproc.pdf) provides instructions to create stored procedures for implementation on a database server.

[ENTERPRISE] Developing EJB components

EJB Development Environment (ejb.pdf) describes how to develop and test Enterprise JavaBeans™ components.

[ENTERPRISE] Adding persistence

Building Applications with Persistence Support (pbinfo.pdf) describes how to add scalable persistence support to object models.

[ENTERPRISE] Developing servlets

- JSP/Servlet Development Environment (jsp.pdf) describes how to use JavaServer Pages technology and servlets for Web development.
- Servlet Builder (sbinfo.pdf) describes how to create servlets with visual design tools.

[ENTERPRISE] Accessing transaction systems

- Enterprise Access Builder for Transactions (eab.pdf) describes how to access transaction systems (for example, CICS, IMS™, or MQSeries).
- The Record Framework (recframe.pdf) describes a framework for record data conversion used by the Enterprise Access Builder
- IMS Connector for Java (imstoc.pdf) describes how to create Java application programs or servlets that access IMS transactions using the IMS TCP/IP OTMA Connection Connector for Java.

[ENTERPRISE] Accessing SAP R/3 systems

Connector and Access Builder for SAP R/3 (sap.pdf) describes how to create Java programs that access SAP R/3 systems.

[ENTERPRISE] Targeting programs for AS/400®

- ET/400 (et400.pdf) describes how to optimize Java code for high performance on AS/400.
- AS/400 Debugging (dbg400.pdf) describes how to debug programs targeted for AS/400.

[ENTERPRISE] Targeting programs for OS/390®

- ET/390 (et390.pdf) describes information for optimizing Java code for high performance on OS/390.
- Performance Analyzer for OS/390 (pa390.pdf) Programs describes how to analyze the performance of Java programs running on OS/390.

- OS/390 Debugging (debug390.pdf) describes how to debug programs targeted for OS/390.

[ENTERPRISE] Doing CORBA development

IDL Development Environment (idl.pdf) describes how to manage IDL source code and generated Java code.

Accessing Domino databases

- Domino AgentRunner (notes.pdf) describes how to build, run and debug Domino agents in VisualAge for Java.
- [ENTERPRISE] Domino Access Builder (domino.pdf) describes how to create beans to access databases or services on Lotus® Domino servers.

[ENTERPRISE] Accessing C++ services

C++ Access Builder (j2cpp.pdf) describes how to create Java code that accesses C++ programs.

Accessing tool integrator APIs

- Remote Access to Tool API (remacc.pdf) describes how to allow client applications to access the Tool Integrator APIs from outside the IDE, using the HTTP protocol.
- Tool Integrators for ISVs (toolint.pdf) describes how to Integrate file-based code with the IDE and programmatically manipulate code within the IDE (advanced topic).

Developing XML

- XML Generator (xmlgen.pdf) describes how to edit a DTD and generate sample XML documents based on that DTD.
- XML for Java parser (xml4j.pdf) describes how to use a Java parser to read and write XML data.
- XML toolkit (xmi.pdf) describes how to visually model Java constructs.

Web resources

The VisualAge for Java online information provides information about the various components that make up VisualAge for Java. The World Wide Web contains resources that complement and extend the online information. This page provides links to those Web resources. You must have a connection to the World Wide Web to access these links.

The Web sites listed on this page were available at the time of writing. With each release of the online information, we will check to ensure the links are still current. Between releases, however, the links to these sites could change.

IBM resources

- **VisualAge Developer Domain**. This is the premier IBM site for Java developers using VisualAge products. You will find links to downloads, demos, tutorials, samples, documentation, and more.
  - **Library**. The Library page in the VisualAge Developer Domain site contains links to technical articles, tutorials, white papers, books, magazines, IBM Redbooks, and FAQs about a wide range of Java programming topics.
- **VisualAge for Java.** This is the primary site for information about the VisualAge for Java products, including the Entry, Professional, and Enterprise Editions. You will find links to downloads, case studies, events, education, and support (including FAQs).

- **Web Application Servers.** This site provides information about the WebSphere™ family of products, including WebSphere Studio and WebSphere Application Server, which can help you develop and manage high-performance Web sites.
- **VisualAge Partner Catalog**. This site provides a rich collection of partner solutions for VisualAge products, including palette components (such as JavaBeans, parts, and widgets), applications, applets, and complementary tools.
- **Java Technology Zone.** This is IBM's site for Java resources such as news, code, tools, support programs, education, and events.
- **jCentral.** This is a webcrawler and search engine designed specifically to help you locate Java applets, beans, source code, newsgroup articles, and more.
- **IBM Software.** This is IBM's home page for software. It provides general information about IBM software and has links to various software products, including VisualAge for Java.

Non-IBM resources

- **Java Technology Home Page (java.sun.com).** This site is where Sun Microsystems provides the latest information about Java technology, including downloads, documentation, specifications, and development kits.
- **Java Users Group.** This site provides links to user groups all over the world where you can talk about Java development.
- **Java Newsgroups.** This site provides links to comp.java.lang newsgroups where you can discuss various Java topics.
- **FreewareJava.com.** This site provides links to free Java resources on the Web, including applets, tutorials, references, articles, and more.
- **Java Directory.** Developer.com's Java Directory provides links to free source code, and numerous sources of information about Java.
- **VisualAge Bookstore.** This site provides an extensive collection of VisualAge books, training, and Web resources, including IBM Redbooks.
- **Java magazines.** The following online magazines cover the world of Java development, and include news, technical articles, how-tos, and product reviews:
  - Servlet Central
  - Java Developers Journal
  - JavaPro
  - Java Report Online
  - JavaWorld
  - Focus on Java
  - Dr. Dobb's Journal

Reader comments

IBM welcomes your comments about the VisualAge for Java online information. You can send your comments using any one of the following methods:

- Electronically to the network ID listed below. Be sure to include your entire network address if you want a reply.
  - Internet: torrcf@ca.ibm.com
  - IBMLink: toribm(torrcf)
- By FAX using one of the following numbers:
  - United States and Canada: 416-448-6161
  - Other countries: (+1)-416-448-6161
- By mail to the following address:

IBM Canada Ltd. Laboratory
Information Development
2G/KB7/1150/TOR
1150 Eglinton Avenue East
North York, Ontario, Canada
M3C 1H7

Chapter 2. Introduction

About VisualAge for Java

VisualAge for Java is an integrated visual environment that supports the complete cycle of Java program development. VisualAge for Java gives you everything you need to perform the development tasks described below.

Rapid application development
You can use VisualAge for Java's visual programming features to quickly develop Java applets and applications. In the Visual Composition Editor, you point and click to:

- Design the user interface for your program
- Specify the behavior of the user interface elements
- Define the relationship between the user interface and the rest of your program

VisualAge for Java generates the Java code to implement what you design in the Visual Composition Editor. In many cases you can design and run complete programs without writing any Java code.

VisualAge for Java also gives you *SmartGuides* (wizards) to lead you quickly through many tasks such as creating new applets, packages, or classes.

Incremental compilation
The VisualAge for Java Integrated Development Environment (IDE) automatically compiles Java source code into Java bytecode. When source code is imported into the workspace (from .java files) or added from the repository, it is compiled and analyzed with respect to the existing contents of the workspace.

When you change, delete, move, copy, or rename program elements, the affected code is automatically recompiled to flag any problems.

If you introduce an error, the IDE warns you and gives you the option of fixing the problem immediately, or of adding the problem to the All Problems page and fixing it later.

Repository-based development
VisualAge for Java has a sophisticated code management system that makes it easy for you to maintain multiple editions of programs. When you want to freeze the state of your code at any point, you can *version* an edition. This marks the particular edition as read-only and enables you to give it a name. This gives you a way to preserve snapshots of significant checkpoints in a development cycle.

Create industrial-strength Java programs
With VisualAge for Java you can develop very robust code. Specifically, you can:

- Build, modify, and use JavaBeans
- Browse your code at the level of project, package, class, or method
- Use the integrated visual debugger to examine and update code while it is running
- Use the distributed debugger to debug Java applications that are developed outside the IDE

- [ENTERPRISE] Write applications that comply with Sun Microsystems Enterprise JavaBeans (EJB) specifications

Program development

The **Integrated Development Environment** (IDE) is a set of windows that provide access to development tools. These windows include the Workbench, Repository Explorer, Log, Console, and Debugger. You can use the IDE to create Java applets, which run in web browsers, and standalone Java applications.

The **Visual Composition Editor** can be used to create graphical user interfaces from prefabricated beans, and to define relationships (called connections) between beans. This process of creating object-oriented programs by manipulating graphical representations of components is called visual programming.

The**JSP/Servlet Development Environment** enables you to develop, run, and test JSP files and servlets.

The **WebSphere Test Environment** (installed with the JSP/Servlet Development Environment**)** provides server run-time support for locally testing and debugging JSP files and servlets. JSP files and servlets that run successfully in the WebSphere Test Environment will also run successfully in a WebSphere Application Server production environment.

The **Create Servlet SmartGuide** enables you to create servlets. When you are developing servlets with the SmartGuide you can use JSP files that inherit from the PageListServlet class. As well, you can also work with existing beans such as an EAB Command or an EJB access bean.

[ENTERPRISE] The **XMI Toolkit** provides an integration bridge between VisualAge for Java and the Rational Rose modeling tool. It enables you to generate VisualAge for Java code from Rational Rose models, and generate Rational Rose models from VisualAge for Java code. This facilitates the rapid transformation of business models.

[ENTERPRISE] The **XML Generator** is a Java program you can use to edit a DTD and generate sample XML documents based on that DTD. Creating sample documents helps you check to see if the DTD is working properly, and get an idea of what kind of XML documents are generated by the DTD.

The **XML Parser for Java** is a high performance, modular XML parser written in 100% Java. This parser makes it easy to give an application the ability to read and write XML data. A single JAR file provides classes for parsing, generating, manipulating, and validating XML documents.

[ENTERPRISE] The **EJB Development Environment** enables you to develop beans that implement Sun Microsystems' Enterprise JavaBeans (EJB) programming specifications. The EJB Development Environment provides all of the necessary run-time support for the IBM WebSphere Application Server. A new incremental consistency checker ensures that enterprise beans conform to the EJB programming specification and indicates whether changes are needed to fix inconsistencies.

[ENTERPRISE] The **IDL Development Environment** provides an integrated interface definition language (IDL) and Java development environment that works with a user-specified IDL-to-Java compiler. You can work with both your IDL source and the generated Java code in one convenient browser page.

[ENTERPRISE] The **Persistence Builder** enables you to map objects and relationships between objects to information stored in relational databases. It also provides linkages to the mapping between Enterprise JavaBeans and relational data.

[ENTERPRISE] The **C++ Access Builder** provides access from Java applets or stand-alone Java applications to services written in C++. You can use it to generate beans and C++ wrappers that let your Java programs access C++.

Database access

Data Access Beans are beans provided with VisualAge for Java for developing programs that access relational databases.

The **DB2 Stored Procedure Builder** enables you to write Java programs that include stored procedures that run on a DB2 server.

SQLJ support provides you with a standard way to embed SQL statements in Java programs. The SQLJ standard has three components: embedded SQLJ, a translator, and a run time environment.

Access to transaction servers

[ENTERPRISE] The **Enterprise Access Builder for Transactions (EAB)** helps developers to quickly and easily extend existing applications to e-business. It consists of frameworks and tools which allow you to access the function and data assets of your existing transaction-based systems. It provides a standard access builder interface for *connectors* that adhere to IBM's Common Connector Framework (CCF). Connectors are provided for CICS (both ECI and EPI), Encina DE-Light, IMS TOC, MQSeries®, Host-on-Demand (HOD), and SAP R/3.

Access to application servers

[ENTERPRISE] The **Domino AgentRunner** helps you build, run, and debug Domino agents in the VisualAge for Java IDE.

[ENTERPRISE] The **Domino Access Builder** provides beans and SmartGuides (wizards) that allow you to create Java applications that access databases and services located on a Domino server or on a Notes™ client. The generic beans include wrapper classes for databases, forms, views, and other Domino design elements.

[ENTERPRISE] The **Access Builder for SAP R/3** is a complete toolkit for creating Java applications, applets, and beans that access an SAP R/3 system. The Access Builder for SAP R/3 helps you to retrieve complete meta-information within the R/3 system, keep it locally for multiple R/3 systems, access without an R/3 connection, and search for business objects. It includes a connector for SAP R/3 that generates EJB proxies for business objects.

Optimization and debugging

The **Distributed Debugger** is a client/server application that enables you to detect and diagnose errors in your programs. You can also use it to debug Java applications that are developed outside the IDE. You can use the distributed debugger to debug applications that execute on the following platforms: Windows NT®, OS/2®, AIX®, AS/400, Solaris, and OS/390.

The **Integrated Debugger** enables you to debug applets and applications running in the IDE. In the debugger, you can view running threads, suspend them, and inspect their visible variable values.

[ENTERPRISE] The **Enterprise Toolkit for AS/400** (ET/400) allows you to develop applications in Java for the AS/400 platform. You can use ET/400 to export Java class and source files to the AS/400, and compile Java code optimized for the AS/400, and to run and debug AS/400 Java applications from the VisualAge for Java IDE.

[ENTERPRISE] The **Enterprise Toolkit for OS/390** (ET/390) enables you to code Java programs at your workstation, then export your code to run in a remote OS/390 Java Virtual Machine (JVM). You can also use ET/390 to bind Java bytecode into optimized code that runs in the OS/390 shell or CICS/ESA® environment. And you can debug your Java applications running on the OS/390 system from the IDE.

ET/390 also includes a Performance Analyzer to help you improve the performance of your Java programs. The Performance Analyzer traces function calls and returns, and collects timing data and call counts for each function called. The Performance Analyzer has two components: the host component, which you use to create a trace of your Java program's execution in the OS/390 UNIX® environment, and the workstation component, which you use to analyze the trace file that you created on the host.

Team development

[ENTERPRISE] **Team Programming** support. VisualAge for Java provides an integrated team development environment that is based on a shared source code repository.

External Version Control enables you to connect to external source code management (SCM) providers such as ClearCase, PVCS Version Manager, TeamConnection™, and SourceSafe from VisualAge for Java.With this tool, you can add classes to your SCM provider, check classes in and out of the SCM system, and import the most recently checked-in version of a class from the SCM system.

Other tools

[ENTERPRISE] The **Tool Integrator API** enables you to integrate your own tools into the VisualAge for Java IDE. From a single class execution in your tool, you can use the API to perform such IDE functions as browsing the workspace, importing and exporting files, or launching the IDE browsers.

[ENTERPRISE] The **Remote Access to Tool API** provides a way for client applications to access Tool Integrator APIs from outside the VisualAge for Java

IDE. Remote Access provides an HTTP daemon that supports tool servlets. A servlet is code you write that invokes the methods of the Tool Integrator API to perform a particular task.

[ENTERPRISE] **Tivoli® Connection** helps you create business-critical distributed Java applications that can be managed by Tivoli's enterprise management software. Tivoli Connection allows you to generate Tivoli events and to interface to the Tivoli Enterprise Console™. Tivoli provides an application management solution that covers the life cycle of an application, from deployment to monitoring and administration.

What's new in VisualAge for Java, Version 3.5

New features
The following features have been added since Version 3.0. For more information about each feature, refer to the list of component summaries in this *Getting Started* guide.

- [ENTERPRISE] XMI Toolkit
- [ENTERPRISE] XML Generator
- XML Parser for Java
- Servlet SmartGuide
- External Version Control

Enhancements to existing features
VisualAge for Java, Version 3.5, also offers enhancements to the following components:

Integrated Development Environment
There are many new and enhanced functions in the Integrated Development Environment (IDE). A new Solutions Browser has been added. A Solution is a grouping of projects. Solutions themselves do not appear in the Workbench, they are a way of organizing your work.

Inner classes and their methods will now appear in the general hierarchy views in the Workbench browsers. Search offers new filtering capabilities, and several new code formatting options, such as setting a maximum line length, and automatically removing blank lines are now available.

You can now version project resources, and configure breakpoints as soon as you set them. A new view in the Workbench, Class Source View, enables you to view a complete class, including all of its methods. In this view, you can see and edit the class definition and all of the methods of a class at the same time.

If your classes contain obsolete class or package references due to renaming, you can repair breakage with the Fix/Migrate SmartGuide. You will find this tool especially useful as you move classes to the J2SDK v 1.2.2. You must use this tool for repairing visual composites, even if you make corrections by hand.

JDK
The JDK has been updated to J2SDK v1.2.2. Please refer to the Documentation release notes (Getting Started Guide section) for the PTF level.

[ENTERPRISE] Team Development

EMSRV is now supported on Red Hat Linux and Windows 2000. You can now version and release your project resource files.

Websphere Test Environment

The WebSphere Test Environment Control Center provides a central location for you to start, stop, and configure WebSphere Test Environment services:

- **JSP Execution Monitor,** which helps you to monitor the execution of JSP source, along with the generated Java servlets and HTML source.
- **Persistent Name Server,** which you can use to work with EJB beans or DataSource objects.
- **Websphere Test Environment Servlet Engine,** which enables you to run multiple Web applications
- **DataSource objects.** You can use DataSource objects to manage a collection of connections to a database.

[ENTERPRISE] EJB Development Environment

The EJB Development Environment now supports Sybase data stores. As well, support for deployment of enterprise beans to Component Broker has been expanded.

The EJB test client has a new user interface and it now dynamically adapts to EJB interfaces. You no longer need to generate the test client code.

Server support has been added for Sybase and DB2/390 databases.

The name server function has been moved from the EJB Server Configuration browser to the new WebSphere Test Environment Control Centre.

[ENTERPRISE] Enterprise Toolkit for AS/400

The following AS/400-specific beans have been added to the Enterprise Toolkit for AS/400.

DFU beans

These beans extend the support of code to access AS/400 database files. You can use them to map GUI forms, tables, and lists to AS/400 databases, and manipulate database records.

Object List beans

These beans provide a method for accessing AS/400 object names, and allow you to set listing properties for selecting the desired type of object list.

JFormatted beans

The JFormatted beans include a set of utility classes that extend the support of code to convert AS/400 fields and attributes and to provide editcode, editword, formatting, and verification capabilities.

The following new classes are offered with the AS/400 toolbox:

- *Program Call Markup Language* (PCML) classes efficiently call AS/400 programs. You can call a program by writing a PCML script and then loading and deploying it from a Java program.
- *Security* classes make secured connections with the AS/400 and verify the identity of a user working on the AS/400 system.
- *HTML* classes quickly create HTML forms and tables.

- *Servlet* classes assist in retrieving and formatting data for use in Java servlets.

[ENTERPRISE] Distributed Debugger for AS/400

You can now debug and trace servlets, JSP files, and EJBs.

[ENTERPRISE] Enterprise Toolkit for ET/390

New functionality has been added to the ET/390 Performance Analyzer.

- Statistics have been added to the node list in the Dynamic Call Graph diagram. You can sort these statistics.
- Users now have more flexibility when pruning the Dynamic Call Graph diagram - the Show and Highlight options are now always enabled
- Percentage values now have two decimal places which improves their preciseness

[ENTERPRISE] Enterprise Access Builder

Several new visual tools and modifications to existing ones have been added to the Enterprise Access Builder. They extend its range of functions and usability.

3270 Importer

This tool imports 3270 terminal source code and outputs both record type and record classes, all in one operation. Formerly, several steps were needed. You can rename fields to more user friendly names and suppress the generation of get and set methods to reduce the generated code size.

Record Type Editor terminal pane

This pane displays a terminal view of the fields you are editing, showing where a field maps to a position on the screen.

IMS support in COBOL importer

Support for IMS conversational transactions have been added to the COBOL importer. Using this COBOL importer, other EAB tools, and the IMS TOC Connector, you can quickly build applications to access IMS conversational transactions.

Create Business Object SmartGuide

This visual tool lets you quickly create an EAB Business Object, which formerly had to be hand coded. Since Procedural Adaptor Objects extend EAB Business Objects, this tool will also allow you to quickly create PAOs for Component Broker applications accessing data and applications through CCF connectors.

Test Client

Prior to this tester, Commands and navigators could only be tested by writing an application to execute them. Now they can be tested anytime after they have been created. This tester lets you check that the returned values are what you expect, before you have done extensive development.

[ENTERPRISE] IMS Connector for Java

IMS Connector for Java now supports IMS conversations, allowing you to build Web applications that access IMS conversational transactions over the internet. Included are programming models and samples that guide you through the steps of creating the dynamic, client-to-program-to-client interactions of an IMS conversation. New Java classes have been added to support easy use of this feature.

[ENTERPRISE] Access Builder for SAP R/3
Support for OS/390 and Linux as deployment platforms have been added for the Access Builder and Connector for SAP R/3.

[ENTERPRISE] Persistence Builder
The Persistence Builder feature now supports Sybase data stores.

Installing VisualAge for Java

Instructions for installing VisualAge for Java can be found in the Installation and Migration guide, which is on the product CD. Please refer to the README file for the exact location of the guide.

When you install VisualAge for Java you can either perform a complete installation or a custom installation that lets you select which non-core components you want to install. In VisualAge for Java, a component is any tool with a distinct function. Some components, such as the IDE and the Visual Composition Editor, are always installed. Other components, such as the JSP Development Environment, are optionally installed.

[ENTERPRISE] You can also select to perform a ″Custom by Scenario″ installation. Custom by Scenario installations lets you customize the installation of VisualAge for Java based on different application scenarios. For example, you might want to develop applications that access relational data, or Web applications that run on a server. For each of these scenarios, you need to use different VisualAge for Java components. Instead of deciding which components to install for the applications you want to develop, all you need to do is select the application scenario.

Loading features into the workspace
After you have installed VisualAge for Java, you will have to load certain features, such as Data Access Beans, into the workspace before you can use them. A feature is one or more projects that relate to a particular sample or component of VisualAge for Java.

To load features into the workspace, follow these steps:

1. In the Workbench, select **File** > **Quick Start**. The Quick Start window opens.
2. Select **Features** > **Add Feature**.
3. Select the component you want to load into the workspace. You can also select to load other features such as libraries and samples into the workspace from this window.

To remove a feature from the workspace (that you have added using **Add Feature**), select **Delete Feature**.

Migrating from earlier releases of VisualAge for Java

Refer to Installation and Migration guide, which is on the product CD, for general instructions for migrating your data from a previous version of VisualAge for Java, and also for instructions on migrating applications from previous versions of VisualAge for Java.

Please refer to the README file for the exact location of the guide.

Chapter 3. Developing a simple applet

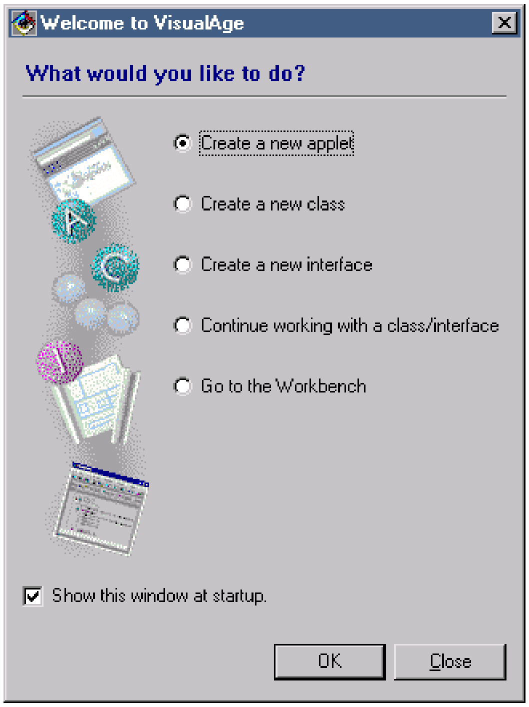
Starting VisualAge for Java

If you have not already installed VisualAge for Java, refer to the Installation and Migration guide, which is on the product CD. Please refer to the README file for the exact location of the guide.

To start VisualAge for Java on Windows 98, 2000, and Windows NT, select **Start > Programs > IBM VisualAge for Java for Windows > IBM VisualAge for Java.** For UNIX, run the command `vajide`.

[ENTERPRISE] When you first start VisualAge for Java, Enterprise Edition, you will be prompted to choose an *owner* for your workspace and a network name for the user called Administrator. For the purposes of this tutorial, you can select **Administrator** as your workspace owner. For a network name, use the name that identifies your computer to your network.

The VisualAge for Java Welcome window opens:



Welcome to VisualAge

What would you like to do?

● Create a new applet

○ Create a new class

○ Create a new interface

○ Continue working with a class/interface

○ Go to the Workbench

☑ Show this window at startup.

Select **Go to the Workbench** and click **OK**.

[ENTERPRISE] In the Enterprise Edition, the current workspace owner is shown on the title bar of the Workbench.

Exploring the IDE

VisualAge for Java is an integrated, visual environment that supports the complete cycle of Java program development. You can create Java applets, which run in Web browsers, and stand-alone Java applications.

Workspace and repository
All activity in VisualAge for Java is organized around a single *workspace*, which contains the source code for the Java programs that you are currently working on.

All code that you write and save in the workspace is also stored in a *repository*. Within the VisualAge for Java environment, you do not manipulate files. Instead, VisualAge for Java manages your code in a database of structured objects, that is, a repository.

All code that you create in VisualAge for Java is automatically stored in the repository. The repository also contains other packages provided by IBM, which you can add to the workspace if you need to use them.

In VisualAge for Java, you manage the changes that you make to a program element by creating *editions* of the program element. The workspace contains at most one edition of any program element, that is, the edition you are currently working on The repository, on the other hand, contains all editions of all program elements.

Working with program elements in the workspacefrom the Workbench. It gives you a view of all the program elements that are in the workspace and their unresolved problems.

From the Workbench, you can open other windows and browsers that help you with your tasks.

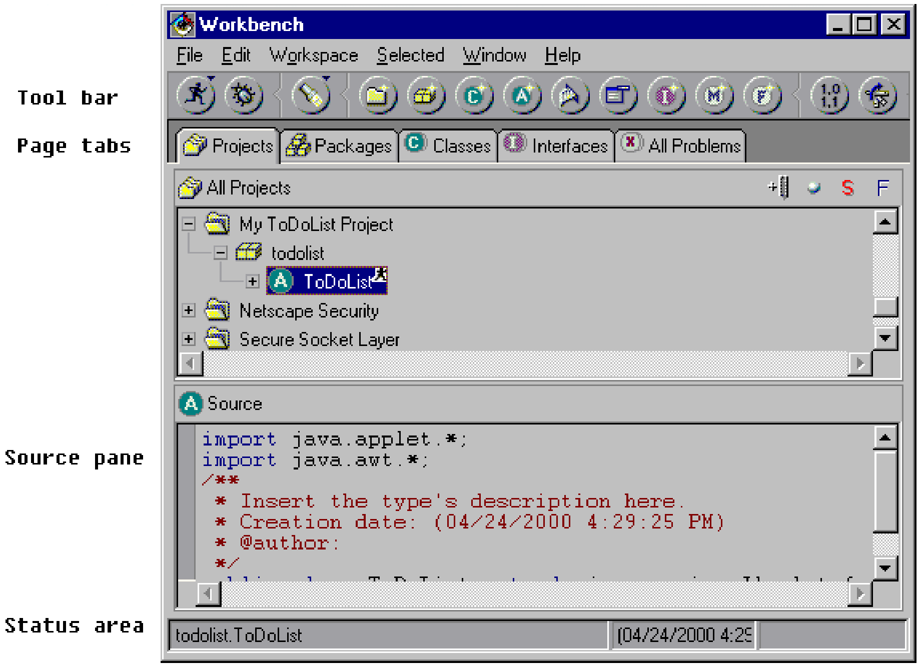
As shown in the diagram below, VisualAge for Java depicts code to you as a hierarchy of *program elements*, which you can expand and collapse by clicking the plus (+) and minus (-) symbols:

📁 project

📦 package

C class or I interface

● public, ▲ default, ◆ protected, ■ private methods



```
import java.applet.*;
import java.awt.*;
/**
 * Insert the type's description here.
 * Creation date: (04/24/2000 4:29:25 PM)
 * @author:
 */
```

todolist.ToDoList | (04/24/2000 4:2

The Workbench toolbar, which is located below the menu bar, provides you with easy access to the tasks you perform most frequently in the Workbench. Place your mouse pointer over each icon on the toolbar. A label will appear that identifies the tool.

In the IDE, you browse a program element by opening it. To open a program element, you can select **Open** from the **Selected** menu or from the program element's pop-up menu. To access a program elements' pop-up menu, select it and right-click. You can also open a program element by double-clicking it. This behavior can be changed in the **Options** window. You can access the **Options** window by selecting **Window > Options** in any window.

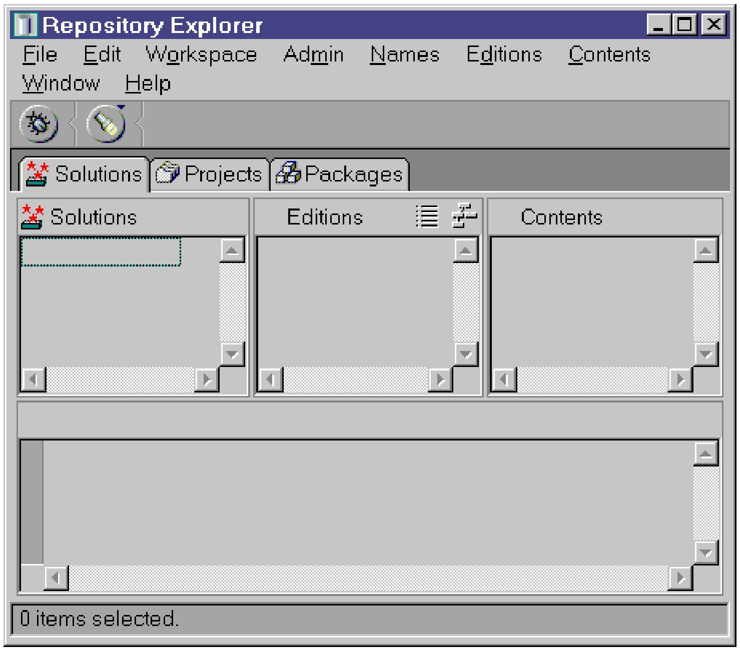
Follow these steps to become more familiar with the Workbench:

1. In the Projects pane, expand **Java class libraries**. Then double-click any package, for example java.awt. The package browser opens.
2. Similarly, double-click any class in the package. The class browser opens.
3. Look at the Source pane for the class that you have opened. You can edit the source code directly in this pane.
4. Look at the Status area (the bar at the bottom of the window). This contains the name of the selected program element.
5. Look at the All Problems page. This page displays all the classes and methods in the workspace that have unresolved problems in them.
6. **[ENTERPRISE]** Look at the Managing page. This page lets you change ownership of program elements and version and release them.

Exploring the source code repository

The Repository Explorer is your visual interface to the repository.

To open the Repository Explorer, select **Window > Repository Explorer** from the menu bar.



The repository includes all editions of all program elements, including those that are currently in your workspace.

Within the Repository Explorer, you can open or compare program elements that are stored in the repository. You do not need to swap editions in and out of the workspace to view them or compare them.

By comparing different editions, you can see:

- What changes have been made as a result of code generation
- Precisely how an edition with errors differs from a bug-free edition

Refer to the online help for more information about the Repository Explorer.

Quick Start window

Several of the tasks you will most frequently perform in the IDE are listed in the Quick Start window. The **Basic** section lets you launch the appropriate SmartGuide (wizard) to create applets, classes, interfaces, projects, and packages, and to experiment with code.

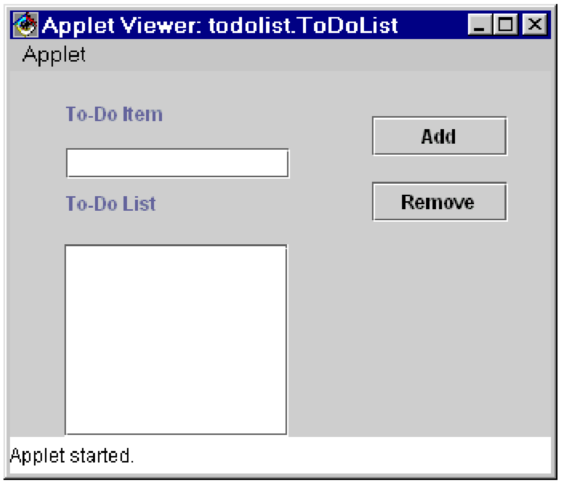
[ENTERPRISE] **Team Development** provides you with easy access to projects in the repository. Also, a team administrator can administer users. With **Repository Management,** you can use the **Change Repository** option to connect your workspace to another repository (shared or local), or to recover if the server fails.

To learn more about what you can do with the Quick Start window, press F1 in the window.

About the To-Do List applet

Overview
You will create a To-Do List applet which consists of a *composite* bean that is made up of several other beans. The applet has a text field bean (JTextField) for entering a To-Do item and a list bean (JList) for displaying the To-Do List items. There are also two button beans (JButton) for adding and removing items from the list.

The user interface for the completed To-Do List applet appears as follows:



When you have completed your applet, you will type an item into the **To-Do Item** field and select **Add**. This adds the item to the To-Do List. If you select an item from the To-Do List and select **Remove**, the item is removed from the To-Do List.

Creating the To-Do List applet

In addition to the applet, you will a project and a package to contain your work.

You will manipulate your new applet in the Visual Composition Editor. That is, you will visually manipulate JavaBeans components (beans). Beans are represented as classes and interfaces when you examine your applet in the Workbench.

Creating a project, package, and an applet
Follow these steps to create the To-Do List project, package and applet:

1. To open the SmartGuide, from the Workbench **File** menu, select **Quick Start.** The Quick Start window opens.
2. Select **Basic** > **Create Applet**.
3. Click **OK**. The Create Applet SmartGuide opens.
4. In the **Project** field, type the project name, `My ToDoList Project`.
5. In the **Package** field, type the package name, `todolist.`
6. In the **Applet name** field, type the applet name, `ToDoList.`
   Tip: Give your applets (and all other classes) names that begin with a capital letter. Class names are case-sensitive, and cannot contain spaces. If a class name consists of multiple words, do not type spaces between the words, but instead capitalize the first letter of each word (for example, *ToDoList*).

7. Select the **JApplet** radio button. Make sure that **Compose the class visually** check box is selected.
8. Click **Next**. Select the **Yes, create an applet which can be run by itself or in an applet viewer** radio button.
9. Click **Finish**.

VisualAge for Java creates a project, a package, and an applet, then opens the Visual Composition Editor for the applet.

Constructing the applet in the Visual Composition Editor

The Visual Composition Editor enables you to develop programs by visually arranging and connecting beans. This process of creating object-oriented programs by manipulating graphical representations of components is called *visual programming*. You can create *connections* to link events, such as clicking a button, with code that you create.

Beans and connections
In VisualAge for Java, *beans* are the components that you manipulate when you program visually. These beans are Java classes that adhere to the JavaBeans specification. In the Visual Composition Editor, you select beans from a palette, specify their characteristics, and make connections between them. Beans can contain other beans and connections to beans.

You use two types of beans within the Visual Composition Editor:

- A *visual* bean can be seen in your program at run time. Visual beans, such as windows, buttons, and text fields, make up the graphical user interface (GUI) of a program.
- A *nonvisual* bean does not appear in your program at run time. A nonvisual bean typically represents an object that encapsulates data and implements behavior within a program.

A bean's *public interface* determines how it can interact with other beans. The public interface of a bean consists of the following features:

- *Events* are signals that indicate something has happened. Opening a window or changing the value of a property, for example, will trigger an event.
- *Methods* are operations that a bean can perform. Methods can be triggered by connections from other beans.
- *Properties* are data that can be accessed by other beans. This data can represent any logical property of a bean, such as the balance of an account, the size of a shipment, or the label of a button.

  Properties can be read, written, or both. Properties can have the following characteristics:
  - A *bound* property triggers the *propertyChange* event when its value is changed.
  - A *constrained* property allows other beans to determine whether the value of the property can be changed (triggers the *vetoableChange* event).
  - An *indexed* property is an array, so it exposes additional methods to address individual elements.
  - A *hidden* property is not visible to humans. It is for use by bean-aware tools only.
  - An *expert* property should only be manipulated by expert users.

– A *normal* property is one that is neither hidden nor expert.

A bean *exposes* a feature when it makes that feature available to other beans.

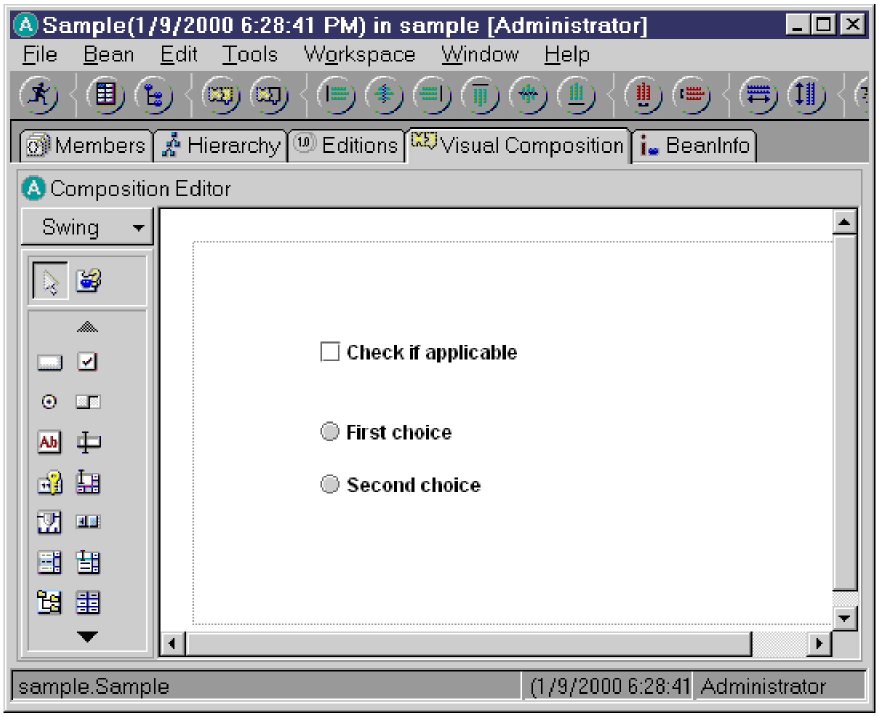
Beans can contain other beans as well as connections between beans. You can think of the beans that you construct in the Visual Composition Editor as *composite* beans because they contain other beans. The composite beans you build make up your program.

VisualAge for Java also includes support for JavaBeans components. JavaBeans components (or, more simply, *beans*) are Java objects that behave according to the JavaBeans specification. Beans are reusable software components that you can manipulate in a development environment like VisualAge for Java. The method signatures and class definition of a bean follow a pattern that permits environments like VisualAge for Java to determine their properties and behavior. This ability for a "beans-aware" environment to determine the characteristics of a bean is called *introspection*.

In the Visual Composition Editor, *connections* define how beans interact with each other. You can make connections between beans and between other connections. A connection has a source and a target. The point at which you start the connection is called the *source*; the point at which you end the connection is called the *target*.

The Visual Composition Editor
When the Visual Composition Editor opens, you can begin constructing your To-Do List applet. The To-Do List applet consists of a that contains several visual beans.

The Visual Composition Editor window includes several components: the *beans palette* along the left side, the *status area* along the bottom, the *toolbar* along the top, and the *free-form surface* where you lay out the beans. In the picture below, three beans are on the surface: a check box bean and two radio button beans.


Sample(1/9/2000 6:28:41 PM) in sample [Administrator]
File Bean Edit Tools Workspace Window Help
Members Hierarchy Editions Visual Composition BeanInfo
Composition Editor
Swing
□ Check if applicable
○ First choice
○ Second choice
sample.Sample
(1/9/2000 6:28:41
Administrator

Beans palette Free-form surface

Toolbar
As in the Workbench window, the Visual Composition Editor toolbar provides easy access to the tools commonly used while manipulating beans (The **Tools** menu also provides access to these tools).

Take a minute to move your mouse pointer over each icon on the toolbar. A label will appear that identifies the icon.

Most of the tools in the toolbar act on the beans that are currently selected in the free-form surface. If no beans are selected for a tool to act on, the tool is unavailable.

Beans palette
The beans palette, located on the left side of the Visual Composition Editor, contains the set of ready-made beans that you use most frequently. The beans palette organizes the beans into categories.

Select **Bean > Modify Palette**. The Modify Palette window, which you can use to perform tasks such as adding categories and beans to the palette opens. You can also use this window to change the size of the bean icons, delete a bean or category from the palette, reorder beansand resize the palette.

Take a moment to view all the categories in the palette by scrolling down. Use the arrows at the top and bottom of the palette to scroll up and down within the palette.

The status bar at the bottom of the Visual Composition Editor indicates the category and bean currently selected in the beans palette, or the bean or connection currently selected on the free-form surface.

You can also identify a bean by placing the mouse pointer over the icon for the bean. A label will appear that identifies the icon.

Free-form surface
The free-form surface, located on the right side of the Visual Composition Editor, is the area where you create your program.

Regardless of the type of bean, every bean has a pop-up menu that contains options you can use to modify or work with that bean.

Adding beans to the To-Do List applet

When the Visual Composition Editor opens for the new To-Do List applet, the default JApplet bean is represented as a gray rectangle on the free-form surface. To build the rest of the user interface, you must add several other visual beans.

As you add beans to the applet, you may find that the default JApplet bean is too small to accommodate all the other beans. If this happens, you can resize the JApplet bean by selecting it and dragging one of the selection handles.

Although this bean uses a <null> layout manager, it could be built using one of the layout managers, such as GridBagLayout. You can also convert from <null> to any other supported layout manager. To learn more about layout managers, refer to the online help.

Tip: If you want to delete a bean, you can do so by selecting it and pressing the **Delete** key, or by selecting the bean, right-clicking it and selecting **Delete** from its pop-up menu.

Adding a text field and a label
You will add a JTextField bean that is used to enter the To-Do List items, and a JLabel bean to identify the field. These beans are in the Swing category of the beans palette. For more information about Swing, go to the Sun website www.java.sun.com

Tip: As you work with the palette, you may find the icons too small. To make them larger, right-click on any gray area on the palette and select **Show Large Icons** from the pop-up pane.

To add a text field and a label, follow these steps:

1. From the beans palette, select the JTextField bean . The status area at the bottom of the Visual Composition Editor displays `Category: Swing Bean: JTextField`, reflecting the current selection in the beans palette. If you cannot find this bean, check that the word **Swing** appears at the top of the beans palette. Click the arrow next to it to view a list of other bean categories.
2. Drag the mouse pointer over the JApplet bean (the rectangle on the free-form surface). The pointer changes to a crosshair, indicating that it is now loaded with the bean you selected. Click where you want to drop the JTextField.

   Tip: If you accidentally picked the wrong bean and have not dropped the bean into the JApplet yet, select the correct bean, or select the **Selection** tool from the beans palette to unload the mouse pointer.

   After you have added the JTextField bean to the JApplet, you can change its position by dragging it with the mouse, as follows:

   - In Windows, click and hold down the left mouse button.
   - In the UNIX platforms, select the bean and hold down the middle mouse button. If you are using a mouse with only two buttons, you must hold down both mouse buttons.
3. Make the bean larger by by dragging a side of the rectangle using the left mouse button.
4. Next, add a label for your text entry field. Select the JLabel bean and drop it just above the JTextField.

   At this point, do not worry about the beans' exact positions. Later you will learn how to automatically match sizes and align beans.

Undoing changes in the Visual Composition Editor
If you make a change in the Visual Composition Editor and then decide that you do not want to keep that change, select **Undo** from the **Edit** menu to restore your work to its previous state. You can undo as many operations as you want, all the way back to when you opened the Visual Composition Editor for the current bean.

If you undo an operation, then decide that you did not want to undo it, select **Redo** from the **Edit** menu. **Redo** will restore the view to the state it was in before the last **Undo**. As soon as you close the Visual Composition Editor for your bean, you lose any ability to undo or redo changes.

Changing the text of a label and adding another label
Change the text of JLabel to `To-Do Item` by editing the text as follows:

1. In the free-form surface, select the JLabel bean. Select **Properties** from the toolbar.
2. In the Properties window, scroll down, select **text**, and type `To-Do Item` to replace JLabel1. The label now contains the new text. You may need to stretch the label to see it.
3. You can add another label below the JTextField by copying JLabel1. To copy a bean:
   a. Select the bean.
   b. Select **Edit** > **Copy**.
   c. Select **Edit** > **Paste**. The pointer changes to a crosshair, indicating that it is now loaded with the bean you selected.
   d. Click below the JTextField to add the new label.
4. Change the text for the new label to `To-Do List` by editing it as you did for JLabel1.

Tip: You can also copy a bean using the **Ctrl** key. Position the mouse pointer over the JLabel bean, hold down the Ctrl key, and drag the copy of the bean to below the JTextField bean.

Adding a scroll pane for your list

Add a scroll pane so that your list of items can be scrolled.

1. Scroll down and select the JScrollPane bean.
2. Add the JScrollPane bean below the To-Do List field.

Adding a list

To create the list in which the To-Do items are displayed, you need to add a JList by following these steps:

1. Select the JList bean and place it inside the scroll pane. The JList bean adjusts to fill the JScrollPane.
2. In the Properties window, change the **selectionMode** property to SINGLE_SELECTION. This simplifies the code needed to handle selection within the list.
3. You should save your work periodically. To save your work, from the **Bean** menu select **Save Bean**.

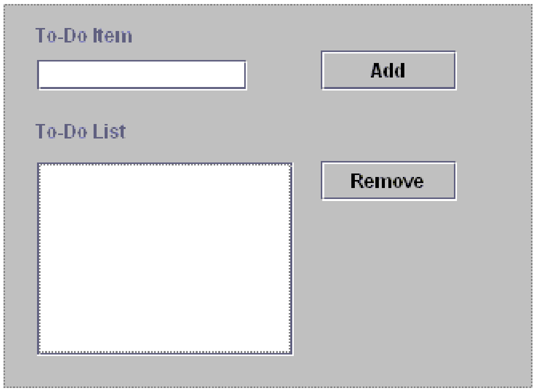
Adding buttons

To add and remove items from the To-Do list, you need to add two buttons:

1. To add more than one instance of a bean at a time, press and hold the Ctrl key before selecting the bean.
2. Select the JButton bean and add a button to the right of the text field.

   Notice that the mouse pointer remains a crosshair, indicating that it is still loaded with the JButton bean. To add another JButton, click below JButton1.
3. Select the **Selection** tool from the beans palette to unload the mouse pointer.
4. As you did with the Label beans, change the text on JButton1 to `Add`. Also in the property sheet, name the bean `AddButton` in the **beanName** field
5. Change the text on JButton2 to `Remove`. Also in the property sheet, name the bean `RemoveButton`.
6. Save your work. Select **Bean** > **Save Bean**.

You have just created your first user interface using VisualAge for Java. The free-form surface of the Visual Composition Editor should look something like this:



Next, you need to size and align the beans within the To-Do List applet.

Sizing and aligning visual beans

Since this bean does not use a layout manager (a class which controls how your beans are laid out), you need to clean up the appearance of your user interface by using the sizing and aligning tools from the toolbar on the Visual Composition Editor. The toolbar provides several different tools for sizing and aligning beans. The best way to learn about them is to experiment with them.

The following steps explain how to match the size of two beans, align the beans with other beans, and evenly distribute the beans within another bean.

Sizing, aligning, and distributing beans
The order in which you size, align, and distribute the beans is not always important. Usually, you start with the upper left corner and work your way through all the beans in the window.

To size the list so it matches the width of the text field, do the following:

1. Select the Beans List from the toolbar.
2. From the list, select JScrollPane1.

   Remember, you want to size the container for the list, which is the scroll pane. You are using the Beans List to do this because the JList bean almost completely overlays the JScrollPane bean, which makes it difficult to select from the free-form surface.
3. Hold down the **Ctrl** key to select multiple items and, in the free-form surface, select the text field.
4. Select the **Match Width** tool from the toolbar.

   Because the text field was selected last, it becomes the anchor bean for the match width operation. The width of the list is changed to match the width of the text field.

Note: The anchor bean has solid selection handles. The other selected items have outlined selection handles.

To size and align the Add button and the Remove button, do the following:

1. Resize the **Remove** button to an appropriate size for the applet.
2. Select the **Add** button, hold down the Ctrl key and select the **Remove** button.

   Then, select the **Match Width** tool from the toolbar.
3. Because the buttons remain selected, you can now align their left edges by

   selecting the **Align Left** tool from the toolbar.

To align the left side of the text field, list, and labels, do the following:

1. Move the label for the text field (the one that says **To-Do Item**) to where you want it in the applet.
2. Select the scroll pane (making certain it is the scroll pane, not the list), the text field and their associated labels, making sure to select the label of the text field last.

   By selecting the text field label last, you make it the anchor bean for the alignment operation.
3. Select the **Align Left** tool from the toolbar.
4. Because the text field, list, and labels are still selected, you can evenly distribute

   them in the window by selecting the **Distribute Vertically** tool from the toolbar.
5. Save your work. Note that the entire applet that you are creating is a bean. When you select **Bean** and then **Save Bean** from the menu, you are saving the entire applet.

You have now completely finished the user interface of your To-Do List applet. If you want to learn more about sizing, aligning, and distributing visual beans, refer to the online help for the Visual Composition Editor (VCE) .

Connecting the To-Do List beans

Now that you have added the visual beans to create the user interface, the next step is connecting them.

Event-to-method and parameter-from-property connections
The To-Do List applet is supposed to add the text entered in the text field to the list when the **Add** push button is selected. In this example, you will extend your list to include a model called DefaultListModel. The Java Foundation Classes, also known as *Swing*, separate data from the view of the data. The actual list items are stored in the list model; an event-to-method connection sends the model's data to the list in the applet.

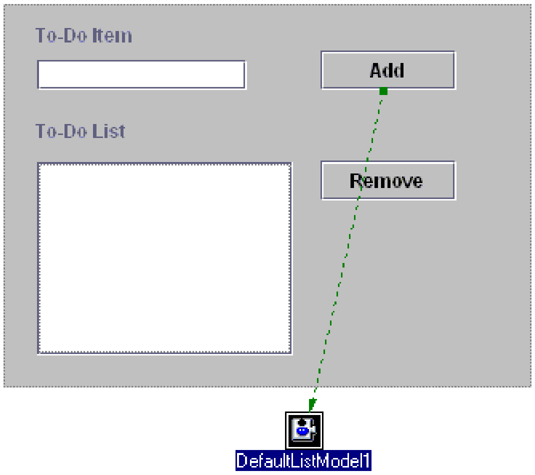
Selecting a push button signals an actionPerformed(java.awt.event.ActionEvent) event and adding an item to the list model is performed by the addElement(java.lang.Object) method. Therefore, the event-to-method connection for adding an item to the list model is between the AddButton's actionPerformed(java.awt.event.ActionEvent) event and the DefaultListModel's addElement(java.lang.Object) method.

Simply adding this event-to-method connection does not actually cause anything to be added to the list model because its addElement(java.lang.Object) method requires a parameter that specifies what object is to be added to the list. You will specify the parameter by creating a parameter-from-property connection. The contents of the text field is provided as the parameter to the addElement(java.lang.Object) method of the list model.

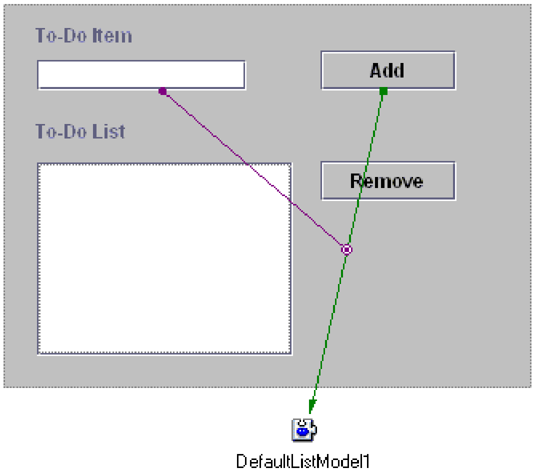
Adding a list model and making your connections

1. Select the **Choose Bean** tool from the palette.
2. Select **Class** as the Bean Type from the Choose Bean window. Using **Browse** with a pattern of de, select the DefaultListModel class. Click **OK** and place the DefaultListModel bean on the free-form surface, below the applet (the gray area). By default, the new bean is named DefaultListModel1.
3. Now you can begin your connections to move items into your list. Select the **Add** push button and select **Connect** > **actionPerformed** from its pop-up menu.

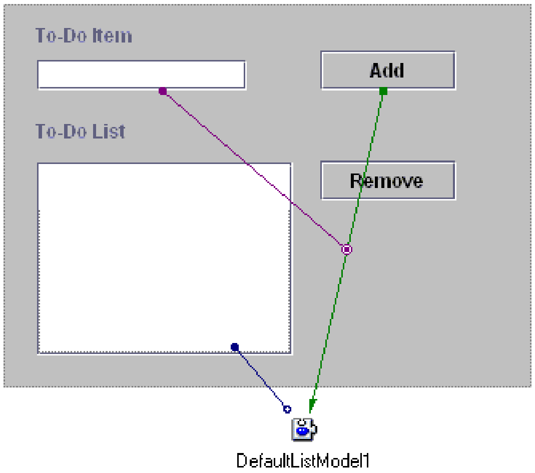
   The mouse pointer changes, indicating that you are in the process of making a connection. If you accidentally started the wrong connection, press the **Esc** key to cancel.
4. To complete the connection, click on DefaultListModel1 and select **Connectable Features**. Select **addElement(java.lang.Object)** from the window that opens. A dashed line appears, which means that more information is necessary. In this case, the value of the parameter for the addElement(java.lang.Object) method is missing.



5. To make the parameter-from-property connection that specifies what to add to the list, follow these steps:
   - Click on the dashed event-to-method connection line and select **Connect** > **obj** from its pop-up menu.
   - Click on the text field, then select textfrom its pop-up menu.



6. Finally, you must get the To-Do items from DefaultListModel1 to the list in the applet, so that users can see them in the applet. This is a connection that sends data from a source (the model) to a view (the list). Right-click on DefaultListModel1 and select **Connect** > **this** from its pop-up menu. Click on the list and select **model** from the pop-up menu. A purple line appears, indicating a property-to-property connection, as shown below.



7. Save your work.

Event-to-code connection
Event-to-code programming enables you to associate a handwritten method with an event. In the first applet you created, you did not need to write any methods because VisualAge for Java generated them based on your visual elements and connections. However, sometimes additional logic is required.

For this example, you will use an event-to-code connection to activate the **Remove** push button. An event-to-code connection enables us to specify several behaviors in one method. We could accomplish this without writing code by hand, but in this case, writing the code by hand is more efficient.

1. Click on the **Remove** push button and select **Connect** > **actionPerformed** from its pop-up menu.
2. Right-click on an empty area of the free-form surface. In the pop-up menu that opens, select **Event to Code**. The Event-to-Code Connection window appears.

By default, VisualAge for Java creates a new method stub called `removeButton_ActionPerformed` (you renamed the button beans so you could know at a glance which methods belong to which button). This method is scoped to the composite bean class (ToDoList).

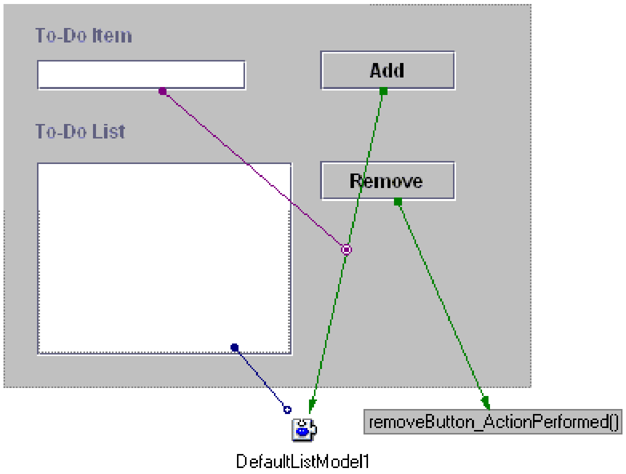
3. Copy the following code over the method stub. You can copy and paste this section from the online version of *Getting Started*:

```
public void removeButton_ActionPerformed(java.awt.event.ActionEvent actionEvent) {
    // Grab selected item
    Object itemToBeRemoved = getJList1().getSelectedValue();
    // Remove item from model
    getDefaultListModel1().removeElement(itemToBeRemoved);
    // Echo item to text field
    getJTextField1().setText(itemToBeRemoved.toString());
    // Refresh list from model
    getJList1().setModel(getDefaultListModel1());
    return;
}
```

This method calls get methods generated by VisualAge for Java to access the beans embedded in ToDoList.

4. Select **OK** to close the window and save the new method.

Your connections should now look like this:



With the user interface complete and the behavior of the applet defined by the connections between the visual beans, you are now ready to test your work.

There are three other types of connections:

Property-to-property
Property-to property connections link two data values together.

Parameter-from-code
Parameter-from-code connections call code whenever a specified event occurs.

Parameter-from-method
Parameter-from-method connections use the result ofmethod as a parameter to a connection.

For more information about connections, refer to the online help.

Saving and testing the To-Do List applet

You have already saved parts of your applet as you have been working. When you save changes to a bean, you replace the old specification of the bean with a new one. When you do this, VisualAge for Java uses the new specification of the bean for all new uses of it. It is good practice to save your changes to a bean periodically as you are working with it and when you have finished editing it.

Saving your visual bean
In the Visual Composition Editor, to save your bean, select **Bean > Save Bean**.

A message box appears saying that your bean is being saved and that *run-time code* is being generated. This generated run-time code is what is used to create your bean when you run your application.

Testing the applet
Now that your work is saved, you can test your To-Do List applet.

1. To begin testing your applet from within the Visual Composition Editor, select **Run** 🏃 from the toolbar. The Applet Viewer starts with your applet in it.
2. When the To-Do List applet appears, experiment with it to ensure that it behaves the way you expect it to. For the To-Do List applet, you need to ensure you can add typed items to the list and remove selected items from the list.

Note: As you design your applet, the Swing beans are presented with Sun's Metal look. For example, a JButton will look like a Metal JButton as opposed to a Windows JButton. You can add your own code to have your beans rendered in a different system's look and feel at run time. For example, you could add code to have your beans have a Windows look and feel. Adding this kind of code is not covered in this document.

Close the applet window when you have completed your testing.

At any time, you may return to the Visual Composition Editor and make changes, save the changes, then test the applet again.

Your To-Do List applet is now finished.

Saving your workspace
Before you continue, save your workspace. When you save your workspace, you are saving the current state of all the code that you are working on and the state of any windows that you currently have open. To save your workspace, select **File** > **Save Workspace**.

Chapter 4. Adding state checking to the To-Do List applet

To keep the applet as simple as possible, we did not include any kind of state checking. The **Add** and **Remove** push buttons are always available. This is not the ideal behavior for the applet. For example, a user should only be able to select the **Remove** push button if there is an item selected in the to-do list.

This section leads you through the steps to add state checking to your applet. You will review what you learned when you created the To-Do List applet and to learn a bit more about how the Visual Composition Editor works. This section only deals with the **Remove** push button, but if you want to experiment, you can try to add the same kind of state checking for the **Add** push button.

Before we update the applet to include state checking, we'll go through the steps to find your applet and to create a versioned edition of it.

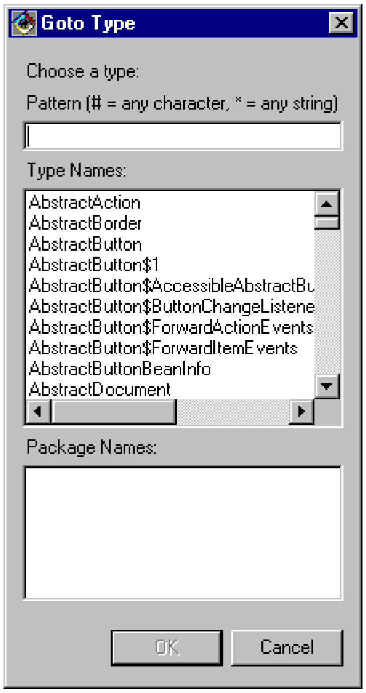
Note: This section assumes that you have completed the steps described in Creating the To-Do List applet. You should now have a completed, working To-Do List applet. If you have not done so already, please complete the steps to create the basic To-Do List applet.

Finding a program element in the Workbench

Before you can add state checking to your To-Do List applet, you may need to find it. When you created it in the Visual Composition Editor, you may not have kept track of where VisualAge for Java put the code that it generated to implement the applet.

VisualAge for Java gives you powerful search capabilities for finding program elements. Here is quick way to find your To-Do List applet:

1. In the Workbench window, select the **Projects** page.
2. From the **Selected** menu, select **Go To** > **Type**. The GoTo Type window appears:


Goto Type
Choose a type:
Pattern (# = any character, * = any string)
Type Names:
AbstractAction
AbstractBorder
AbstractButton
AbstractButton$1
AbstractButton$AccessibleAbstractBu
AbstractButton$ButtonChangeListene
AbstractButton$ForwardActionEvents
AbstractButton$ForwardItemEvents
AbstractButtonBeanInfo
AbstractDocument
Package Names:
OK
Cancel

3. Enter the name of your To-Do List applet (for example, `ToDoList`) in the **Pattern** field. As you enter the name, the **Type Names** list changes to include only the types (that is, the classes and interfaces) that match what you have entered so far.
4. Select your To-Do List applet from the **Type Names** list. If packages are listed in **Package Names**, it means that more than one package has a class with the name you specified. Select the package in which you created the To-Do List applet and click **OK**.
5. The list of projects is updated. The project and package that contain your applet are expanded, and the class for your applet is selected.

Now you have found the class for your applet, you are ready to version it.

Tip: You could also search for the To-Do List applet using the Open Type dialog. You can access it by selecting **Workspace** > **Open Type Browser,** and following steps 3 to 5 above.

If you want to search for a program element that is not a type (for example, a a project or a package, you can search for it in the Workbench. For example, in program element panes, if you type a letter, VisualAge for Java selects the first displayed program element that begins with that letter. If you type the same letter again, the next program element that begins with that letter is selected.

Versioning the To-Do List applet

When you *version* an edition of a program element, you give it a name and explicitly save its current state. When you make more changes to the code and save these changes, a new edition is created based on the versioned edition. If you decide you want to undo your changes or try a different set of changes, you can simply return to the versioned edition.

Before you make any changes to your To-Do List applet class, version it so you can return to it if necessary. To version the class for your applet:

1. In the Workbench window, ensure that the class for your applet is selected. From its pop-up menu, select **Manage** > **Version**. The Versioning Selected Items SmartGuide opens.
2. Ensure that the **Automatic** radio button is selected and click **OK**. If **Show Edition Names** is selected in the toolbar, a version name appears next to the class.

The next time you modify this class and save it, VisualAge for Java creates a new edition based on the code in this versioned edition. If you run into problems while you are making updates to your applet, you can return to the previous edition of the To-Do List applet that you versioned.

[ENTERPRISE] Editions are managed by a team of developers in the Enterprise edition. Only owners of editions can version them. Also, versioning is often followed by releasing. For more information on the team development environment, refer to the team programming section of this document.

Enabling state checking of the To-Do List applet

Now that you have versioned an edition of your applet, you are ready to add state checking.

Desired behavior of the Remove push button
Currently, the **Remove** push button is always enabled, even if no items are in the list. Here is how the **Remove** push button should work:

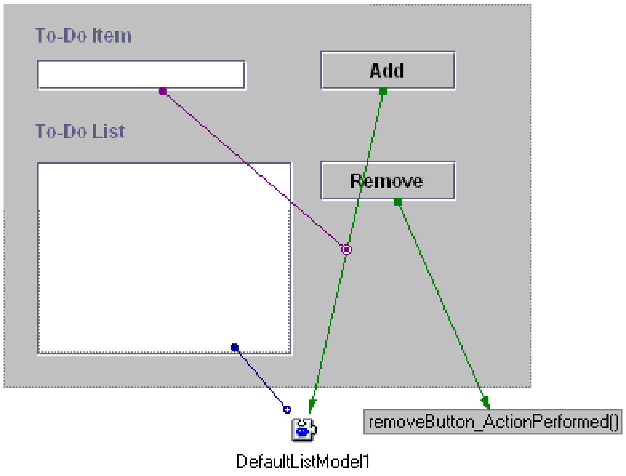
- When the applet starts, the **Remove** push button should be disabled.
- When an item is selected in the To-Do List, the **Remove** push button should be enabled.
- When a selected item in the To-Do List has been deleted, the **Remove** push button should be disabled.

To get the desired behavior for the **Remove** button, you will do the following tasks:

- Open the To-Do List applet in the Visual Composition Editor .
- Set the properties of the **Remove** push button so it is disabled when the applet first starts.
- Add an event-to-code connection to enable the push button when an item is selected in the To-Do List.

Opening your To-Do List applet in the Visual Composition Editor
First, open your To-Do List applet class in the Visual Composition Editor:

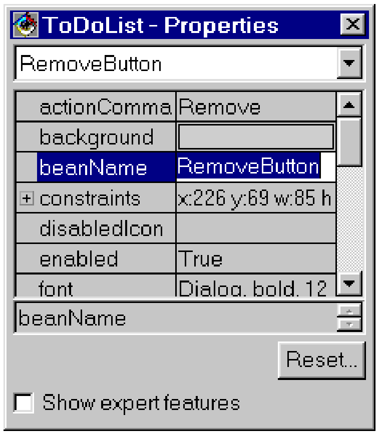
1. Right-click the **ToDoList** class in the Workbench.
2. From the pop-up menu, select **Open To > Visual Composition.**
3. The free-form surface should look like this:



Setting the properties of the Remove push button

Now set the properties of the **Remove** push button so it is disabled when the applet starts:

1. From the Remove push button's pop-up menu, select **Properties**. The Properties window opens.



ToDoList - Properties

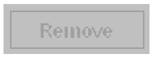
RemoveButton

actionComma	Remove
background	
beanName	RemoveButton
⊞ constraints	x:226 y:69 w:85 h
disabledIcon	
enabled	True
font	Dialog, bold, 12

beanName

☐ Show expert features

2. Select the field to the right of **enabled**. Select **False** from the drop-down list in this field and close the Properties window. The **Remove** push button should now appear disabled:



Adding a connection to enable and disable the Remove push button

Next, add the connection that enables the **Remove** push button when an item is selected in the To-Do List:

1. Select the JList bean.
2. From its pop-up menu select **Connect** > **Connectable Features**. A connection window opens.
3. Select the **Event** radio button. Select **listSelectionEvents**. Click **OK**. The mouse pointer changes to indicate that you are in the process of making a connection.
4. Complete the connection by clicking on the free-form surface and selecting **Event to Code**.

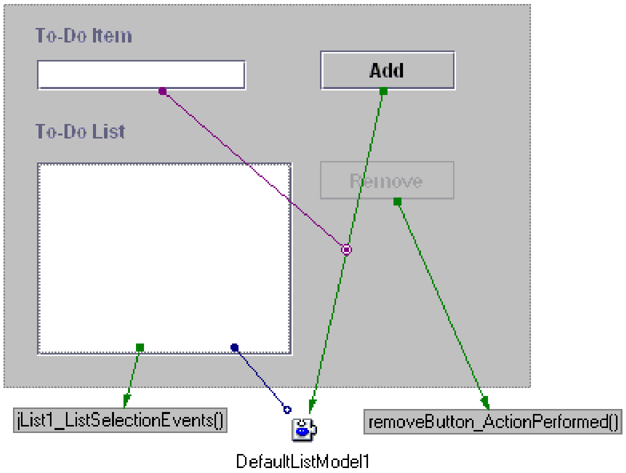
The Event-to-Code Connection window appears. By default, VisualAge for Java creates a new method stub called `jList1_ListSelectionEvents()`.

5. Copy the following code over the method stub:

```
public void jList1_ListSelectionEvents() {
    if (getJList1().getSelectedIndex() < 0)
        getRemoveButton().setEnabled(false);
    else
        getRemoveButton().setEnabled(true);
    return;
}
```

   This method calls the getSelectedIndex() method of JList1. If the method returns -1, no items are selected in the list, and the *enabled* property of RemoveButton is set to false. Otherwise, *enabled* is set to true.

6. Click OKto close the window. Your To-Do List should now look like this:



Saving and testing your changes

Before you continue, save your work and test it:

1. To save the current state of your work in the Visual Composition Editor, select **Bean > Save Bean**.

2. To test the changes you made, select **Run** from the toolbar. The Applet Viewer appears with your applet.

3. Experiment with the applet to ensure that the behavior of the **Remove** push button is correct. Ensure that the **Remove** push button is disabled when the applet starts and then becomes enabled as soon as an item is selected in the To-Do List.

You have successfully added state checking to your To-Do List applet. Now that you have a new level of your code working, create another versioned edition of it by following the steps in “Versioning the To-Do List applet” on page 32.

Working with your source code

Now that you have create your applet and added state checking to it, take a moment to look at the code that VisualAge for Java has automatically created by following these steps:

1. In the Workbench, select the **ToDoList** class.
2. Expand it by clicking the plus (+) sign.

3. Select **main**. This method (which was automatically created by VisualAge for Java), enables you to run your applet as an application. The Java source code is displayed in the Source pane of the Workbench window.
4. Select other methods and fields to examine their code.

You can edit the code for these methods and fields directly in the Source pane of the Workbench. The IDE automatically format how your code appears when you type it in the Source pane. To set code formatter options, including indentation and new-line controls, follow these steps:

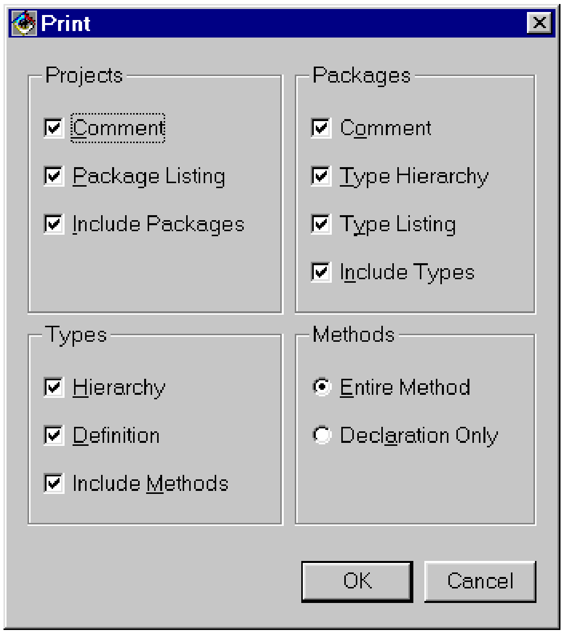
1. In any window select **Window > Options**. The Options dialog opens.
2. Expand the **Coding** item by clicking the + sign.
3. Select **Formatter** to select when you want a new line of code to start.
4. Select **Indentation** to select an indentation style for your code.

These specifications are automatically applied to all new code. If you have imported code from the file system, or if you change the formatting options, you can apply the options to code in a source code pane's pop-up menu. Press F1 in the Options dialog to learn more about these options.

You can print the contents of any projects, packages, types (classes/interfaces), or methods in the Integrated Development Environment (IDE). When you print a program element, you can select to print the contents of the program elements it contains. For example, when you print a package, you can also print the classes in the package.

To print the To-Do List project, follow these steps:

1. In the Workbench, select the **My ToDoList** project**.** From its pop-up menu, select **Document** > **Print**. The Print dialog box opens:



Print

Projects
☑ Comment
☑ Package Listing
☑ Include Packages

Packages
☑ Comment
☑ Type Hierarchy
☑ Type Listing
☑ Include Types

Types
☑ Hierarchy
☑ Definition
☑ Include Methods

Methods
● Entire Method
○ Declaration Only

2. By default, all the options under **Projects**, **Packages**, and **Types** are selected, and **Entire Method** is selected under **Methods**. You can clear any of these options.
3. Click **OK** to start printing. If no default printer has been selected, a message appears asking you to select one.

Chapter 5. Enhancing the To-Do List applet

In the previous section, you added state checking to your simple To-Do List applet. This section leads you through the steps of modifying your simple To-Do List applet so that it can save To-Do lists to named files and open files containing To-Do lists.

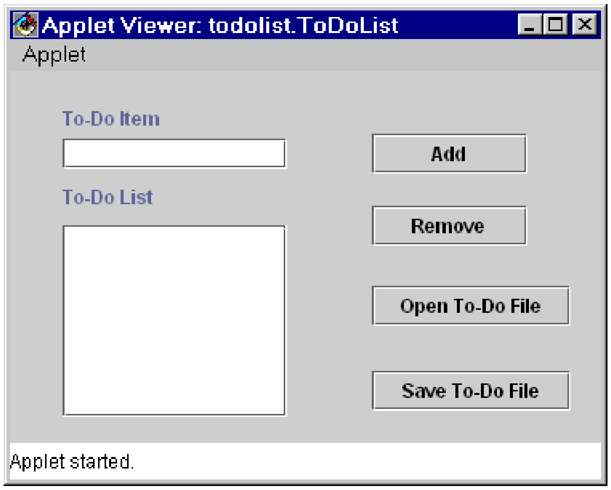
As you modify your applet, you will learn about:

- Creating new methods
- Adding business logic code in the Visual Composition Editor
- Updating the user interface
- Running code as an applet or an application

Note: This section assumes that you have completed the steps described in "Adding state checking to your applet." You should now have a completed, working To-Do List applet with simple state checking. If you have not done so already, please complete the steps to add state checking to your To-Do List applet.

Before jumping into the modifications that you will be making to your applet to create the updated To-Do List program, review how the finished program will work.

Here is what the To-Do List program will look like:



Like your existing applet, the updated To-Do List program adds the text in the **To-Do Item** field to **To-Do List** when you select the **Add** button. When you select the **Remove** button, the program removes the selected item from the To-Do list.

What about the new buttons? Here is an overview of their behavior:

- When you select **Open To-Do File**, the program loads the contents of a list file into the To-Do List program.
- When you select **Save To-Do File**, the contents of the To-Do List are copied into the list file.

In addition to the differences in interface and behavior, there is one other important difference between the To-Do List applet and the To-Do List program. Because it needs access to the file system to read and write files, the To-Do List program must be run as an application rather than an applet. Java applets are not allowed to access the file system.

Important: If you run this as an applet, do not attempt to save or open the To-Do List file. If you do, the applet will not save or open the file properly. If you run this sample from main(), the save function works properly.

Running the To-Do List from main():
1. From the **Project** page of the Workbench, right-click the **ToDoList** class and select **Run**.
2. Select **Run main.** You may need to resize the window to view the applet properly.

Searching the workspace

You can use the Search dialog to perform powerful searches of the workspace. This search mechanism searches the workspace (or a part of it) for program elements or text. You cannot use the Search dialog to search directly for projects or packages, but you can search for the program elements (classes, interfaces etc.) in them. It can find declarations of and references to program elements.

To search for the ToDoList class:

1. Click the **Search** button on the Workbench's toolbar. The Search dialog opens. If you selected text or a program element before launching the search, the Search string field will contain what was selected.
2. Type `ToDoList` in the Search string field.
3. If it is not already selected, select the **Type** radio button.
4. Click **Start**.
5. The Search Results window opens. The ToDoList class will be listed in your search results matches.

You can edit the code for the ToDoList class in the source pane of the Search Results window. For more information about the Search dialog, refer to the online help.

Importing the required classes

The new methods require two classes to be imported: com.sun.javax.swing.* and java.io.* To import these additional classes:

1. From the Workbench, select the **ToDoList** class.
2. In the Source pane, add these statements beneath the existing import statements for the java.applet.* and java.awt.* classes:

```
import javax.swing.*;
import java.io.*;
```

Adding a static variable required by the new methods

The new methods require a static variable to hold the name of the file in which the to-do list will be saved. Add this variable as follows:

1. From the Workbench, select the **ToDoList** class.
2. In the Source pane, add this statement beneath the fields defined for the class:

```
static String FILE_NAME = "todo.lst";
```

Save your changes by right-clicking in the Source pane and selecting **Save**.

Tip: You can search for references to a selected type or one of its fields. From the ToDoList class' pop-up menu select **References To > This Typ**e to search for references to the selected type. Select **References to > Field** to search for references to one of its fields.

For methods, the **References To** pop-up menu option has sub-options that search the workspace for references to the selected method, methods it calls, fields it accesses, or types it references. The **Declarations Of** sub-options search for declarations of these same program elements. Results of the search are displayed in the Search Results window.

Adding a method for reading files

Now you are ready to create a method called readToDoFile(). This method reads the contents of a list file line by line and adds each line to the DefaultListModel1 instance.

1. Right-click the **ToDoList** class in the Workbench.
2. From the pop-up menu, select **Add > Method.**
3. When the Create Method SmartGuide appears, enter the following method name:

```
void readToDoFile()
```

4. Select **Finish** to generate the method.
5. Select the new method and add the code to implement it. If you are viewing this document in a browser, you can select the following code, copy it, and paste it into the **Source** pane. The finished method should look like this:

```
public void readToDoFile() {
  FileReader fileInStream;
  BufferedReader dataInStream;
  String result;
  try {
   // read the file and fill the list
   fileInStream = new FileReader(FILE_NAME);
   dataInStream = new BufferedReader(fileInStream);
   // clear the existing entries from the list
   getDefaultListModel1().removeAllElements();
   // for each line in the file create an item in the list
   while ((result = dataInStream.readLine()) != null) {
     if (result.length() != 0)
      getDefaultListModel1().addElement(result);
   }
   fileInStream.close();
   dataInStream.close();
  } catch (Throwable exc) {
    handleException(exc);
  }
  return;
}
```

Tip: You can access context-sensitive help for Java keywords while you are entering the source code. For example, when you type the line:

```
try {
```

highlight `try` and select **View Reference Help** from its pop-up menu.The Java keyword help file for `try`, which contains information about what the keyword does and when to use it, opens.

You can also access API reference help for a package, type, or member. For example, when you type the line:

```
String result;
```

highlight `String` by double-clicking it, and select **View Reference Help** from its pop-up menu.

6. Select **Edit** > **Save** to save your changes and recompile.

Before continuing with the next task, review the code in this method:

1. At the beginning of the method, there are declarations of the fields that are used to manipulate the file and its contents.

```
FileReader fileInStream;
BufferedReader dataInStream;
String result;
```

2. Next, statements associate the file with a FileReader and associate the FileReader with a BufferedReader. Using a BufferedReader makes it possible to read the file a line at a time. You defined the static `FILE_NAME` variable previously.

```
try {
  // read the file and fill the list
  fileInStream = new FileReader(FILE_NAME);
  dataInStream = new BufferedReader(fileInStream);
```

3. Next, we clear the list. Then, a loop reads the file one line at a time into the String `result`. If `result` is not a zero-length String, it adds the value of `result` to the list model.

```
// clear the existing entries from the list
getDefaultListModel1().removeAllElements();
// for each line in the file create an item in the list
while ((result = dataInStream.readLine()) != null) {
  if (result.length() != 0)
   getDefaultListModel1().addElement(result);
}
```

4. At the end of the `try` block, statements close the streams associated with the file.

```
fileInStream.close();
dataInStream.close();}
```

5. The catch block passes any exceptions to handleException*()*. VisualAge for Java generates this method for all visually composed classes. The comment delimiters have been removed.

```
catch (Throwable exc) {
  handleException(exc);
  }
}
...
handleException(Throwable exc) {
```

```
/* Uncomment the following lines to print uncaught exceptions to stdout */
System.out.println("--------- UNCAUGHT EXCEPTION ---------");
exception.printStackTrace(System.out);
}
```

Adding another method for writing files

You have one more method to add to the ToDoList class: writeToDoFile(). This method writes the contents of the list model line by line into a list file.

1. Right-click the **ToDoList** class in the Workbench.
2. From the pop-up menu, select **Add > Method**. When the Create Method SmartGuide appears, enter the following in the method name:

   ```
   void writeToDoFile()
   ```
3. Select **Finish** to generate the method.
4. Select the new writeToDoFile() method and add the code to implement it. If you are viewing this document in a browser, you can select the following code, copy it, and paste it into the Source pane. The finished method should look like this:

   ```
   public void writeToDoFile() {
     FileWriter fileOutStream;
     PrintWriter dataOutStream;
     // carriage return and line feed constant
     String crlf = System.getProperties().getProperty("line.separator");
     // write the file from the list
     try {
      fileOutStream = new FileWriter(FILE_NAME);
      dataOutStream = new PrintWriter(fileOutStream);
     // for every item in the list, write a line to the output file
     for (int i = 0; i < getDefaultListModel1().size(); i++)
      dataOutStream.write(getDefaultListModel1().getElementAt(i) + crlf);
     fileOutStream.close();
     dataOutStream.close();
     } catch (Throwable exc) {
       handleException(exc);
     }
     return;
   }
   ```
5. Select **Save** from the **Edit** menu to save your changes and recompile.

This code is similar to that for readToDoFile(). Before continuing with the next step, review the loop that actually writes lines to the file:

```
// for every item in the list, write a line to the output file
for (int i = 0; i < getDefaultListModel1().size(); i++)
  dataOutStream.write(getDefaultListModel1().getElementAt(i) + crlf);
```

This loop goes through each item in the list model. Each item is appended with `crlf` (a String consisting of the line separator characters) and written to the file. The line separator characters force each item to be written on a separate line in the file.

Testing the code using the Scrapbook
Before continuing, pause and consider the line separator for a moment. Suppose you have never seen this before and you want to see how it works. You can use the Scrapbook window to test out a code fragment that exercises this part of your class.

To test the line separator code:

1. Select **Window** > **Scrapbook**. The Scrapbook window opens.

2. Enter the following code into a page in the Scrapbook window:

   ```
   String crlf = System.getProperties().getProperty("line.separator");
   System.out.println("Here is one line."+crlf+"And here's another line.");
   ```

   Tip: When you are entering the code, you can use *code assist* to help you. Press Ctrl+Spacebar in the source code pane or Scrapbook. Code assist is available in any window that contains source code.

   Code assist shows you a list of classes, methods, and types that could be inserted in the code at the cursor. For example, when you type `String`, press Ctrl+Spacebar after you have typed `St` and a pop-up list of options will appear. Scroll down to **String** and double-click it to select it.

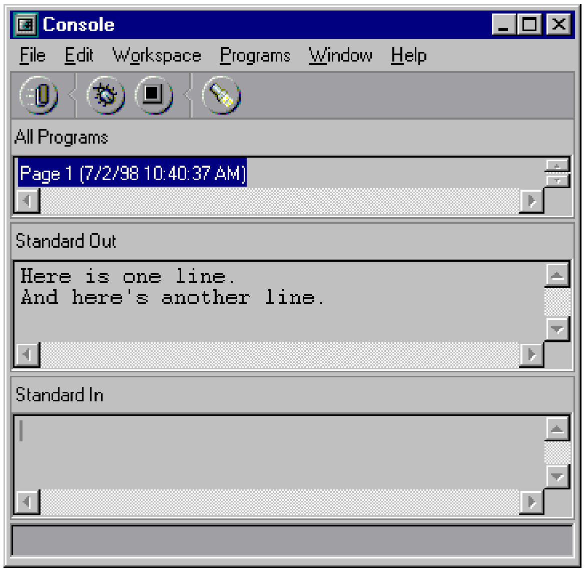
   To access code assist for methods and fields, you can enter the name of a object, a period(.), and a few letters from the start of a method or field, then press Ctrl+Spacebar. For example, type String crlf = System.get and then press Ctrl+ Spacebar. A list of methods, including getProperties appears. Double-click getProperties to select it.

   If pressing Ctrl+Spacebar does not launch code assist on your system, try using Ctrl+L. For more information about using and customizing code assist, refer to the online help.

3. Select both lines of code and select **Run** from the Scrapbook window toolbar.
4. Select **Window** > **Console**. The Console window displays standard output. Notice that the line separator splits the output so that it appears on separate lines.

   It also gives you an area for entering input to standard input.If more than one thread is waiting for input from standard in, you can select which thread gets the input.

   The Console window should look like this:



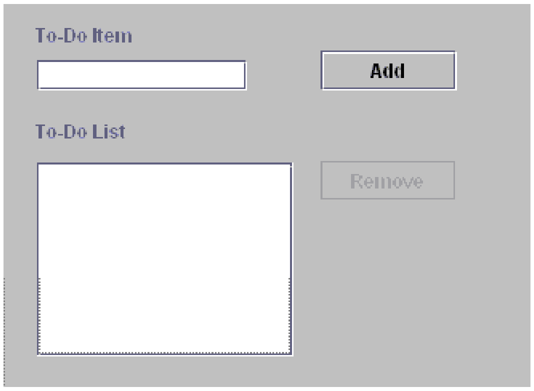
```
Here is one line.
And here's another line.
```

This simple example demonstrates how you can use the Scrapbook window to try out a piece of code quickly and conveniently.

You can clone the Console window. Cloning a window in the IDE creates a duplicate instance of the current window. To clone it, select **Window > Clone**. For more information, refer to the online help.

Adding buttons to the To-Do List applet

You have completed all of the steps that added logic to the To-Do List class. Now you are ready to make modifications to the user interface of the To-Do List applet. Your current To-Do List applet should look like this:

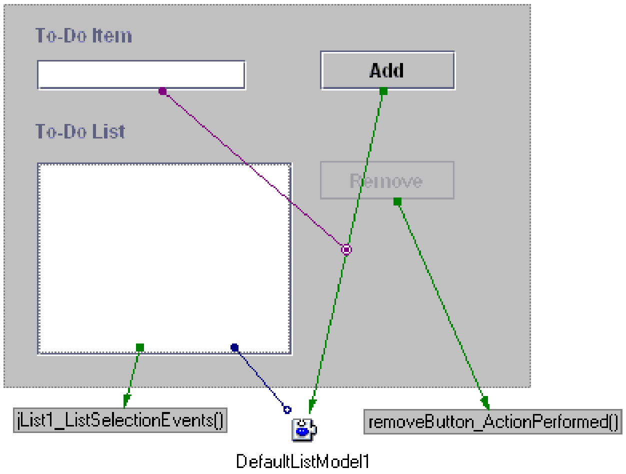


You need to add two new buttons to this user interface:

- An **Open To-Do File** push button to trigger opening a file to read into the To-Do list
- A **Save To-Do File** push button to trigger saving the contents of the To-Do list to a file

To add these two push buttons:

1. Right-click the **ToDoList** class in the Workbench. From the pop-up menu, select **Open To > Visual Compositon.**
2. The free-form surface appears. It should look like this:



3. Select a JButton bean and add a push button under the existing **Remove** push button. Name it `ReadButton`. You may need to move your **Add** and **Remove** push buttons or lengthen the free-form surface to make space for your new push button.

4. Select the push button you just added and change its text to **Open To-Do File**. To change the text:
   - Open the **Properties** window for the `ReadButton`.
   - Change the value of *text* to `Open To-Do File`.
5. Follow the same procedure to add another push button below ReadButton. Name it `SaveButton`. Change the text of this push button to `Save To-Do File`.
6. Size the new push buttons to match the width of the existing buttons:
   - Select the **Open To-Do File** push button. Hold down the **Ctrl** key and select the remaining buttons so that all four buttons are selected. The last bean selected has solid selection handles, indicating that it is the *anchor bean*. The anchor bean is the bean that acts as the guide for resizing or the bean that the other selected beans match.
   - Select **Match Width** from the toolbar.
7. Align the two new push buttons with the existing buttons:
   - Select **Save To-Do File** push button. Hold down the **Ctrl** key and select the other three push buttons. The **Add** push button should be your anchor bean.
   - Select **Align Left** from the toolbar.
8. Distribute evenly all the four push buttons. Since you have all the push buttons already selected, click **Distribute Vertically** on the toolbar.

You have added the two new push buttons for the To-Do List program. Now you are ready to associate them with an action.

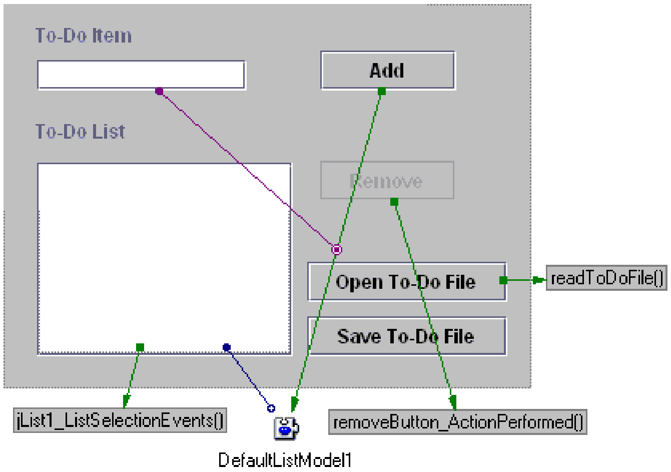
Connecting the Open To-Do File button

Now that you have added all the new beans to the free-form surface, you are ready to begin connecting them.

The To-Do List applet should call the readToDoFile() method when a user selects the **Open To-Do File** button. To implement this, you must connect the **Open To-Do File** to the readToDoFile() method.

1. Select the **Open To-Do File** button. Select **Connect** > **actionPerformed** from its pop-up menu.
2. Click on the free-form surface and select **Event to Code**.

3. In the Event to Code window, select **readToDoFile()** from the **Method** list and then click **OK**. The free-form surface should look like this:



4. Save your current work in the Visual Composition Editor by selecting **Bean > Save Bean**. VisualAge for Java generates code to implement the connections you just specified.

You have now implemented the **Open To-Do File** button. Now you are ready to test the work you have done so far on the To-Do List program.

Testing the Open To-Do File button

Now that you have made all the connections for the **Open To-Do File** button, you are ready to test the work you have done so far.

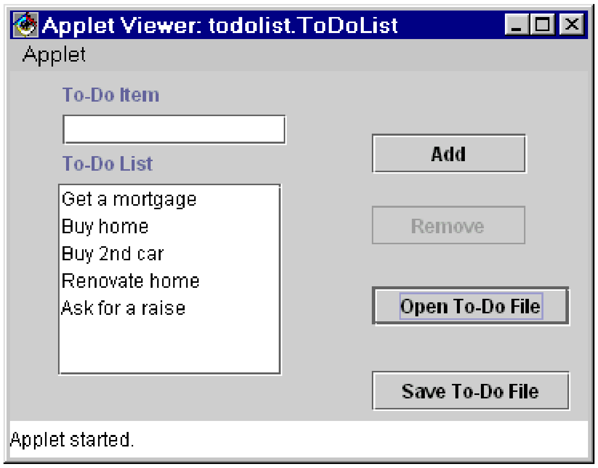
To test the current state of the To-Do List program:

1. Prepare a simple text file to use for testing. Use the Scrapbook window to create a file called todo.lst with the following lines in it:

   ```
   Get a mortgage
   Buy home
   Buy 2nd car
   Renovate home
   Ask for a raise
   ```

   Save the file in the c:\IBMVJava\IDE\project_resources\My ToDoList Project directory, where c:\IBMVJava is your VisualAge for Java installation directory.
2. Right-click the **ToDoList** class in the Workbench. From the pop-up menu, select **Run > Run main.** The To-Do List program appears.

   Important: You must run the file using **Run** > **Run main**. Do *not* click the Run 🏃 icon, because this will run the program as an applet, and the applet will not save or open the file properly.
3. Select the **Open To-Do File** button. The To-Do List in your program should now be loaded with the items from the todo.lst file:



Get a mortgage
Buy home
Buy 2nd car
Renovate home
Ask for a raise

Applet started.

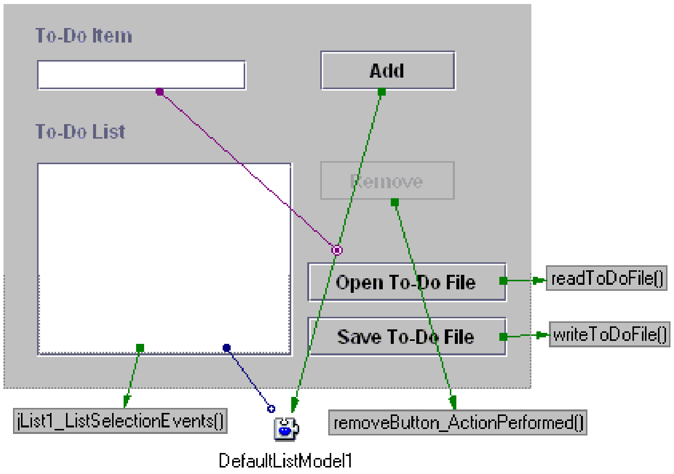
Now that you have tested your current progress on the To-Do List program, you are ready to complete the program by making the connections from the **Save To-Do File** button.

Connecting the Save To-Do File button

You are now ready to make the final connection from the **Save To-Do File** push button.

The To-Do List applet should call the writeToDoFile() method when a user selects the **Save To-Do File** push button. To implement this, you must connect the the **Save To-Do File** push button to the writeToDoFile() method.

1. Select the Save To-Do File button. From its pop-up menu, select **Connect** > **actionPerformed**.
2. Click on the free-form surface. From the pop-up menu that opens, select **Event to Code**.
3. In the Event to Code window, select **writeToDoFile()**. Click **OK**. The free-form surface should look like this:



Now that you have completed the connection from the **Save To-Do File** button, your To-Do File program is complete. You are ready to test it.

Saving and testing the enhanced To-Do List applet

To save and test your completed To-Do List program:

1. Select **Bean** > **Save Bean** to save your changes. VisualAge for Java generates the code to implement all the work you have done in the Visual Composition Editor since the last time you saved.
2. Right-click the **ToDoList** class in the Workbench. From the pop-up menu, select **Run > Run main.**
   Important: You must run the file using **Run** > **Run main.** Do NOT click the Run push button because this will run the program as an applet, and the applet will not properly save or open the file.
3. Try creating and saving a new To-Do file:
   - Add the following items to the To-Do List. For each item, enter the item in the **To-Do Item** field and select **Add**:
     ```
     Get paint
     Get wallpaper
     Spouse says OK?
     Start painting
     Start wallpapering
     ```
   - Select **Save To-Do File**.
4. Close the program, then rerun it.
5. Select **Open To-Do File**. The list that you entered previously should appear in the window.

You have now completed a Java program that combines a user interface created in the Visual Composition Editor with nonvisual code that you created directly.

Chapter 6. Managing the To-Do List applet edition

You have just reached a milestone in the development of your program, and you are ready to start coding some new features. Maybe you just want to explore a different implementation of a method that already works, but you are not sure if changes or additions will introduce new problems. This is a good time to create a versioned edition of your code.

With VisualAge for Java, you can manage multiple editions of program elements. You have already seen some of the concepts for managing editions. This section briefly reviews these concepts and shows you how to use the edition management features of VisualAge for Java.

[ENTERPRISE] Editions are managed by a team of developers. For information, refer to the team programming section of this document.

As you have been saving your program elements, VisualAge for Java has been keeping track of your code. In fact, the code on which you are working is saved in an edition. An *edition* is a "cut" or "snapshot" of a particular program element.

To see more information about the edition you are working on, from the Workbench window's toolbar click the Show Edition Names button. Each program element includes either an alphanumeric name or a time stamp beside it; this is the edition information (described below in more detail). You can also see edition details from the Source pane. For example, select your ToDoList class and move the mouse pointer over the applet icon in the **Source** pane's title bar. The hover-help window displays the edition information. The same information is also displayed in the status area below the **Source** pane.

An edition of a program element keeps track of all code within that program element, and all the other program elements that it contains. For example, an edition of a package includes classes and interfaces and the methods within these classes and interfaces.

At any time, the workspace contains only one edition of a given program element: the edition that you are currently working on. VisualAge for Java also includes a source code repository, which contains all editions of the same program element. The workspace is the center of activity in the VisualAge for Java programming environment. The repository is not a development environment, but you can browse and retrieve its contents as needed. You can save as many editions of a program element as you wish. All editions are stored and are accessible from the repository.

There are two fundamental types of editions:

Open edition.
: An open edition of a program element can be modified. You can bring this edition into the workspace, making it the current edition, and change it as required. Open editions are marked by time stamps. For example, `(7/9/98 8:44:52 AM)` is an open edition.

Versioned edition.
: A versioned edition of a program element cannot be changed. When you

version an edition, you establish a frozen (read-only) code base to which you can revert any time. Versioned editions are designated by alphanumeric names (for example, 1.0).

The edition that is in the workspace may be a versioned edition, although any changes you make and save automatically create a new open edition.

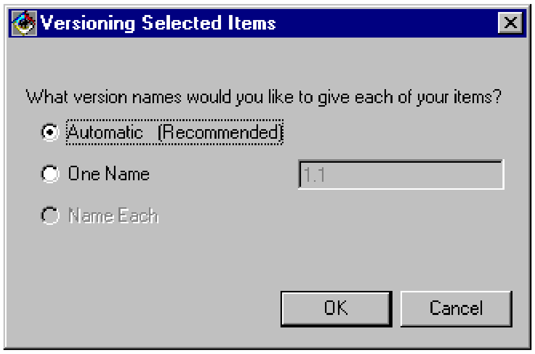
Every time you save a program element, your code is incrementally compiled, and the open edition is updated in both the workspace and the repository.

Versioning the applet

You can version a project, a package, or a class. When you version one of these program elements, all program elements contained within it are also versioned. For example, if you version a package, all classes that are part of that package are also versioned.

Create a versioned edition of your code:

1. Select the package in which you created your To-Do List applet. From the pop-up menu, select **Manage** > **Version**. The Versioning Selected Items SmartGuide opens.



Versioning Selected Items

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

● Automatic (Recommended)

○ One Name 1.1

○ Name Each

2. Ensure the **Automatic** radio button is selected and then select **OK**.

In the Workbench hierarchy, notice that the time stamp beside the package name has been replaced with the new version number. This versioned edition is now permanently stored in the repository.

Updating the To-Do List code again

Now that you have a versioned edition of your program in the repository, you can change your program elements in the workspace with the assurance that you can always revert to the versioned edition.

Because a versioned edition cannot be modified, you will need to create a new open edition from the versioned edition before you can continue changing the program element. If the edition in the workspace is the versioned edition, a new edition is automatically created for you if you make changes to the program element and then save it. For example:

1. Select your **ToDoList** class in the Workbench, and type a new comment in the Source pane.
2. From the pop-up menu in the Source pane, select **Save**.

Notice that the edition name changes from the versioned edition name to a time stamp. Because the workspace can only hold one edition of a program element at any given time, the new edition replaces the versioned edition. (Of course, a copy of the versioned edition can always be retrieved from the repository.)

You can replace an edition that is in the workspace with another edition from the repository. Note that the current edition is always marked by an asterisk (by default) to the left of the edition name when you browse an edition list in the repository.

Follow these steps to browse the Repository Explorer and view your versioned edition of the ToDoList class:

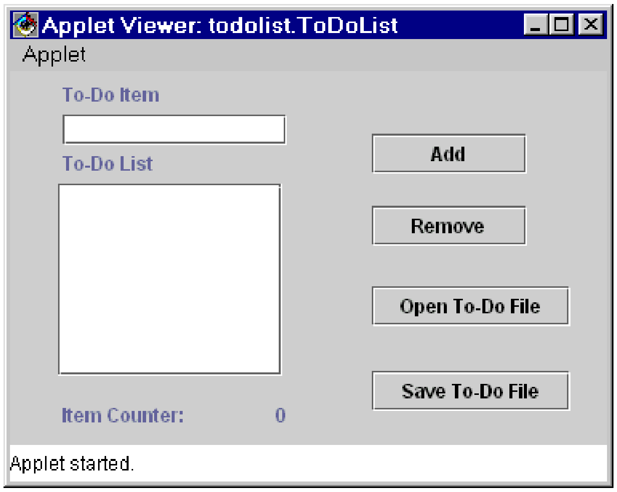
1. In the Workbench, select **Window > Repository Explorer**.
2. Click the **Projects** tab. In the Project Names column, select **My ToDoList Project**.
3. In the Editions column, select the current edition, which will be marked by an asterisk *(4/14/2000 1:56:2 , and then select the todolist package in the Package column.
4. The versioned copy of the ToDoList class you just created will appear in the Types column.

Adding a counter to the To-Do List applet

Now add a counter to the To-Do List applet, which will reflect the number of items in the To-Do list at any given time. To add this feature, you need to change the applet as follows:

1. Add labels for the counter name and the counter itself.
2. Connect the **Add**, **Remove**, and **Open To-Do File** push buttons to the counter label.

When modified, the running applet will look like this:



Applet Viewer: todolist.ToDoList

Applet

To-Do Item

To-Do List

Item Counter: 0

Applet started.

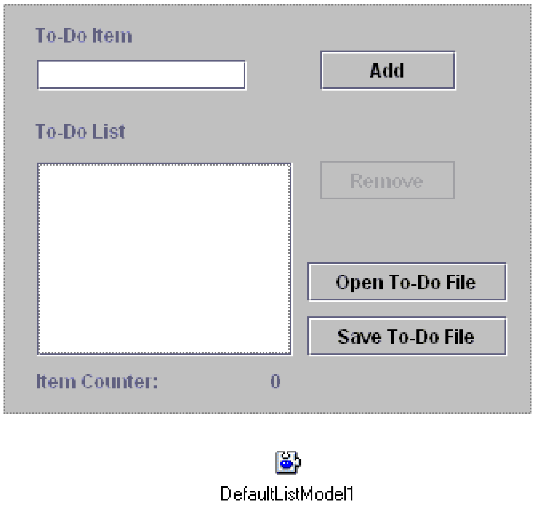
Adding counter labels to the To-Do List applet

To add the two labels using the Visual Composition Editor:

1. Right-click the **ToDoList** class in the Workbench.

2. From the pop-up menu, select **Open To > Visual Composition**. This opens the ToDoList class in the Visual Composition Editor.
3. To make it easier to create the new connections, hide the existing connections by selecting **Hide Connections** from the toolbar:
4. Select a JLabel bean from the palette.
5. Click beneath the list to add the label. You may wish to select the scroll pane, the text field and the labels and move them slightly up to make room for the new label.
6. Modify the text of the JLabel bean to `Item Counter:`
7. Add another JLabel bean to the right of the JLabel bean you just added.
8. Double-click on this new JLabel bean to open its Properties window. Select the value field to the right of the **horizontalAlignment** field. From its pull-down menu, select **RIGHT**, which right-justifies the value. In the **text** field, change the value to **0**, which is the initial value of the counter. Close the Properties window.

The visual beans have been added. The free-form surface should look like this:



Now you are ready to add the connections.

Note: All of the other connections you made are still there, they are just hidden now because you selected **Hide Connections** from the toolbar. The new connections that you make in the next step will not be hidden.

From the **Bean** menu, select **Save Bean**. The changes you've made are reflected in this open edition, both in the workspace and in the repository.

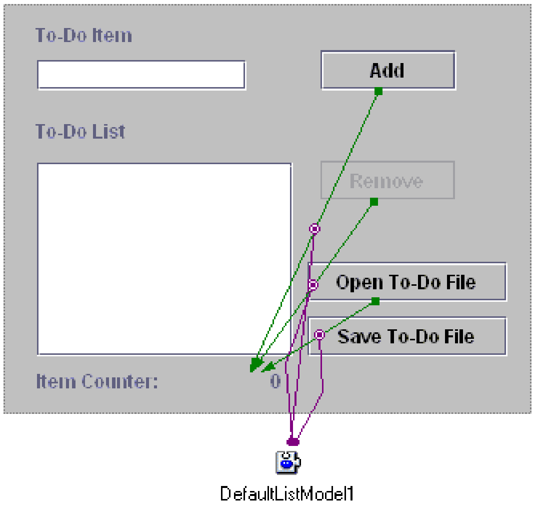
Connecting the To-Do List counter labels

To connect the **Add** push button to the counter:

1. Select the **Add** push button and from its pop-up menu select **Connect** > **actionPerformed**.
2. Position the mouse over the counter and click.
3. From the pop-up menu, select **text**. A dashed green line now appears, indicating an incomplete connection.

4. Select the connection and select **Connect** > **value** from its pop-up menu.
5. Position the mouse over the DefaultListModel bean and click.
6. From the pop-up menu, select **Connectable Features** to bring up the End connection to (DefaultListModel1) window.
7. From the **Property** list, select **size** and then click **OK**. This provides the count of the list of items as input for setting the counter string. The connection is now complete.
8. Connect the **Remove** and **Open To-Do File** push buttons in the same manner. You're simply updating the count of items in the list whenever an action is taken that may modify the count. In this applet, any of the top three buttons have this potential.

   Now the free-form surface should look like this:

9. Select **Bean > Save Bean**. Right-click the **ToDoList** class in the Workbench. From the pop-up menu, select **Run > Run main.**

   Important: You must run the file using **Run** > **Run** main. Do NOT click the

   Run button because this will run the program as an applet, and the applet will not save or open files properly.
10. Test the To-Do List applet to make sure the counter works by adding and removing items from the To-Do List.

Returning to a previous edition of the To-Do List

Your program now contains a counter. It works fine, but after thinking about it for a while, you decide that you want to keep the interface as clean as possible - no bells and whistles. So, now you want to take out the counter code. Of course, you could just delete the labels and connections you've added, but you might inadvertently delete one of the other program elements or connections. Do not worry: Remember, you versioned the previous edition!

Follow these steps to replace the current edition with a previous edition from the repository:

1. Select the **ToDoList** class in the Workbench.
2. From its pop-up menu, select **Replace With** > **Another Edition**.

3. From the **Select replacement for ToDoList** window, select the edition that you previously versioned and select **OK**.

   (Because you want to replace the current edition with the previous edition, you could have also selected **Replace With** and then **Previous Edition** from the pop-up menu.)

The edition information beside the class name now indicates the version number, not the time stamp of the open edition you had been working on.

If you change your mind again and decide that the counter should stay, you can always add it back; the edition that contained the counter is still in the repository.

Debugging

Now that you have finished developing your applet, you can debug it with the *IDE integrated debugger.* You can use the integrated debugger to debug applets and applications running in the IDE.

Opening the debugger
You can open the debugger manually by selecting **Window > Debug > Debugger**. If a program is running, you can suspend its thread and view its stack and variable values. The debugger will also open automatically, with the current thread suspended, for any of several reasons:

- A breakpoint in the code is encountered.
- A conditional breakpoint that evaluates to true is encountered.
- An exception is thrown and not caught.
- An exception selected in the Caught Exceptions dialog is thrown.
- A breakpoint in an external class is encountered.

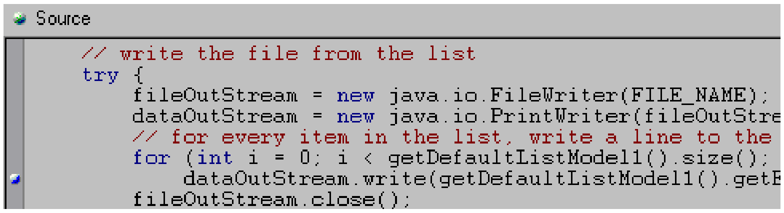
Setting breakpoints
When a program is running in the IDE and encounters a breakpoint, the running thread is suspended and the Debugger browser is opened so that you can work with the method stack and inspect variable values. In the IDE, you can set breakpoints in any text pane that is displaying source. Suppose that you want to set a breakpoint in the writeToDoFile() method in the ToDoList class from the To-Do List program.

To set this breakpoint:

1. Select the **ToDoList** class in the Workbench. Expand the class to show its methods.
2. Select the writeToDoFile() method. The source for the method is shown in the Source pane.
3. Place the cursor in the following line and select **Edit** > **Breakpoint**:

   ```
   dataOutStream.write(getDefaultListModel1().getElementAt(i) + crlf);
   ```

   Click **OK** to close the Configuring: Breakpoint window.
4. A breakpoint indicator appears in the margin of the Source pane beside this line:



```
// write the file from the list
try {
    fileOutStream = new java.io.FileWriter(FILE_NAME);
    dataOutStream = new java.io.PrintWriter(fileOutStre
    // for every item in the list, write a line to the
    for (int i = 0; i < getDefaultListModel1().size();
        dataOutStream.write(getDefaultListModel1().getE
    fileOutStream.close();
```

You can also set a breakpoint on a line that does not already have a breakpoint by following these steps:

1. Move the cursor to the line.
2. Right-click and select **Breakpoint** from the pop-up menu.

If you wanted to make this a conditional breakpoint (one that suspends the thread under only certain conditions), you could do so by right-clicking the breakpoint indicator, selecting **Modify**, and entering the expression in the field. See the online help for more information on configuring breakpoints.

Removing breakpoints

To remove a breakpoint in a source pane, double-click the breakpoint indicator. You can also remove a breakpoint by following these steps:

1. Select the breakpoint.
2. Select **Clear** from its pop-up menu.

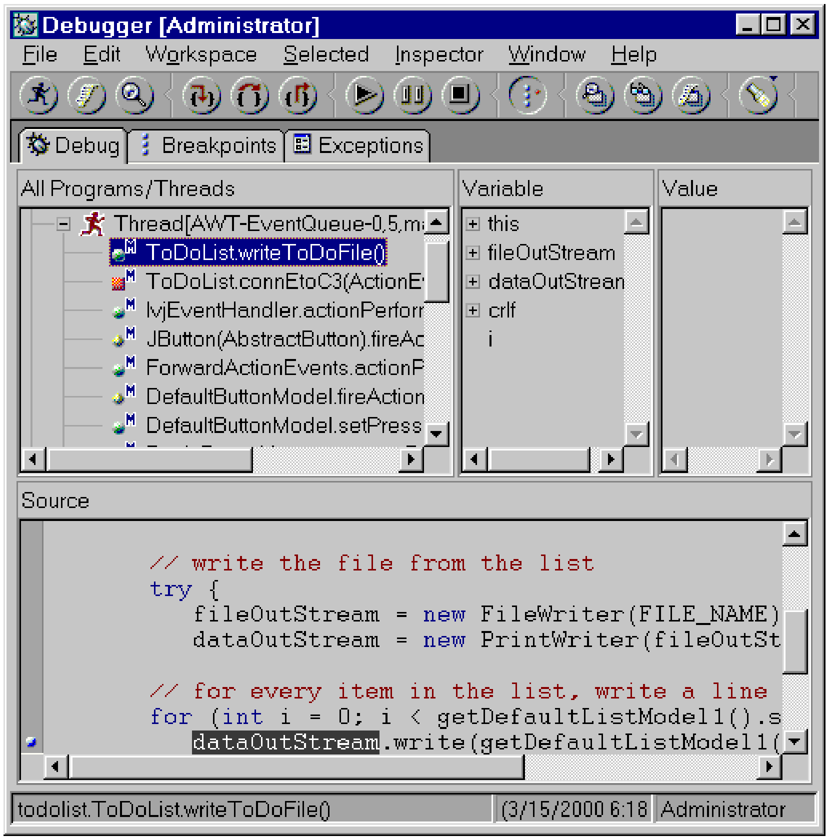
Try removing the breakpoint you just set. Now reset it. You will be using this breakpoint in the next section to examine the features of the Debugger browser.

Using the Debugger browser

The Debugger browser opens automatically when the program you are executing reaches an active breakpoint or has an unhandled exception.

Now that you have set a breakpoint, run the To-Do List program to see what happens:

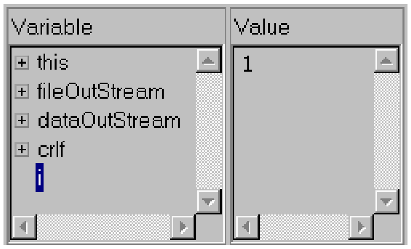
1. In the Workbench, make sure that breakpoints are enabled:
   - From the **Window** menu, select **Debug** and then **Breakpoints**.
   - Ensure that the Global Enable Breakpoints button on the toolbar is in the down (enabled) position. If not, click it.
   - Close the Debugger window.
2. Right-click the **ToDoList** class in the Workbench. From the pop-up menu, select **Run > Run main.**
   Important: You must run the file using **Run > Run main**. Do *not* click the Run button because this will run the program as an applet, and the applet will not save or open the file properly.
3. When the To-Do List program appears, add at least three items to the To-Do List and then select **Save To-Do File**. The Debugger browser window opens. It should look like this:


Debugger [Administrator]
File
Edit
Workspace
Selected
Inspector
Window
Help
Debug
Breakpoints
Exceptions
All Programs/Threads
Thread[AWT-EventQueue-0,5,m
ToDoList.writeToDoFile()
ToDoList.connEtoC3(ActionE
IvjEventHandler.actionPerforr
JButton(AbstractButton).fireAc
ForwardActionEvents.actionP
DefaultButtonModel.fireAction
DefaultButtonModel.setPress
Variable
this
fileOutStream
dataOutStrean
crlf
i
Value
Source
todolist.ToDoList.writeToDoFile()
(3/15/2000 6:18
Administrator

```
    // write the file from the list
    try {
        fileOutStream = new FileWriter(FILE_NAME)
        dataOutStream = new PrintWriter(fileOutSt

    // for every item in the list, write a line
    for (int i = 0; i < getDefaultListModel1().s
        dataOutStream.write(getDefaultListModel1(
```

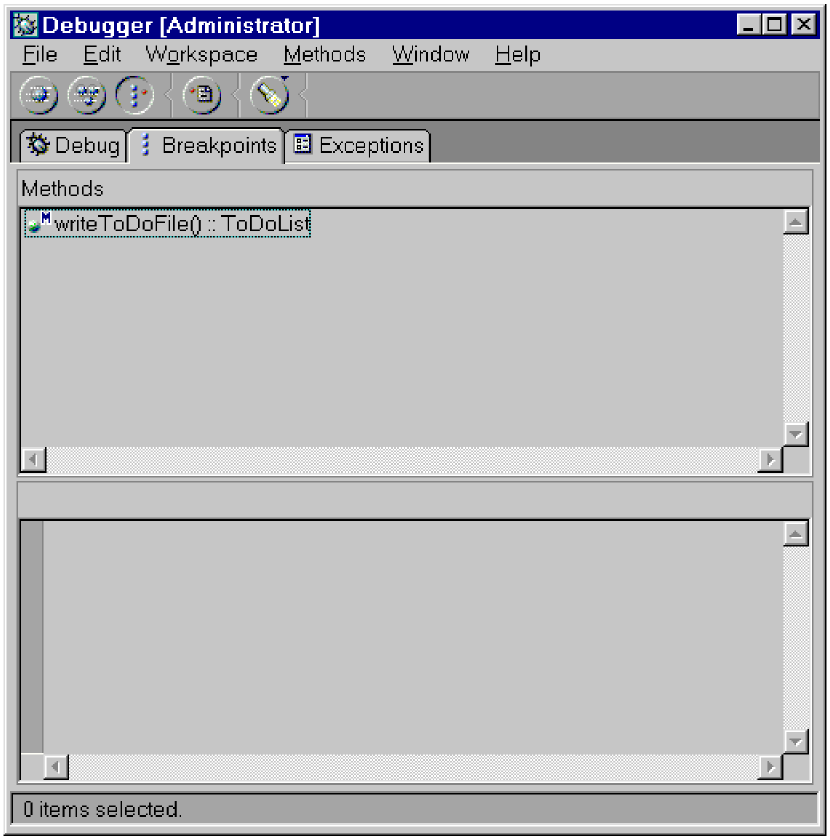
The thread that you are debugging is selected in the All Programs/Threads list. The list of methods below the thread is the current stack. When you select a method in the stack the Variable pane shows its visible variables. The Source pane shows the source where the breakpoint is set.

4. Select the **Resume** button on the toolbar to continue execution of the program. Because this breakpoint is inside a loop that writes each item to the file, the thread is suspended again and the Debugger window displays it.
5. Examine some of the variables in the Variable list. For example, to see the value of the loop counter variable `i`, select **i** from the Variable pane (it is at the bottom of the list). Its value appears in the Value pane:



This value of the loop counter is exactly what you would expect, because the loop has been run once.

6. Now disable this breakpoint:
   - Select the **Breakpoints** tab in the Debugger browser. The Breakpoints page appears:



writeToDoFile() :: ToDoList

0 items selected.

- The Breakpoints page lists all the breakpoints that you have set in the workspace. The Methods pane lists all the methods in which you have set breakpoints. The Source pane displays the source for the method that is selected in the Methods pane.

- To disable your breakpoints, click the **Global Enable Breakpoints** button on the toolbar so that it is in the up (disabled) position. The breakpoint indicator changes colors to show that it is disabled. Note that it is not removed, but it will be ignored when the program resumes.
- Select the **Debug** tab to return to the Debug page.

7. Now select **Resume** from the toolbar to continue execution of the program. In the To-Do List program, first remove the values you added in step 3, then add the following values to the To-Do List and then select **Save To-Do File** to save them to a file:

   ```
   item A
   item B
   item C
   last item
   ```

8. Remove all the items from the To-Do list. Now select **Open To-Do File** and open the file you just saved. The **To-Do List** should look like this:

   ```
   item A
   item B
   item C
   last item
   ```

Before you continue, return to the Breakpoints page and enable your breakpoints again by clicking the **Global Enable Breakpoints** button into the down position.

More tasks you can perform with the integrated debugger

The debugger has many other features that you will find helpful for debugging your programs. To learn more about the following tasks, as well as others, see the online help.

Setting external and caught exception breakpoints
Besides setting breakpoints in code in the workspace, you can also set breakpoints on methods in external classes (classes that reside outside the workspace, in the file system, and that are loaded at run time). You can also specify exception types that will break execution if they are thrown, even if your code catches and handles them.

Stepping through code
When a running thread has been suspended, you can step through code line by line or method by method, in a variety of ways. This is a controlled way of checking variable values at each point in your program.

Using inspectors to view and modify variables
You can open an inspector window to look closely at a particular variable in a suspended thread. The inspector lets you view and modify variable values and evaluate expressions.

Chapter 7. Continuing applet coding in a team

Earlier you created a To-Do List program. Now, a team of six developers will develop a more complicated version of it. See “Team programming” on page 83 for the structure of the To-Do List team.

Team programming is only available with the Enterprise Edition of VisualAge for Java.

Each team member’s responsibilities are determined by assigning ownership of program elements. See “The To-Do List team roles” on page 84 for how the ownership of objects in the team repository maps to the roles of the To-Do List team members.

The team works together on the To-Do List program. See the “The To-Do List team development cycle” on page 87 for how the team works together to develop a product.

The To-Do List team held their first meeting this week. Tasha volunteered to set up the server that will contain the shared repository, and to be the repository administrator. Tasha has decided that the server will be called todoserv and that the team will use a new repository called todoteam.dat for their project.

Setting up the server for the To-Do List team

Before the team can share source code, you must set up the server, start the server, and configure the client workstations to connect to the server.

Each VisualAge for Java team server has the following components:

- At least one shared repository
- A program called EMSRV, which manages the client connections to the shared repository

For detailed information about setting up the team server, refer to the “Server setup and administration file” (emsrv70.htm) which is on the product CD. Refer to the README in the root directory of the CD for the exact location of the Server setup and administration file.

To set up the repository server, follow these steps:

1. Confirm that your server has an active connection to your TCP/IP network. (You will need to know the host name or IP address of your server later, when you configure the client workstations.) The host name of Tasha’s server is todoserv.
2. If you have not already done so, install the VisualAge for Java server software from the product CD. The server installation program will install the EMSRV program in the directory that you specify, and it will create a repository called ivj.dat in the same directory. (You will create a second repository later) Instructions on installing the server software can be found in the Installation and Migration guide on the product CD. Please refer to the README file for the exact location of the guide.

Starting and stopping the server

After the VisualAge for Java server has been installed, you can start the EMSRV program, which will manage the client connections.

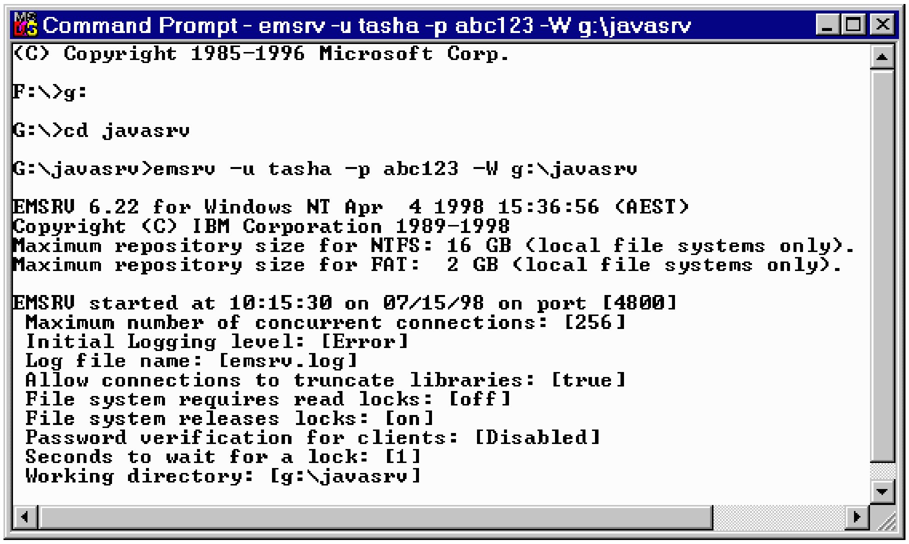
1. The user who starts and stops the EMSRV repository server is called the *EMSRV user*. In Windows NT, the EMSRV user must belong to the Administrators user group and must have the NT advanced user right, “Act as part of the operating system”. The EMSRV user requires access to the following files:
   - The shared repositories that EMSRV manages on the server
   - The EMSRV log file (automatically created and owned by the EMSRV user)
   - The passwd.dat file, if you are using it for password validation

   Tasha has decided that she will start the server with her own user ID. She is already a member of the Administrators group in Windows NT and she has added the required authorizations. To start the server, use the **emsrv** command. In this exercise, you will only use the basic options to start the server:

   EMSRV -u *userid* **-p** *password***-W** *working directory*

   where *userid* is the EMSRV user’s ID, *password* is the EMSRV user’s password, and *working directory* is the server directory where the ivj.dat repository file resides.

   The following example shows the command that Tasha enters to start the server. Her password is abc123, and she has installed the VisualAge for Java server files in a directory called javasrv. The messages confirm that the server has started successfully.



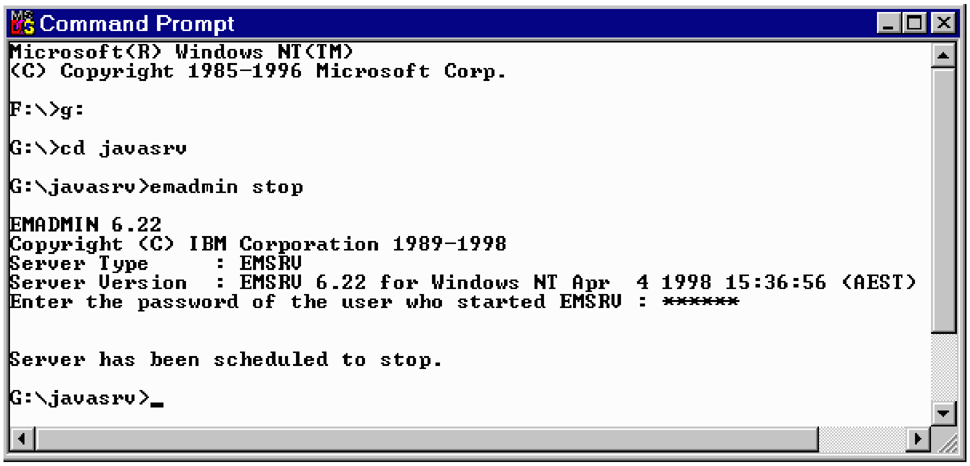
```
(C) Copyright 1985-1996 Microsoft Corp.

F:\>g:

G:\>cd javasrv

G:\javasrv>emsrv -u tasha -p abc123 -W g:\javasrv

EMSRV 6.22 for Windows NT Apr  4 1998 15:36:56 (AEST)
Copyright (C) IBM Corporation 1989-1998
Maximum repository size for NTFS: 16 GB (local file systems only).
Maximum repository size for FAT:  2 GB (local file systems only).

EMSRV started at 10:15:30 on 07/15/98 on port [4800]
 Maximum number of concurrent connections: [256]
 Initial Logging level: [Error]
 Log file name: [emsrv.log]
 Allow connections to truncate libraries: [true]
 File system requires read locks: [off]
 File system releases locks: [on]
 Password verification for clients: [Disabled]
 Seconds to wait for a lock: [1]
 Working directory: [g:\javasrv]
```

   At this point, you should start the EMSRV program on your own server, with a user ID that has the required authorization. The “Server setup and administration” file (emsrv70.htm) on the product CD provides detail instructions on the following items:
   - Authorizing the EMSRV user
   - The EMSRV command

- Starting EMSRV as a Windows NT service

Refer to the README in the root directory of the CD for the exact location of the Server setup and administration file.

Should you need to stop the repository server, the command to issue is **emadmin stop**. Issuing this command will stop EMSRV after prompting you for the EMSRV user's password, as shown below.



```
Microsoft(R) Windows NT(TM)
(C) Copyright 1985-1996 Microsoft Corp.

F:\>g:

G:\>cd javasrv

G:\javasrv>emadmin stop

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.

G:\javasrv>_
```

If you have stopped EMSRV, restart it before you continue with the next section.

Connecting a client to the todoserv server

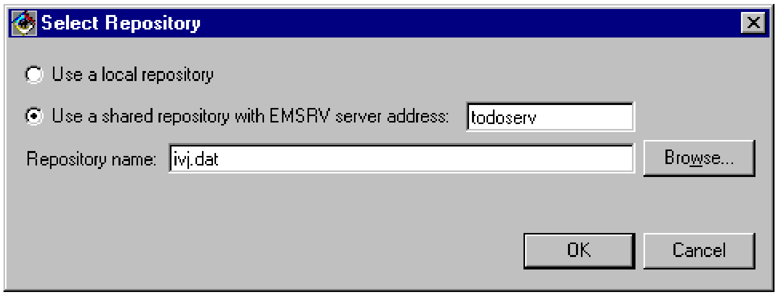
Now that the server is up and running, Tasha would like to connect from her workstation to the ivj.dat repository on the todoserv team server that she just set up.

Note that when you connect to a server, *you must know its host name*. Once you connect to that server, VisualAge for Java will show you a list of the repositories available.

In the following steps, it is assumed that you have already been using the VisualAge for Java IDE on your workstation. You have probably been working with a local repository on your own hard drive.

To connect your workspace to the server's repository, do the following:

1. Verify that you have an active TCP/IP connection on the same network as your server. Tasha does this by entering `ping todoserv` from a command prompt. She receives a reply from the server, so she knows her network connection is good.
2. From the Workbench window of the IDE, select **Window** > **Repository Explorer.** The Repository Explorer window opens.
3. Select **Admin** > **Change Repository**.
4. Select **Use shared repository**. Enter the host name or IP address of your server in the **EMSRV server address** field. Enter ivj.dat in the **Repository name** field. Click **OK**.



The previous picture shows the information that Tasha provides to connect to the ivj.dat repository on the todoserv server. Because she specified a working directory when she started the server, EMSRV will automatically locate ivj.dat in that directory. Otherwise, Tasha could click **Browse** to find the repository on the server now.

5. If you are the first person who has ever connected to this repository on this server, you will be prompted to provide a network name for Administrator. Administrator is the only VisualAge for Java user who can add other users to the repository.

   If you choose to enable password validation later, VisualAge for Java will use Administrator's network name to validate the administrator's login. Tasha plans to enable password protection later, so she enters her Windows NT user ID. In these sections, you will work without password validation, so it does not really matter what you enter as Administrator's network name.

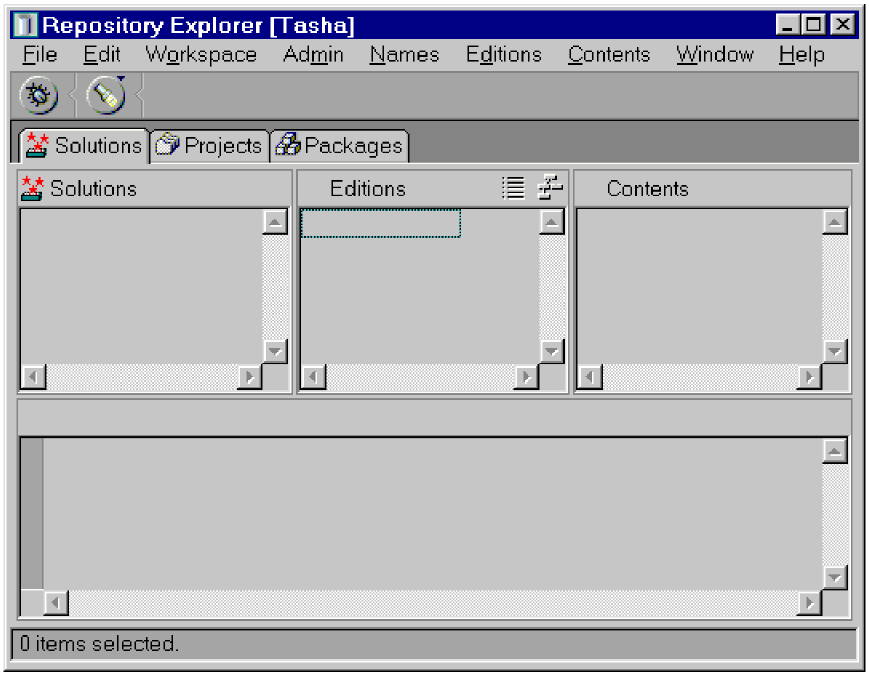
   For more information refer to the "Server setup and administration" file (emsrv70.htm), which is on the product CD. Refer to the README in the root directory of the CD for the exact location of the Server setup and administration file.

6. A progress indicator confirms that VisualAge for Java is connecting your workspace to the ivj.dat repository on the server. When the connection is complete, the Repository Explorer window shows you the projects that are in the ivj.dat repository on the *server*.

Creating the To-Do List repository

In order to create a new repository for her team and set up VisualAge for Java user IDs for them, Tasha will have to work as the repository administrator. Tasha knows how VisualAge for Java projects are organized, and which responsibilities have been assigned to each developer on the team.

1. In the following picture, Tasha's name appears in the title bar of the Repository Explorer window. She is the current *workspace owner*. That is why her name appears in the window's title bar.


Repository Explorer [Tasha]
File Edit Workspace Admin Names Editions Contents Window Help
Solutions Projects Packages
Solutions
Editions
Contents
0 items selected.

If you do not see *Administrator's* name in the title bar of your VisualAge for Java windows, you will need to change to Administrator before you can continue with this section. In the Repository Explorer window, select **Workspace** > **Change Workspace Owner**. A window containing a list of users for this repository opens. Select **Administrator** from the list and click **OK**. Administrator's name should now appear in the window's title bar.

Adding the To-Do List team repository to the server

Tasha promised her team that she would create another repository, called todoteam.dat, just for them. She will do this using a process called *compacting*, which copies the existing repository. (Specifically, compacting copies versioned elements from the existing repository, but does not copy open or purged elements.)

If you have several large software projects, you might choose to give each one a repository and let each team manage its own repository. However, if you have only a few small projects, continue using only ivj.dat.

In this section, you will learn how to create a new repository called todoteam.dat. You will use this repository as you work on the team development scenarios in this tutorial. Once you have finished all of the team sections, you can delete the todoteam.dat file.

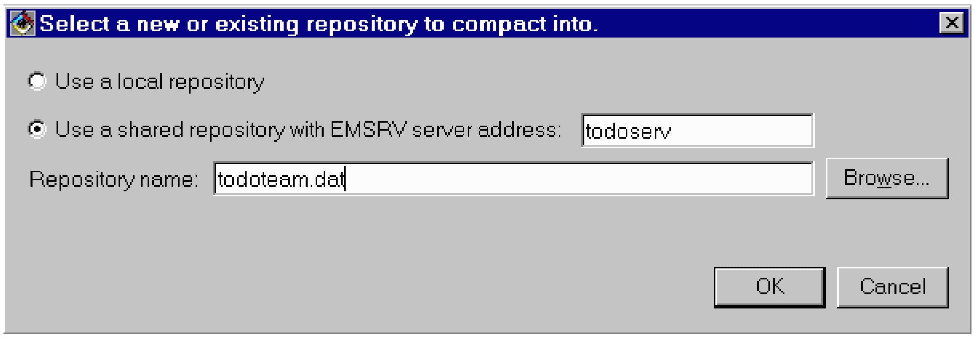
Also in this section, you will add users to the todoteam repository. Creating repositories and adding users to repositories are two functions that only Administrator can do. You will learn more about the role of the Administrator later.

Creating a new repository

To create a repository, follow these steps:

1. Connect to the ivj.dat repository on your server as shown in an earlier section.
2. Confirm that Administrator is the current owner of your workspace.
3. Switch to the Repository Explorer window.

4. Select **Admin** > **Compact Repository**.
5. When warned that only versioned elements will be copied to the new repository, click **Yes** to continue.
6. As when you first connected to the server, you need to specify the host name of the server where the new repository will be created. Tasha enters todoserv; you must enter the name of your own server. You also need to enter the name of the repository that you want to create, todoteam.dat.



Select a new or existing repository to compact into.

○ Use a local repository

● Use a shared repository with EMSRV server address: todoserv

Repository name: todoteam.dat

7. Click **OK** to start the process of repository creation. When prompted to confirm that you wish to create a new repository, click **Yes**. It may take a few minutes for the new repository to be created.

Because you have created the new repository by compacting an existing one, todoteam.dat will contain copies of all versioned elements from ivj.dat. For information about other ways to create a repository, refer to the online help.

Connecting to the To-Do List team repository

Before you can use the new repository, you must connect to it by following these steps:

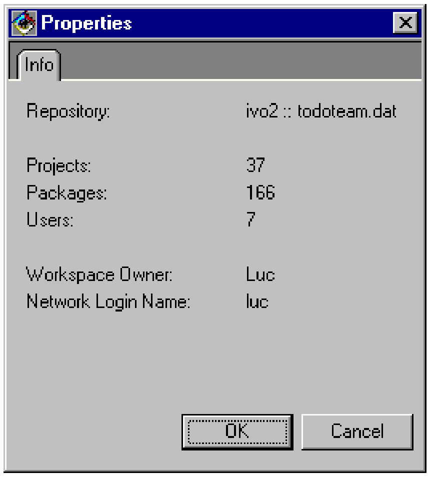
1. From the Repository Explorer window's **Admin** menu, select **Change Repository.**
2. Click **Browse** to see a list of repositories (.dat files) in the working directory for your server.
3. Select **todoteam.dat** from the list. If prompted, select **Administrator** as the workspace owner.

Verifying which repository you are connected to
If, at any time, you want to confirm which repository you are connected to, you can do so in the **Properties** window.

To view information about the repository you are connected to, perform the following steps:

1. In the Repository Explorer select **Admin > Properties**. The repository status appears in the Properties window. The repository name is qualified by the server name. In the picture below, ivo2 is the server name and todoteam.dat is the name of the repository. Luc's name appear in the Workspace Owner field.

   When you open the Properties window, the name of the server and repository you are connected to will appear in the repository field, and the name of the user who owns the workspace will appear in the Workspace Owner field.



Repository: ivo2 :: todoteam.dat

Projects: 37
Packages: 166
Users: 7

Workspace Owner: Luc
Network Login Name: luc

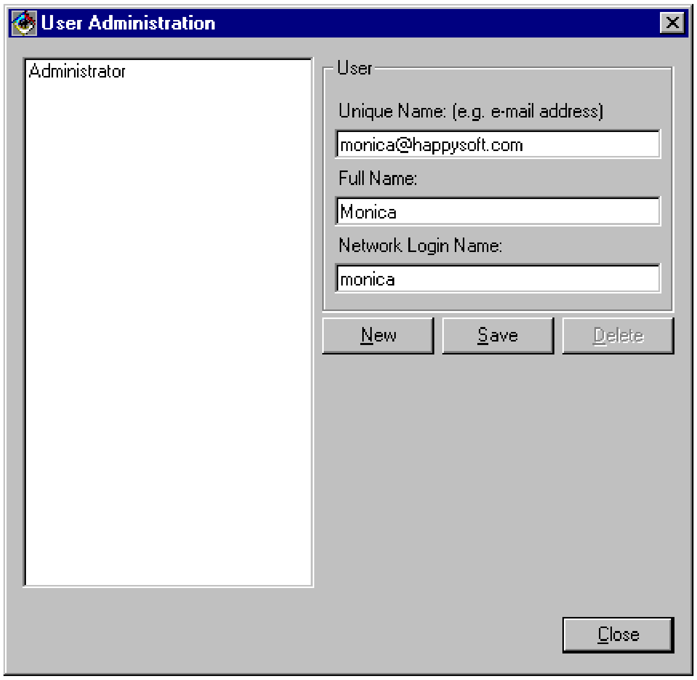
2. Click **OK** to close the Properties window.

Adding users to the To-Do List team repository

Tasha has to do one more thing before the team can use the repository. She needs to add a list of users who will be working with the todoteam.dat repository. Specifically, she needs to add: Monica, Tasha, Luc, Margaret, Sam, and Susan.

Like Tasha, you will create a list of users for the todoteam repository before you can continue with this section.

1. Select **Admin > Users**. The User Administration window opens, showing the repository users. Because this is a new repository, Administrator is the only user. You will create the others.
2. To add Monica as a user, click **New**. Enter the same information you see in the picture below.



Unique Name: (e.g. e-mail address) monica@happysoft.com
Full Name: Monica
Network Login Name: monica

The Unique Name is the name that VisualAge for Java actually uses to identify the user. The Full Name is the name that VisualAge for Java displays in windows and lists. The Network Login Name is used for password protection, which you will not be doing in this section. For information on password protection, refer to the online help or the “Server setup and administration” file (emsrv70.htm) on the product CD. Refer to the README in the root directory of the CD for the exact location of the Server setup and administration file.

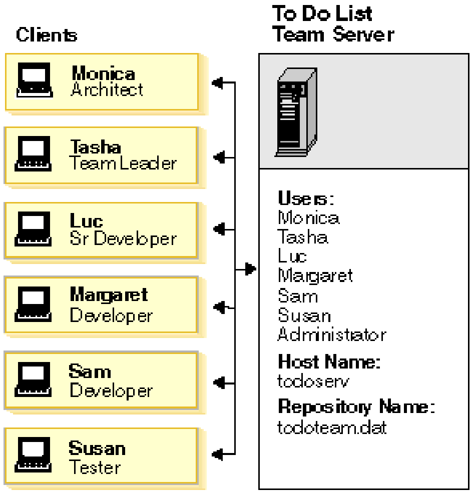
Click **Save.** Monica is added to the user list.

3. Add Tasha, Luc, Margaret, Sam, and Susan in the same way.
4. Click **Close** when you have finished adding the six members of the team.
5. Now that you have finished setting up the users for the new repository, you should work under your own name instead of the Administrator’s. *In the team environment, Administrator should only perform repository administration tasks, not programming.*

To identify yourself to the server as Tasha, from any menu bar of any VisualAge for Java window, select **Workspace > Change Workspace Owner**. Select Tasha from the list of users.

Server ready for team development
The To-Do List team server has been set up and started. The To-Do List team repository, todoteam.dat, has been created. The users of that repository are: Monica, Tasha, Luc, Margaret, Sam, Susan, and Administrator. In the following sections, you will be connecting to the repository as these various users, so you should verify now that you can connect as any one of them.

You are now ready to begin team development with the setup shown in the following picture.



For detailed information about setting up the team server, refer to the “Server setup and administration” file (emsrv70.htm), which is on the product CD. Refer to the README in the root directory of the CD for the exact location of the Server setup and administration file.

Owning To-Do List code

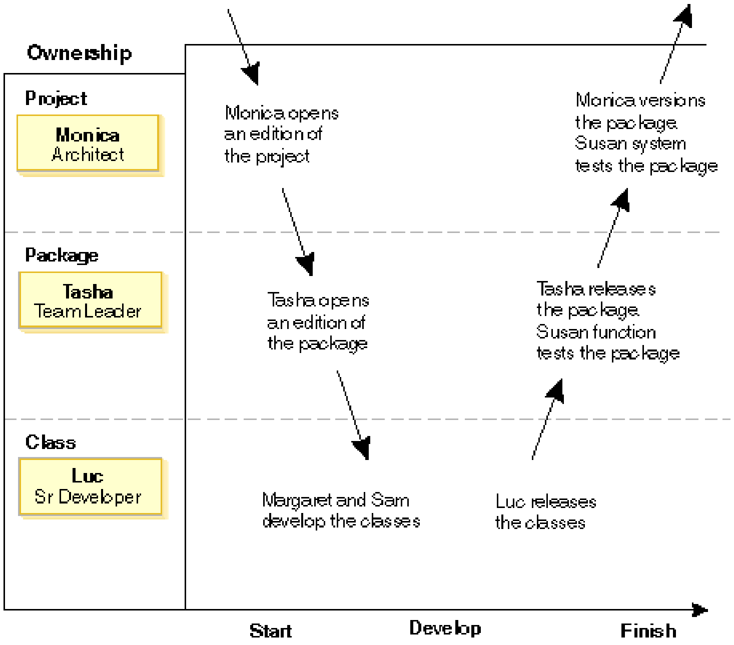
During the To-Do List team's second meeting, Tasha informed them that the server had been set up, but two things needed to be done before coding could begin. Owners needed to be assigned to program elements, and these owners would have to open editions of the program elements for the rest of the team.

The To-Do List team members are assigned various roles. Monica is the project owner, Tasha is the package owner and Luc owns the classes. All of the team members except Monica become package group members.

Project and package owners have three main responsibilities:

1. They create open editions of program elements to allow the team to make changes to them.
2. They release their editions to update the team baseline.
3. They version their editions to preserve a team baseline.

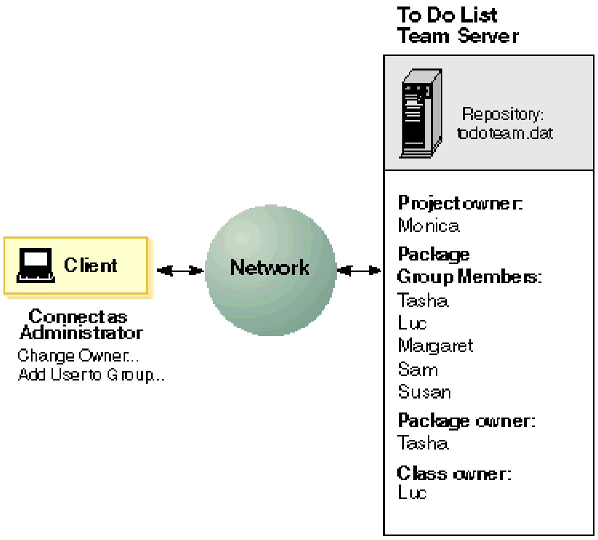
As shown in the next diagram, Monica opens an edition of the To-Do List project. The project can only be opened by the project owner. Similarly, Tasha opens the package. When Monica and Tasha open the project and package editions, everyone can use them. If Monica and Tasha do not open editions, then the team members can only work with private editions called scratch editions. VisualAge for Java enables owners of the code to decide when they want their code open or closed for public development.


Ownership
Project
Monica
Architect
Package
Tasha
Team Leader
Class
Luc
Sr Developer
Monica opens an edition of the project
Tasha opens an edition of the package
Margaret and Sam develop the classes
Luc releases the classes
Tasha releases the package. Susan function tests the package
Monica versions the package. Susan system tests the package
Start
Develop
Finish

Since Java development is iterative, opening editions and versioning projects will usually happen several times during the development cycle, as the team sets new baselines.

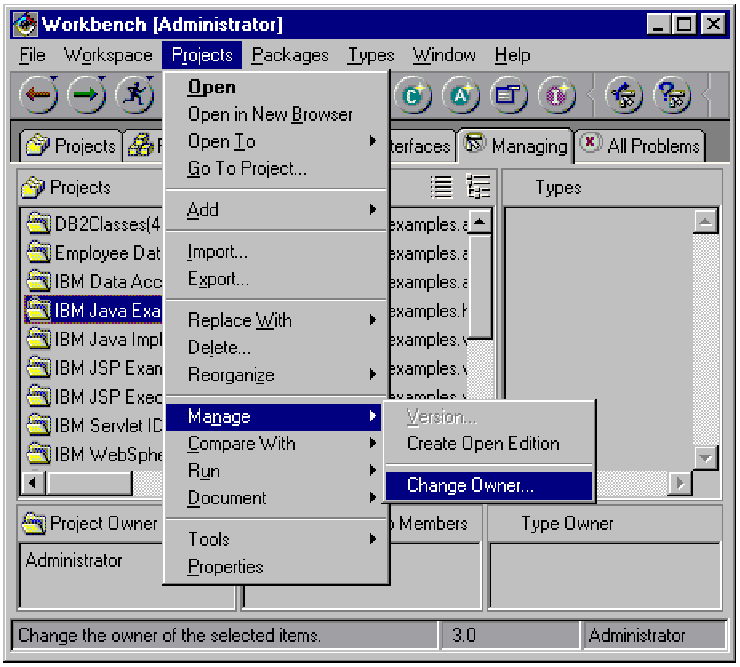
Assigning team roles for the To-Do List project

Now, working as Administrator, you will add a project from the shared repository to your workspace, assign ownership of various program elements, and add members to a package group.



The To-Do List project is found in the IBM Java Examples project. To assign ownership of the project and its contents, follow these steps:

1. Connect to the todoteam.dat repository as Administrator.
2. If the IBM Java Examples project is not already in your workspace, add it from the shared repository now. Select the Projects tab in the Workbench. From the **Selected** menu, select **Add > Project**. Select **Add projects from the repository**. Select **IBM Java Examples** from the available project names pane and the latest edition from the available editions pane. Click **Finish**. The project is added to your workspace.
3. In the Workbench window, click the **Managing** tab. The Managing page provides a management view of the project. The code, the owners of the code, and the package group members are displayed.
4. In the Projects pane of the Managing page, click **IBM Java Examples**. The Project Owner pane shows who currently owns the project: **Administrator**. You will change the owner to Monica.
5. Select Projects > Manage > Change Owner.



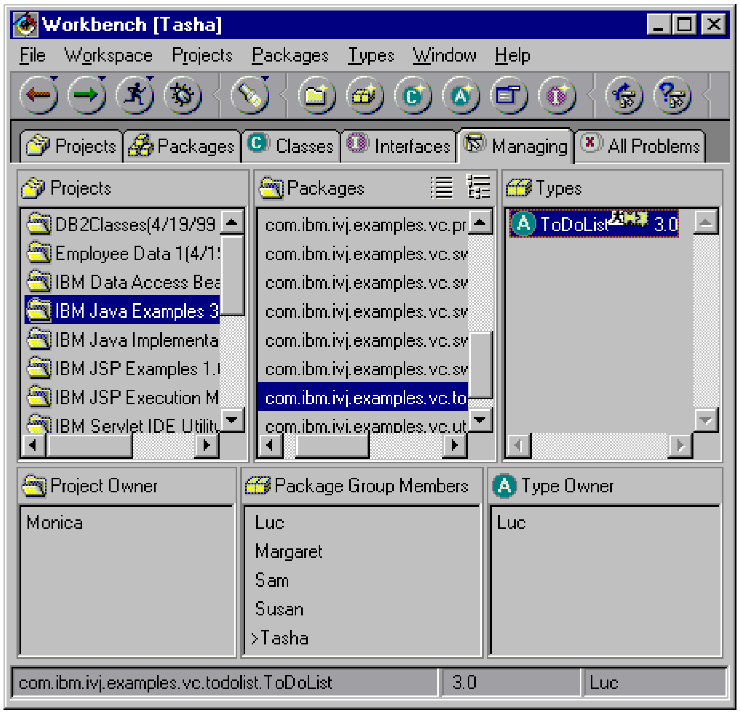
Change the owner of the selected items.

6. The Change Owner window opens. Select **Monica** from the list. Click **OK.** Monica is now the project owner.
7. Before you can make Tasha the owner of the com.ibm.ivj.examples.vc.todolist package, she must be a package group member. Select **com.ibm.ivj.examples.vc.todolist** from the Packages pane. Select **Packages > Manage > Add User to Group**.
8. The Add Users window opens, listing the repository users who are not yet members of the package group. Hold down the **Ctrl** key to select **Luc**, **Margaret**, **Sam**, **Susan**, and **Tasha** from the list. Click **OK**. The package group members now appear in the Package Group Members pane.
9. Administrator is the current package owner, as indicated by a right angle bracket (>). To change the package owner to Tasha, select her name in the Group Members pane, and then select **Set as Owner** from the pop-up menu. The package ownership marker (>) now appears next to Tasha’s name. Make sure that you select **Set as Owner,** not **Set as Workspace Owner**.
10. Next, you will change the class owner of ToDoList to Luc. Select **ToDoList** from the Types pane. From the pop-up menu, select **Manage** and then **Change Owner** The Change Owner window opens. Select **Luc** from the list and click **OK**.

The team roles have now all been assigned. As the new package owner, only Tasha can add and delete package group members now. She should immediately remove **Administrator** from the group, so that no one accidentally creates a class or makes code changes while working as Administrator.

1. Change the workspace owner to Tasha.
2. Select **Administrator** from the Package Group Members pane, then select **Remove** from the pop-up menu.

The Managing page of the Workbench should now appear as follows:



DB2Classes(4/19/99
Employee Data 1(4/1!
IBM Data Access Bea
IBM Java Examples 3
IBM Java Implementa
IBM JSP Examples 1.(
IBM JSP Execution M
IBM Servlet IDE Utility

com.ibm.ivj.examples.vc.pr
com.ibm.ivj.examples.vc.sv
com.ibm.ivj.examples.vc.sv
com.ibm.ivj.examples.vc.sv
com.ibm.ivj.examples.vc.sv
com.ibm.ivj.examples.vc.sv
com.ibm.ivj.examples.vc.to
com.ibm.ivj.examples.vc.ut

ToDoList 3.0

Monica

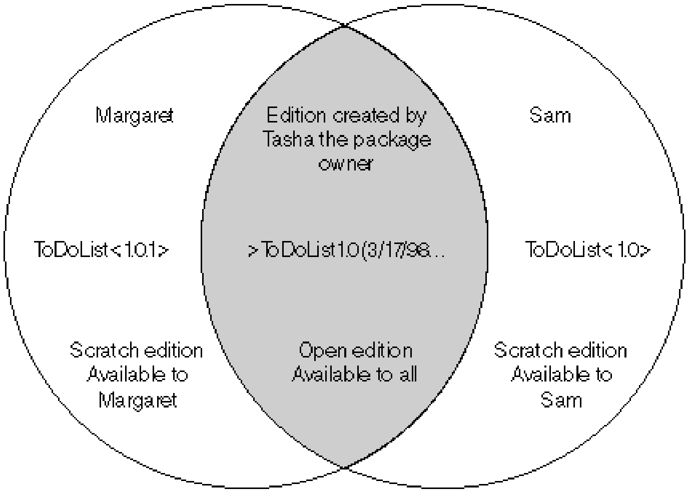
Luc
Margaret
Sam
Susan
>Tasha

Luc

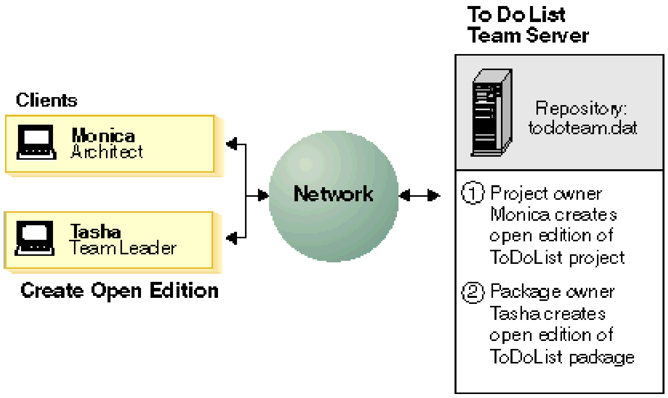
com.ibm.ivj.examples.vc.todolist.ToDoList 3.0 Luc

Opening a project edition of the To-Do List

Scratch editions are private editions. Only their creator can use them. In the diagram below, Margaret and Sam have their own private editions of the ToDoList package. Scratch editions allow developers to experiment with code without affecting the mainstream development of the project. Open editions, created by the owners of program elements, are the ones used to develop the project.

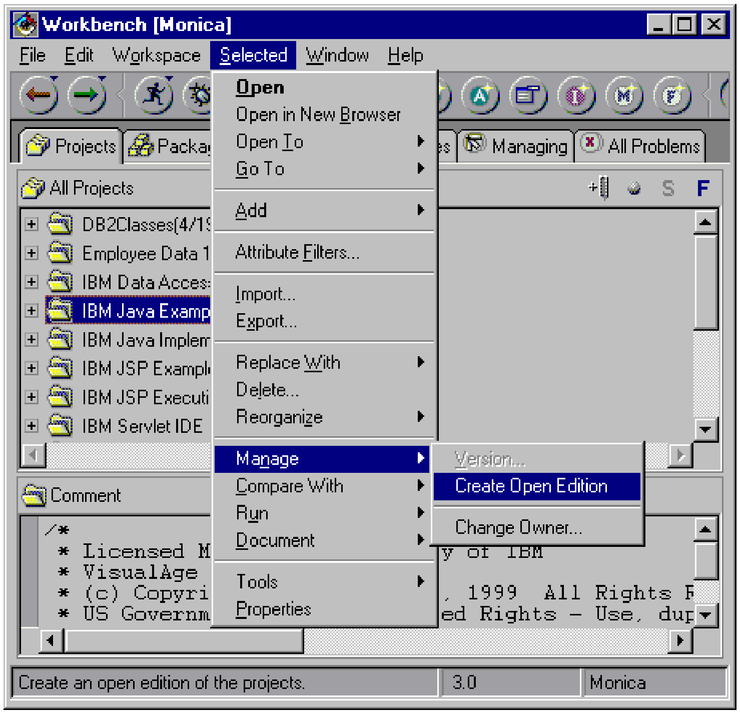

Margaret
Edition created by Tasha the package owner
Sam
ToDoList<1.0.1>
>ToDoList1.0(3/17/98...
ToDoList< 1.0>
Scratch edition Available to Margaret
Open edition Available to all
Scratch edition Available to Sam

To open an edition in a standalone environment, you simply open an edition yourself. In a team environment, you are dependent on the owners of the project and package to open editions.



As Monica, open an edition of the To-Do List project by following these steps:

1. Change the workspace owner to Monica.
2. Click the **Projects** tab in the Workbench window. Select **IBM Java Examples**. Select **Selected > Manage > Create Open Edition**.



```
/*
 * Licensed M                y of IBM
 * VisualAge
 * (c) Copyri              , 1999 All Rights R
 * US Governm              ed Rights - Use, dup
```

Create an open edition of the projects. | 3.0 | Monica

3. An open edition of the project is created. To verify this, click the **Managing** tab. In the **Projects** pane, **IBM Java Examples** is followed by a date and time in parentheses, indicating that it is an open edition.

Creating a scratch edition of the To-Do List package

The next step in the process should be for Tasha, the package owner, to open an edition of the To-Do List package. Before she does that, however, we are going to look at what would happen if a team member decided to start working with the To-Do List class before Tasha had opened the To-Do List package.

Margaret has decided she wants to start working with the To-Do List class and experiment with changing the color of the To-Do Item text field. She decides to start working with the class before Tasha has opened an edition of the To-Do List package. VisualAge for Java creates a *scratch edition* of the To-Do List package for her.

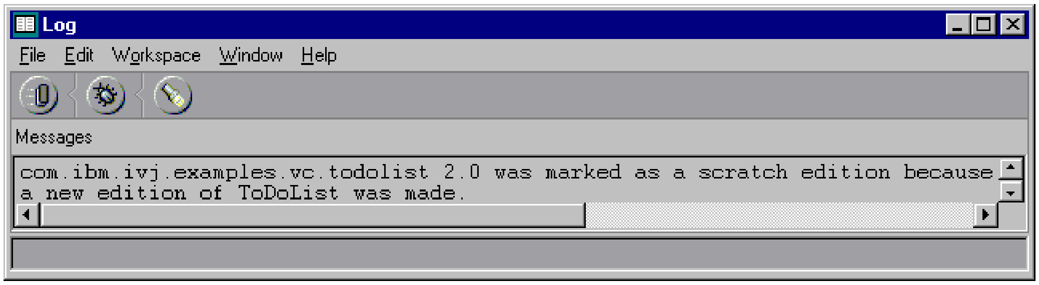
A scratch edition, being private, is not visible to other users. If you have configured your VisualAge for Java options to show edition names, your scratch editions will be designated with < > around the program element's version name.

Scratch editions can be used for various purposes such as to privately experiment with code, or to start development on a class before the owner has opened the containing package.

In this section, you will be Margaret. Follow these steps to create the situation of a developer working on a class *before* the package owner opens the edition:

1. Change the workspace owner to Margaret.
2. Click the **Managing** tab in the Workbench window. Select **ToDoList** from the Types pane and double-click it.
3. Change the text in the To-Do Item text field to magenta. Right-click the To-Do Item text field. Select **Properties**. Scroll down and select **Foreground**. In the right column, click More. In the Foreground window, select magenta. Click **OK**. The foreground color changes to magenta. Close the Properties window.
4. Save the bean by selecting **Bean > Save Bean**. When asked to confirm if a new edition should be created, click **Yes**. Close the Visual Composition Editor.
5. Look at the Log window. The Log window displays messages and warnings.

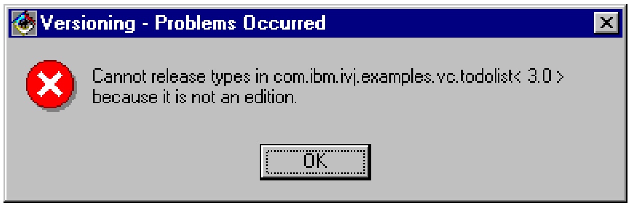
   A message confirms you have created a *scratch edition* of the ToDoList package.

   ```
   com.ibm.ivj.examples.vc.todolist 2.0 was marked as a scratch edition because
   a new edition of ToDoList was made.
   ```

   The Log window can also display other kinds of messages and warnings. Information about compilation errors, inconsistencies between the workspace and the repository, and which program elements have imported and exported are also displayed in the Log window.

   The Log window is locked by default. If you try to close it, a dialog will prompt you confirm your intent, and will give you the option of removing the lock. To lock a window select **Lock Window** from its **Window** menu. Select it again to unlock the window.
6. Try to version the package. Select **ToDoList** from the Types pane in the Workbench window. Select **Types > Manage > Version**. In the Versioning Selected Items window, click **OK**.
7. A messages states that none of the classes in the package be released.

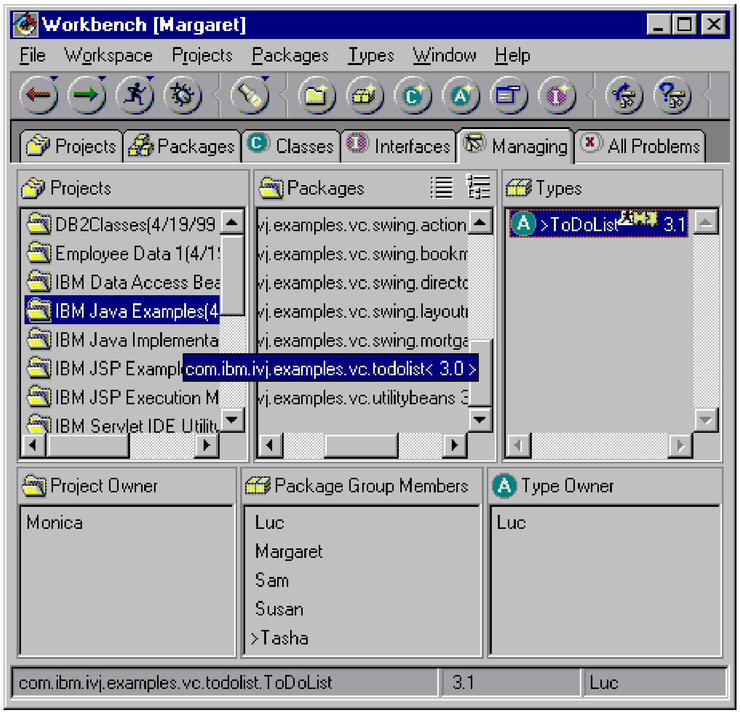


Cannot release types in com.ibm.ivj.examples.vc.todolist< 3.0 > because it is not an edition.

You will work with releasing editions later, but the essence of this message is that Margaret can not make her class available to others on the team because the package that contains it is a scratch edition. Changes can only be released into an open edition. Because Tasha (the package owner) has not yet created an open edition of the package, not even Luc (the class owner) can release new versions of the ToDoList class into the package. Margaret is now working *privately* with a scratch edition; she cannot create a *public* version for others to work with.

Click **OK** to clear the warning message.

8. From the Workbench window, click the **Managing** tab and look at the ToDoList package in the Packages pane. The version name is surrounded by angle brackets <>, which indicate that this is still a scratch edition of the package. Only the scratch edition owner (in this case, Margaret), can use the scratch edition.



DB2Classes(4/19/99
Employee Data 1(4/1!
IBM Data Access Bea
IBM Java Examples(4
IBM Java Implementa
IBM JSP Exampl
IBM JSP Execution M
IBM Servlet IDE Utility

vj.examples.vc.swing.action
vj.examples.vc.swing.bookm
vj.examples.vc.swing.directo
vj.examples.vc.swing.layout
vj.examples.vc.swing.mortga
com.ibm.ivj.examples.vc.todolist< 3.0 >
vj.examples.vc.utilitybeans 3

>ToDoList 3.1

Monica

Luc
Margaret
Sam
Susan
>Tasha

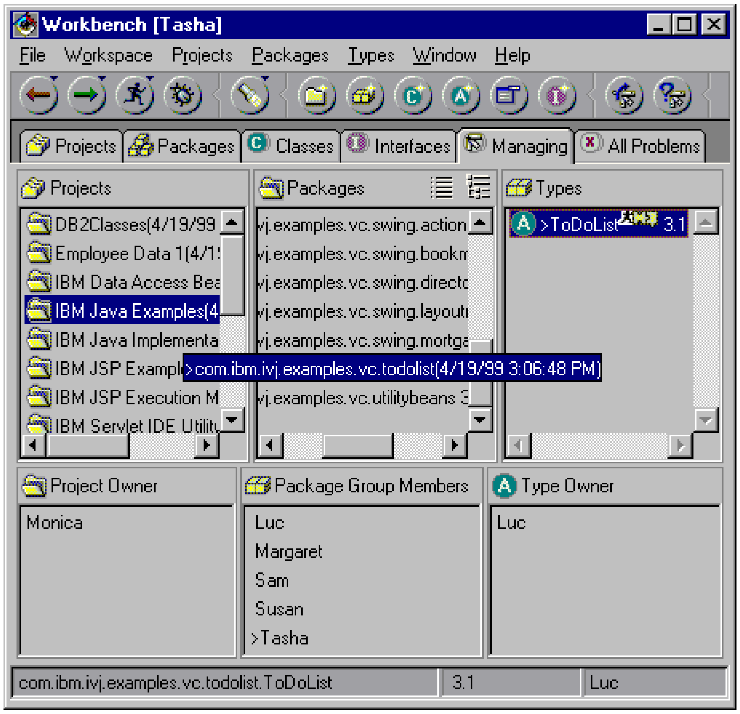
Luc

com.ibm.ivj.examples.vc.todolist.ToDoList | 3.1 | Luc

Opening a package edition of the To-Do List

As Tasha, create an open edition of the To-Do List package so that class owners can release their changes. Follow these steps:

1. Change the workspace owner to Tasha. From the Workbench window, select **todolist** from the Packages pane. Select **Packages > Manage > Create Open Edition.**
2. An open edition of the ToDoList package is created in the **Packages** pane. It is followed by a date and time in brackets, indicating an open edition.


Workbench [Tasha]
File Workspace Projects Packages Types Window Help
Projects Packages Classes Interfaces Managing All Problems
Projects
Packages
Types
Project Owner
Package Group Members
Type Owner

DB2Classes(4/19/99
Employee Data 1(4/1!
IBM Data Access Bea
IBM Java Examples(4
IBM Java Implementa
IBM JSP Exampl
IBM JSP Execution M
IBM Servlet IDE Utility

vj.examples.vc.swing.action
vj.examples.vc.swing.bookm
vj.examples.vc.swing.directc
vj.examples.vc.swing.layoutt
vj.examples.vc.swing.mortga
>com.ibm.ivj.examples.vc.todolist(4/19/99 3:06:48 PM)
vj.examples.vc.utilitybeans 3

>ToDoList 3.1

Monica

Luc
Margaret
Sam
Susan
>Tasha

Luc

com.ibm.ivj.examples.vc.todolist.ToDoList 3.1 Luc

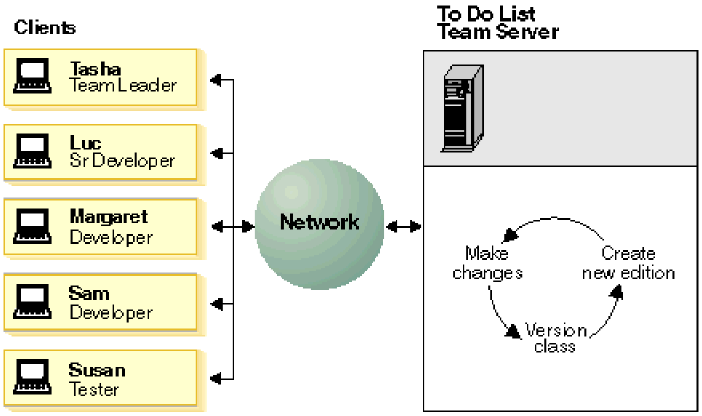
Tasha and the To-Do List team are now ready to start developing their project.

Sharing code

All the To- Do List team members now have access to the todoteam.dat repository, and project and package editions have been opened. They can begin coding together. Margaret has been assigned the task of changing the text in the To-Do Item box to blue. Sam has been assigned the task of changing the Add push button text.

Once the server has been set up and everyone has access to the repository, development of code can begin. First, project and package editions must be opened by their respective owners. Thereafter, day-to-day development is similar to working standalone. Developers open editions, make changes to the editions, and then version them. VisualAge for Java tracks who made the changes so that later, when the code is completed and ready for use, the class owner can merge all the changes into a final version of the class.

In the To-Do List team, the coding is mainly done by Margaret and Sam, although Tasha, Luc, and Susan are occasionally involved. Daily development as a team is shown below.

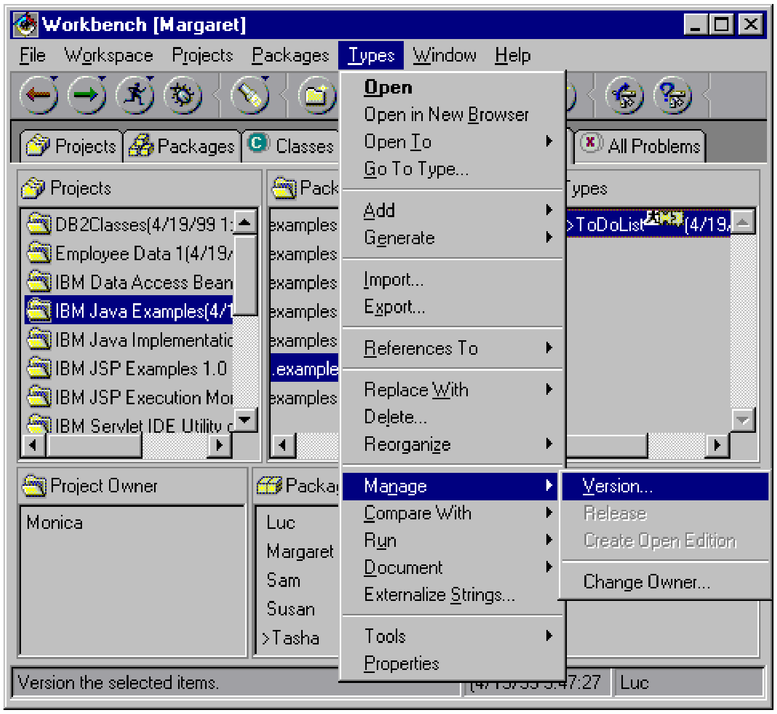


To learn about daily team development, you will connect as Margaret first and make the change that she was assigned (change the item text to blue). You will then connect as Sam and make the change that he was assigned (change the Add push button text).

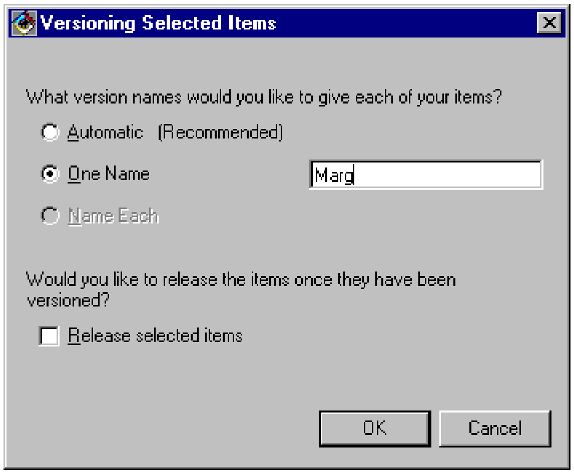
Changing a To-Do List class available to the team

Change the To-Do List class code as Margaret by following these steps:

1. Connect as Margaret to the todoteam.dat repository. Previously, Margaret versioned her own (scratch) edition of the class, but now she should use the original version. From the Workbench window, click the **Managing** tab. Select the **IBM Java Examples** project, the **todolist** package, and the **ToDoList** class. Select **Types > Replace With > Another Edition**.
2. A window containing a list of editions opens. Select the original version. Click **OK**.
3. Open the ToDoList class in the Visual Composition Editor by double-clicking it.
4. Change the text in the To-Do Item box to blue. Right-click the To-Do Item text field. Select **Properties**. Scroll down and select **foreground**. In the right column, click More . In the Foreground window, select blue. Click **OK**. The foreground color changes to blue. Close the Properties window.
5. Test the change. Click **Run**. When asked to confirm that you wish to create a new edition, click **Yes**. Test the applet by entering text in the **To-Do Item** text field. It should appear in blue. Close the applet. Close the Visual Composition Editor.
6. Version Margaret's open edition. Select **ToDoList** from the Types pane in the Workbench window. Select **Types > Manage > Version**.



7. Margaret’s version should be distinctly named. In the Versioning Select Items window, select the **One Name** radio button. Type `Marg` in the text field. Clear the **Release selected items check box** (only the class owner, Luc, can release this class). Click **OK**.



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

○ Automatic (Recommended)

● One Name Marg

○ Name Each

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

□ Release selected items

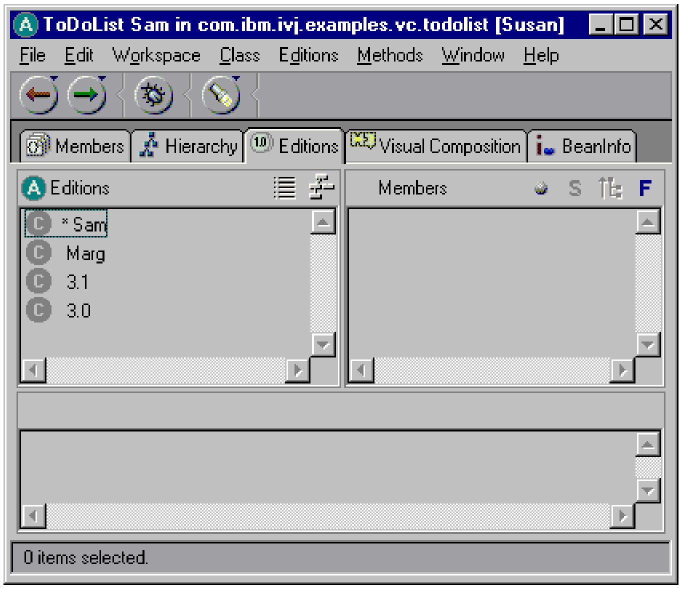
8. A message indicates that you have chosen not to release the classes. Click **OK**.

By versioning her edition of the class, Margaret has made it available to other members of the team. When they open the Repository Explorer window on their workstations, they will be able to browse the Marg version. And, if they select **ToDoList** from the Types pane of the Workbench window, and then select **Replace With > Another Edition** from its pop-up menu, they will be able to load the Marg version into their workspaces.

Changing the same To-Do List class

Now a second developer named Sam changes the To-Do List class. Develop the To-Do List class code as Sam by following these steps:

1. Change the workspace owner to Sam. Sam began his code changes at the same time as Margaret. Like her, he also began with the original version. In the Workbench window, click the **Managing** tab. Select the **IBM Java Examples** project, the **todolist** package, and the **ToDoList** class. Select **Types > Replace With > Another Edition.**
2. A window containing a list of editions opens. Select the original version (the version that predates Margaret's change). Click **OK**. The original version is loaded into the workspace.
3. Open the ToDoList class in the Visual Composition Editor by double-clicking it.
4. Change the Add push button text. Right-click the Add push button. Select **Properties**. Scroll down and select **text**. Change `Add` to `Add Item`. Close the Properties window.
5. Test the change. Click **Run**. A warning appears asking if a new edition should be created. Click **Yes**. The Add push button text should be `Add Item`. Close the applet. Close the Visual Composition Editor.
6. Version Sam's changes. Select **ToDoList** from the Types pane in the Workbench window. Select **Types > Manage >Version**.
7. Sam's version should be distinctly named. In the Versioning Select Items window, select the **One Name** radio button. Type Sam in the text field. Deselect **Release selected items**. Click **OK**. A message indicates that you have chosen not to release the classes. Click **OK**.
8. To verify that everyone on the team can see either Margaret's or Sam's code, change workspace owner to Susan and select **Types > Open To > Editions**. Both Margaret's and Sam's editions appear.



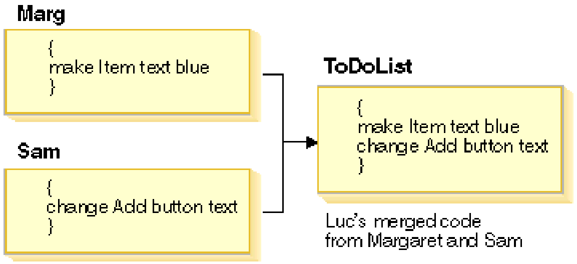
0 items selected.

In the next section, you will see how Luc (the owner of the To-Do List class) consolidates Margaret's and Sam's changes.

Merging code

Luc, the class owner, has decided to merge together everyone's changes to the To-Do List class. Luc will take the original version of the To-Do List class and merge Margaret and Sam's changes into it.

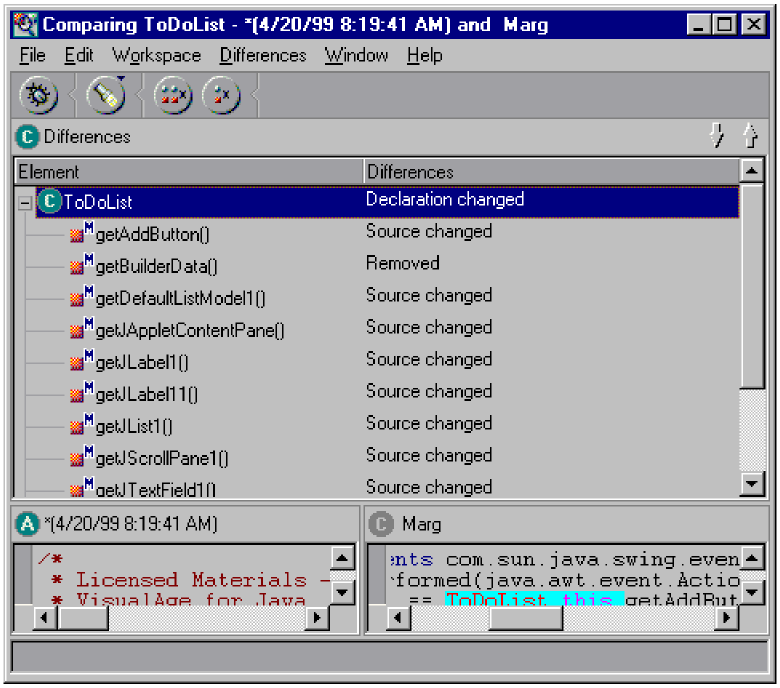
Margaret changed the Item text to blue and Sam changed the text of the Add push button. The following diagram shows the final code that Luc, the class owner, should have after merging their work.



In this section, you will learn how to merge the different editions of a class created by different developers into one versioned edition.

To merge code from several developers at the class level, follow these steps:

1. Change the workspace owner to the class owner, Luc.
2. From the Workbench window, click the **Managing** tab. In the Types pane, select the **ToDoList** class. Select **Types > Replace With > Another Edition**. You should see a list of versions that are available in the repository, including Marg and Sam.
3. Select the original version (the version that predates Margaret and Sam's changes). Click **OK**. This action will load the original To-Do List into your workspace. Using this as your base, you will create a new open edition and merge in the other developers' changes.
4. From the Managing page of the Workbench, select **ToDoList** in the Types pane. Select **Types > Manage > Create Open Edition**.
5. First, Luc will merge Margaret's code. Select **ToDoList** from the Types pane. Select **Types > Compare With > Another Edition**.
6. A window containing a list of editions, including Sam and Marg, opens. Select **Marg** from the list. Click **OK**.
7. A comparison window opens. Select the ToDoList class name. Margaret's code is on the right and Luc's open edition (the original code) is on the left. The "Source changed" identifier under the Differences column indicates where the code has changed.

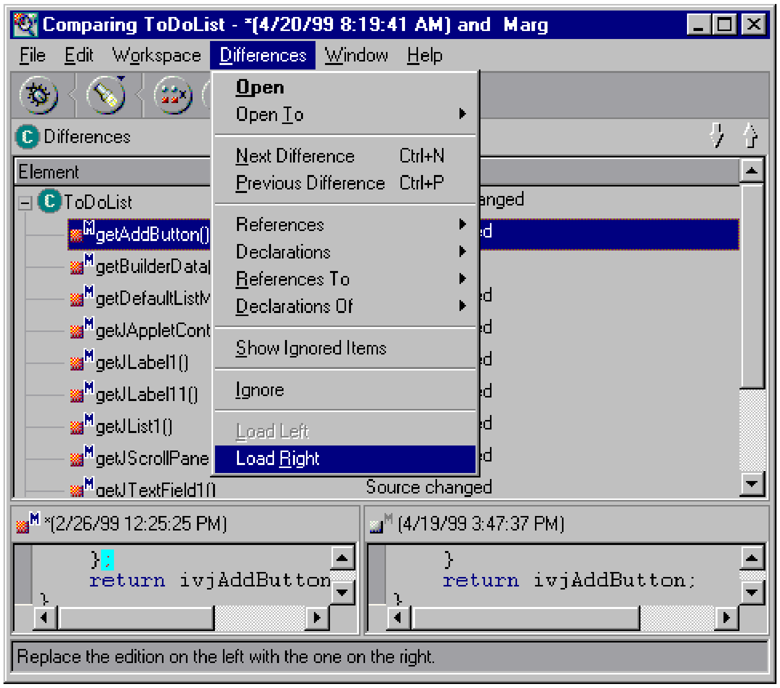

Comparing ToDoList - *[4/20/99 8:19:41 AM] and Marg
File Edit Workspace Differences Window Help
Differences
*[4/20/99 8:19:41 AM]
Marg

Element	Differences
ToDoList	Declaration changed
getAddButton()	Source changed
getBuilderData()	Removed
getDefaultListModel1()	Source changed
getJAppletContentPane()	Source changed
getJLabel1()	Source changed
getJLabel11()	Source changed
getJList1()	Source changed
getJScrollPane1()	Source changed
getJTextField1()	Source changed

```
/*
 * Licensed Materials -
 * VisualAge for Java
```

```
nts com.sun.java.swing.even
formed(java.awt.event.Actio
 == ToDoList.this.getAddBut
```

8. Select **Differences** > **Load Right**, which merges Margaret's code (shown on the right) into the open edition (shown on the left).



```
    };
    return ivjAddButton
}
```

```
    }
    return ivjAddButton;
}
```

Replace the edition on the left with the one on the right.

9. Continue merging all methods that say "Source changed". Ignore (do not load) any class differences such as Declaration changed. To select multiple methods, hold down the **Shift** key while you are selecting the methods. Close the Comparing window.

10. Run the applet by clicking **Run**. Test that the text in the Item text field is blue.

11. Next, Luc will merge Sam's code into the open edition. In the Managing page of the Workbench window, in the Types pane, select your open edition of **ToDoList**. (This edition now includes Margaret's changes). From the pop-up menu, select **Compare With > Another Edition**. Select **Sam** from the list of editions in the repository. Click **OK**. The comparison window now has Sam's code on the right.
12. Using **Load Right**, merge only Sam's **getAddButton( )** method into your open edition. *Do not merge* the getTextField1 method. Margaret's code changed the text to blue, and if you merge Sam's code, you will return the Item text to black. While merging code, you must know which changes you want and do not want. Close the Comparing window.
13. Finally, run the applet to see that text in the Item text field is blue (Margaret's change) and that the Add push button text is `Add Item`. (Sam's change). You should always test changes before you release them into the team baseline.

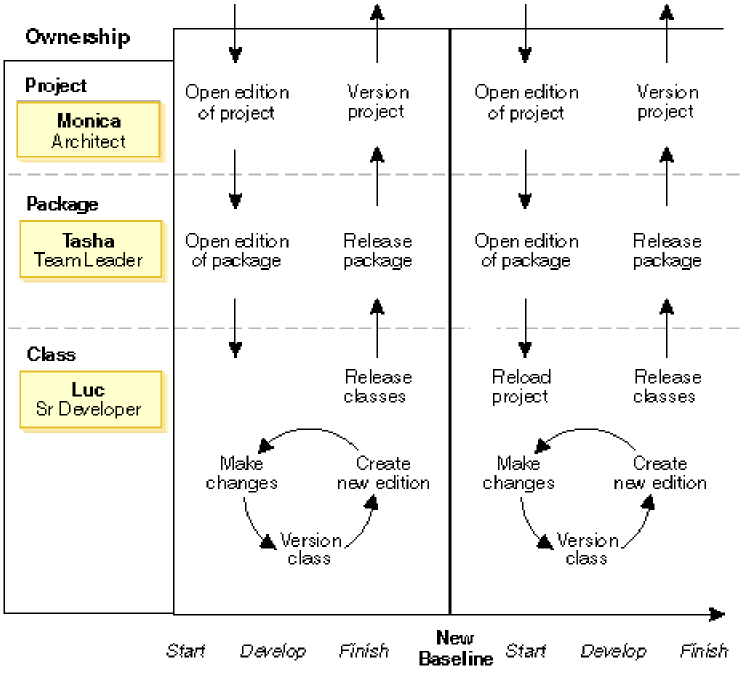
Now that you have merged Margaret and Sam's changes and tested them, you are ready to set a new baseline.

Setting a new baseline

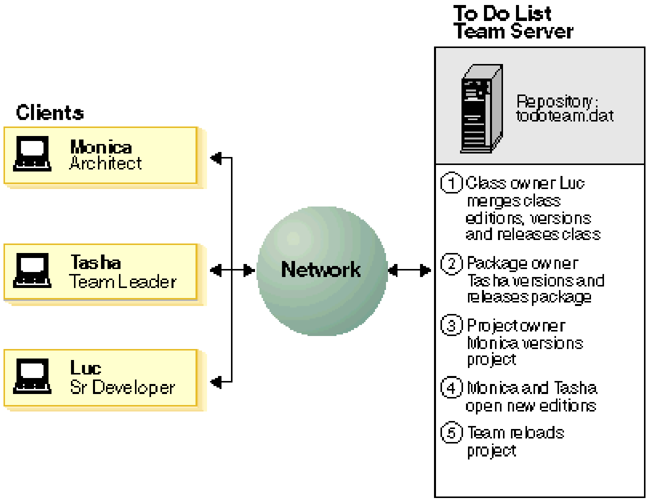
Periodically, after class owners have tested their changes, the team will decide to set a new baseline. Setting a baseline is like synchronizing the code.

After a baseline has been set, the team opens a new edition of the project and continues developing. code. Eventually, the baseline becomes the finished product. In this section, you will learn how to set a new baseline by releasing the classes, releasing the packages, creating a new version of the project, and opening a new edition of the project.

So that you can practice going through a complete development cycle, in this section you will set a baseline at the project level. Setting a baseline can also be done at the package level. Refer to the online help for more information on setting a baseline at the package level.



Setting a baseline at the project level
The diagram below shows the process for setting a baseline at the project level. The process works in the reverse direction of starting a project - it starts at the the class level and proceeds to the package and project levels. Once Monica versions the project, the baseline is reset by opening a new project and package edition, which all of the team members reload.

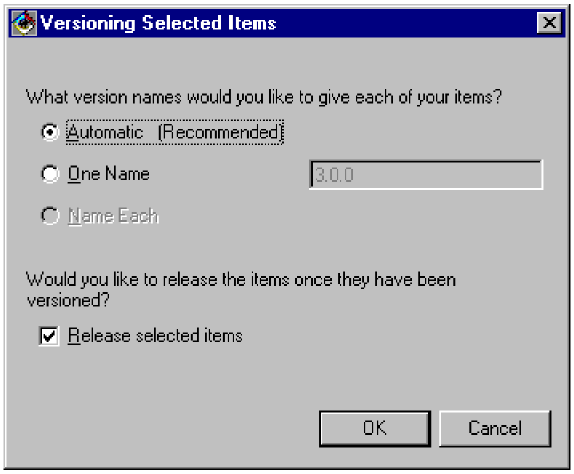

To Do List
Team Server
Repository:
todoteam.dat
1 Class owner Luc merges class editions, versions and releases class
2 Package owner Tasha versions and releases package
3 Project owner Monica versions project
4 Monica and Tasha open new editions
5 Team reloads project
Clients
Monica
Architect
Tasha
Team Leader
Luc
Sr Developer
Network

Releasing the To-Do List classes

Luc will now version and release the class that he just merged:

1. Confirm that the workspace owner is still Luc. If not, change the workspace owner to Luc.

2. From the Managing page of the Workbench, select the open edition of the **ToDoList** class, which now contains Margaret's and Sam's changes. Select **Types > Manage > Version.**
3. In the Versioning Selected Items window, select **Automatic** to change the To-Do List version to 3.0.0. This time, select **Release selected items** so the class will be released to the package level. Click **OK**.



Versioning Selected Items

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

● Automatic (Recommended)

○ One Name 3.0.0

○ Name Each

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

☑ Release selected items

4. Look at the Types pane of the Workbench window. The unreleased marker (>) should *not* appear beside the class name.

When Luc releases a version of the class, he is updating the team baseline at the package level. Team members can synchronize with all the classes that have been released into a package, by selecting **Replace With > Released Contents** for that package in the Workbench. This action refreshes their workspaces with the most recently released class versions.

Releasing the To-Do List package

Next, the package owner releases the package. To release a package, follow these steps:

1. Change the workspace owner to Tasha. Only the package owner, Tasha, or the project owner, Monica, can release the package.
2. Select **com.ibm.ivj.examples.vc.todolist** from the Packages pane. Select **Packages > Manage > Version.**
3. In the Versioning Selected Items window, select **Automatic**. Select **Release selected items** to release the package to the project level. Click **OK**.
4. Look at the Packages pane of the Workbench window. There should *not* be an unreleased marker (>) beside the package's name.

Every time Tasha releases her package, she is updating the team baseline at the project level. Team members can update their workspaces with all of the latest packages *and* classes for that project, by selecting **Replace With > Released Contents** for the project.

Versioning the To-Do List project

Finally, the project owner creates a new version of the project, to preserve the team baseline at a particular point in the development cycle.

1. Change the workspace owner to Monica. Only the project owner, Monica, can version the project.

2. On the Managing page of the Workbench, select **IBM Java Examples** from the Projects pane. Select **Projects > Manage > Version**.
3. In the Version Selected Items window, select **Automatic**. Note that there is no **Release** check box for projects. Click **OK**.
4. The Managing page of the Workbench should now show that the project, package and class are at the new level.

The team can revert to this version of the project at any time by reloading the project. To reload, they would select **Replace With > Previous Edition** or **Replace With > Another Edition** from the project's menu in the Workbench.

Preparing for new development
In Owning To-Do List code, you started your project by opening editions of the To-Do List project and package. You can do this again, but use the latest version of the To-Do List project as your base. New editions of the project and packages must be opened. If they are not, the team members can only work with scratch (private) editions that they cannot share with each other and that they cannot release into the team baseline.

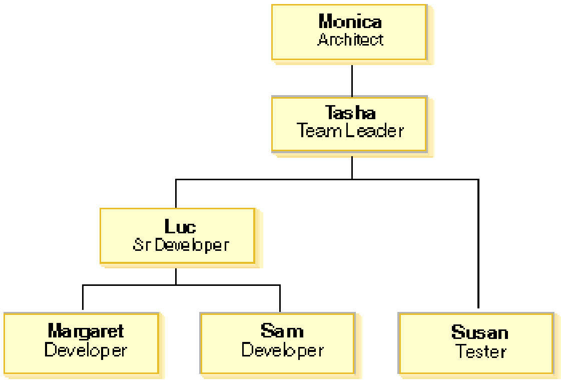
After Monica creates a new open edition of the project, Tasha would load that edition into her workspace and create an open edition of her package within that project edition. The rest of the team would then reload the project and start working with the code again. They are now working from a baseline that includes the changes that Margaret and Sam made.

Team programming

Developing code in a team is different from working alone. It involves shared responsibility for developing code and shared access to the files containing the code. It requires working in a client/server configuration - many clients are connected to one server that contains shared code.

The team development model that is provided with VisualAge for Java, Enterprise is particularly suited for object-oriented programming by small groups of developers who develop classes in parallel.

With this *Getting Started* guide, you can create a To-Do List program in a standalone environment. Then a team of six developers will develop a more complicated version of it. The figure below shows the structure of the To-Do List team.



Teams are collaborative, but team members are not equals and each have different responsibilities. In our example, Monica has the highest responsibility and her position has the broadest scope. She is concerned with the overall design of the code, but not the low-level implementation of it.

As team leader, Tasha is closer to the actual code development than Monica. She knows about the specific functions of each component, as well who is working on it.

Luc is a senior developer with two developers, Margaret and Sam, reporting to him. Luc is involved in the design of the code, makes key decisions about its implementation, and also does a lot of coding.

Margaret and Sam have a narrower scope of work. They actually develop the code for the components. Susan, like them, is focused on a specific task: testing.

In your environment, you may distribute roles similar to these across a very small team that may have only two or three people or a very large team that may consist or twenty or more people.

Client/server view of the To-Do List project

You already know that VisualAge for Java stores editions of all program elements in a source code repository. By contrast, your workspace contains the source code for the program you are currently working on.

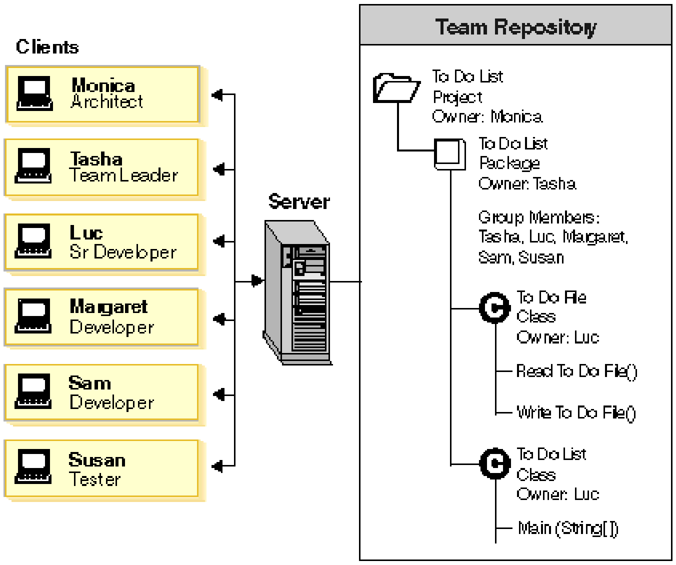
In the team development environment, the repository is a shared file that resides on a file server and stores all the code for all developers on the team. Team members can access the repository from their own computers, which are set up as clients to the server.

In the team development environment, each VisualAge for Java client has its own workspace. For example, Margaret and Sam each have their own individual workspaces, which can contain both code from the shared repository, as well as their own personal code.

To connect to a shared repository, your workspace must have an *owner*. The owner can be any of the team members, as long as they have been added to the *repository user list.* The workspace owner's privileges determine what you can do with program elements in the shared repository.

The To-Do List team roles

Each team member's responsibilities are determined by assigning ownership of program elements. The following diagram shows how ownership of objects in the team repository maps to the roles of the To-Do List team members.



The scope of each person's ownership closely matches his or her level of responsibility in the team. Luc owns two classes; Tasha owns the package containing all classes; and Monica owns the project containing the packages.

The four main roles for developers are:

Class developer
Anyone who opens an edition of an existing class

Class owner
Can open, edit, release or delete a class

Package owner
Can open, version and release as package, and add and delete users from the package group

Project owner
Can open and version a project, and create and delete packages in the project

There is also another, specialized role, that of the Administrator. The repository administrator can perform the following tasks:

- Add new users to repositories
- Change or delete existing users
- Compact repositories
- Change ownership of existing projects
- Purge open editions and versions of projects and packages

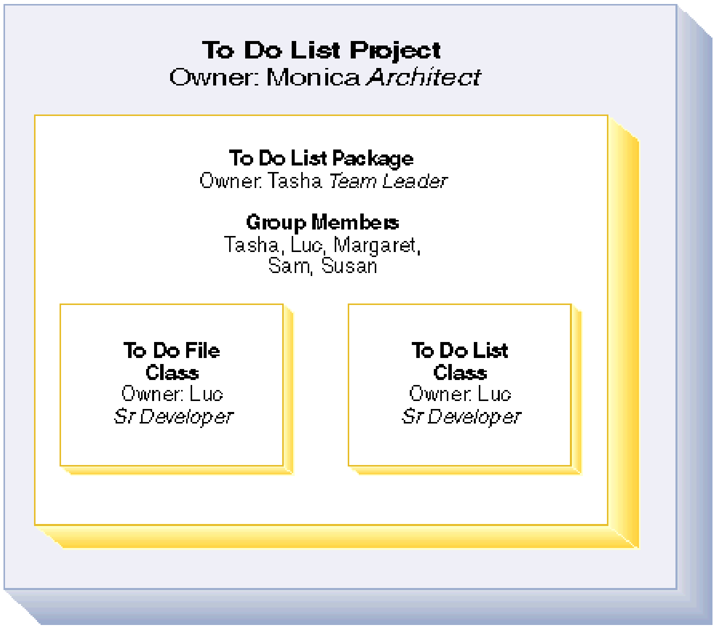
Several of these tasks are described in this guide. The person who performs these server administration tasks may also be a member of your development team. In this case, that person should connect to the repository with a different user name when developing code. The Administrator ID should only be used to perform administration functions.

You cannot delete Administrator from the repository, and you cannot change the unique name for that user. If you wish, you can change the repository administrator's full name.

Scope and responsibilities of ownership

In the team development environment, every project, package, and class has an owner.

The diagram below shows the scope of ownership in our To-Do List example.



Change control is based on ownership. Different team members can work with the same program element at the same time, and their changes will automatically be saved in the shared repository. Owners, however, control the mainstream of development; only they can *release* the program element that they own.

For example, more than one developer is allowed to change a class, but the class owner must review these changes, decide which are acceptable, and determine what version of the class will be released to the containing package. Margaret and Sam can develop methods, but their methods do not automatically become part of the ToDoFile class. Luc, the class owner, must approve of the changes and then release the class.

Releasing is like advancing stable code to a higher level. Releasing typically occurs after stable code has been developed, reviewed, and tested. When all the classes in a package have been released, Tasha, as package owner, has the authority to release the package. Similarly, when the To-Do List package has been released, Monica can create a new version of the To-Do List project. Monica's new version could be thought of as a new *baseline* (see "Setting a new baseline" on page 80 for more information).

After a class is released to a package, all members of the team see that version of the class when they load the updated version of the package from the repository into their workspace.

Group membership

In VisualAge for Java, Enterprise Edition, each edition of a package has a group of developers who are assigned to work with classes in that package. These developers are the *package group* for that edition.

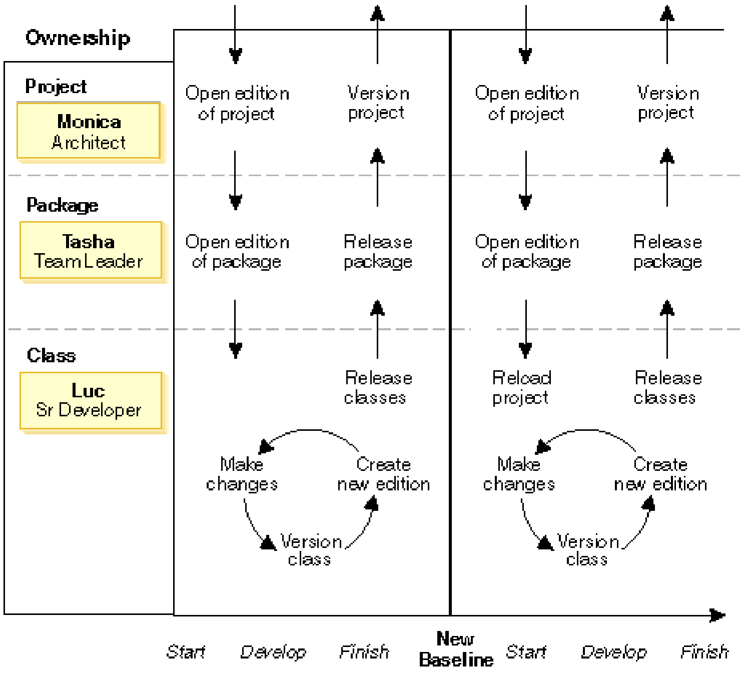
Margaret, Sam, and Susan (along with Tasha and Luc) are package group members of the To- Do List package. Group members can own, create, and change classes in the package. Since Margaret and Sam are group members, they can change classes in the package. Group membership enables Susan to access and test all the classes in the To- Do List packages.

Monica is not a package group member, even though the To- Do List package is in her project. Monica does need to be a package group member because she does not code or test, and is not involved in the day-to-day development of the code.

The package owner is the only person who can add or delete users to or from the package group. Different editions of packages can have different package group members.

The To-Do List team development cycle

The following diagram shows how the team works together to develop a product. Monica begins the cycle by opening an edition of the project. Then Tasha opens an edition at the package level. Opening an edition is like kicking off a project. It simply means that the group can now develop the shared code. That development takes place iteratively, as indicated by the wheel that shows the developers (Luc, Sam, and Margaret) making changes, making a version of their class with the changes, and then opening a new edition of the class.



When the classes are tested, Luc releases them and Tasha releases the package. Monica then versions the To-Do List project. This synchronizes the code. Everyone’s changes have been captured. In iterative development, this point is called “Setting a new baseline” on page 80. The diagram shows that baseline at a project level, but you can also set baselines at the package level.

You may set a new baseline at several stages during the development of an application. In the preceding example, Monica's version leads to the start of further development.

Accommodating growth of the team

The team model is scalable; it accommodates the growth of a team. If you are developing a very complex product, you may have you have may have many class, package, and project owners.

Conversely, very small teams can also work effectively in the VisualAge for Java team environment. One person might combine the roles of project owner, package owner, and class owner. The other developers would be package group members.

Chapter 8. Interface to external version control systems

The version control interface enables VisualAge for Java users to connect to source code in an external source code management (SCM) system. An SCM system enables developers to manage their source code, and they offer features such as versioning and exclusive locks for editing.

The interface from VisualAge for Java to external version control systems uses Microsoft's Source Code Control (SCC) API. It supports the following SCM systems:

- ClearCase for Windows NT, from Rational Software Corporation
- PVCS Version Manager, from MERANT (formerly INTERSOLV)
- VisualAge TeamConnection from IBM Corporation
- Visual SourceSafe from Microsoft®

For a list of the supported versions of these SCM systems, refer to the release notes.

VisualAge for Java projects can be associated with a specific file group in a SCM provider's repository. An example of this is an SCCI project or a Team Connection release. Associating a VisualAge for Java project with an SCM file group is called "adding a project to version control".

Adding projects to version control provides you with a convenient way for you to access an SCM provider without leaving the VisualAge for Java IDE. After you have added a project to version control, you can use it to perform tasks such as checking classes in and out of your SCM system, and importing the most recently checked-in version of a class from the SCM system, all without leaving the IDE. You can also use the Version Control interface to add new classes created in VisualAge for Java to your SCM system.

External SCM system versus VisualAge for Java team development
VisualAge for Java, Enterprise Edition, provides a team development environment that offers version control and a shared repository management. The shared repository offers excellent support for day-to-day team programming activities.

You may, however, wish to install the interfaces to external version control systems as a complementary feature for any of the following reasons:

- You already use another SCM provider as your standard for application development.
- Your build process relies on external SCM providers
- You have established practices for archiving applications on a particular enterprise server, for example for disaster recovery purposes.
- The repository in VisualAge for Java manages Java objects only; you may wish to manage all of your development artifacts with a single provider, or to integrate multiple programming languages across your environment.

The VisualAge for Java online help discusses the interface in more detail.

Chapter 9. Server-oriented programming

This section looks at the development of Java applications specifically designed for servers. Unlike small, standalone applications that can run on a single computer, large applications usually involve a client/server model or a multi-tier model that has a single client and several servers.

Server-oriented programming also results from the industry move to the development of software *components*. For example, an enterprise bean is a component that resides on a server and provides services to other components. Together, the interacting components comprise complex, server-oriented (sometimes referred to as server-side) applications.

The following server-oriented programming tools are examined in this section:

The **JSP/Servlet Development Environment** enables you to develop, run, and test JavaServer Pages (JSP) files and servlets.

The Create Servlet SmartGuide enables you to create servlets, which are Java programs that plug into Web servers. You can use them to perform tasks such as connecting databases to the Web, and expanding from client/single-server applications to multi-tier applications.

[ENTERPRISE] The **Enterprise JavaBeans (EJB) Development Environment** enables you to develop EJB components that provide server-side services such as database transactions. An enterprise bean is unlike a typical Java bean because it is non-visual and resides on a server. It is transaction-aware, meaning it is used to provide transactions, including complex distributed transactions with relational databases. One practical use of enterprise beans in a large application is to create persistent objects (objects that exist beyond the current session).

The EJB Development Environment is available only with the Enterprise Edition of VisualAge for Java.

JSP/Servlet Development Environment

JavaServer Pages (JSP), a server-side scripting technology, allows you to embed Java code within static Web pages (HTML documents), and execute the Java code when the page is served. By separating presentation logic (content presentation) from business logic (content generation), the JavaServer Pages technology makes it easy for both the Java programmer and the Web page designer to create HTML pages with dynamic content.

A Java servlet is a Java program that runs on a Web server. Servlets are highly extensible and flexible, making it easy to expand from client/single-server applications to multi-tier applications. For example, you might write a servlet that connects a web client to an existing database on a host.

You can develop, debug, and deploy servlets within the VisualAge for Java IDE. In the IDE, you can set breakpoints within servlet objects, and step through code to make changes that are dynamically folded into the running servlet on a running server, without having to restart each time.

When JSP-generated servlet code is imported into the VisualAge for Java IDE, the following occurs:

1. The JSP source is fed to a page compiler, which creates an executable object (for example, a Java HTTP servlet).
2. VisualAge for Java then imports the generated servlet code. You can run and debug the servlet by using your browser to call the JSP page that created the servlet.

Note: You can select not to have the generated servlet code imported into the IDE. You may want to do this to help decrease the amount of space that is taken up in the repository by the servlet code.

A JSP file can be directly requested as a URL, called by a servlet, or called from within an HTML page. In all three cases, the servlet engine compiles the JSP into a servlet and runs it. The compilation is performed the first time the JSP is requested, and each time the JSP source changes. This dynamic compilation allows you to deploy new versions of JSP files inside a running Web application. As well, performance is improved because you do not have to compile, load, and run a servlet each time a request is made to the server.

ViusalAge for Java also includes the following tools to help you with JSP development:

- **JSP Execution Monitor,** which helps you to monitor the execution of JSP source, along with the generated Java servlets and HTML source. This tool allows you to view the generated Java code, the original JSP source code, and the HTML output as it is generated. The JSP Execution Monitor also highlights the location of syntax errors in both the JSP and JSP-generated Java source.
- **Persistent Name Server,** which you can use to work with EJB beans or DataSource objects.
- **Websphere Test Environment Servlet Engine,** which enables you to run multiple Web applications, each having its own document root. You can also use it to configure various parts of the Web application.
- **DataSource objects.** You can use DataSource objects to manage a collection of connections to a database. Using connection pools helps you save time, simplify resource allocation, and simplify connection calls.

These tools are all part of the WebSphere Test Environment Control Center, which provides a central location for you to start, stop, and configure WebSphere Test Environment services.

The VisualAge for Java online help discusses the JSP/Servlet Development Environment in more detail and provides samples. You can also refer to the online help for more information about the WebSphere Test Environment Center.

Servlet SmartGuide

The Servlet SmartGuide is a wizard that enables you to create servlets and related Web resource files (HTML and JSP pages). Together, they make up a Web application. You can use the SmartGuide to import Java beans, and then generate HTML, JavaServer pages, and servlet configuration files from the beans.

Servlets are Java programs that run on Web servers. They enable businesses to perform tasks such as connecting databases to the Web, and expanding from client/single-server applications to multi-tier applications. To learn more about servlets, refer to the online help for the JSP/Servlet Development Environment online help.

When you develop servlets with the SmartGuide, you can use JavaServer pages files that inherit from the PageListServlet class.

When you create servlets with the SmartGuide, you can select which fields you want displayed on the input page, which fields you want displayed on the results or output page (JSP page), and which action methods to call when the server logic runs. As well, you can specify which modifiers you want to use and what method stubs you want to create.

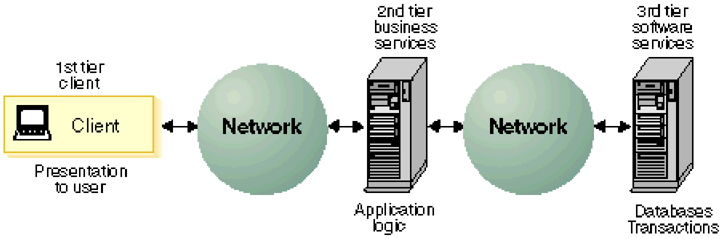
You can use the tools in the WebSphere Test Environment Centre to test the servlets generated by the Servlet SmartGuide. For more information about these features, refer to the online help.

Enterprise JavaBeans

This section looks at the server-oriented programming aspects of VisualAge for Java. To understand server-oriented programming, you need to understand the evolution of the multi-tier client/server model.

Evolution of the multi-tier model
For many years, client/server systems have used workstations to provide easy-to-use, graphical applications. Traditional client/server applications contained all the presentation, business, and data manipulation logic at the client. This “fat client” scenario evolved into a two-tier system in which a “thin client” contained the presentation logic but the business logic became part of the server. An application is easier to manage and better optimized for performance if the application code remains on a server.

Today, a three-tiered model has evolved as the most efficient one for an application of distributed objects, with one server containing the application logic and another server or servers containing software services needed by the application logic, such as databases.



Moving business and data manipulation logic to servers means applications can take advantage of high-end, multithreaded and multiprocessor systems. Many concurrent users can be supported. Also server components can share resources such as processes, threads, and database connections. Data can be partitioned and replicated, enhancing reliability. Components can be distributed across servers, making applications scalable.

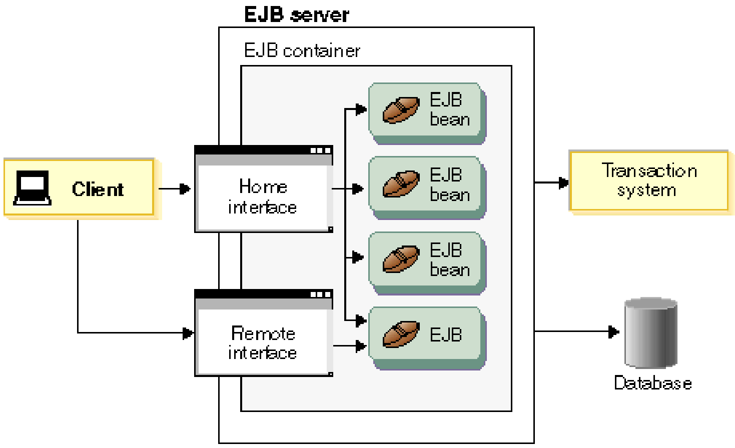
The Enterprise JavaBeans component model

The Enterprise JavaBeans component model is a new model for developing and deploying applications suited for the multi-tier model. It describes a server framework for distributed Java components. Let's contrast it with JavaBeans. A standard Java bean, such as a button, is a reusable software component that is manipulated visually with a builder tool. The JavaBeans architecture is usually regarded as a component model for the client. But for enterprise requirements, the JavaBeans architecture does not offer life-cycle management; that is, the ability to guarantee that objects persist through time. It does not have infrastructure APIs for services like transactions management and security. You could, of course, code these services into each bean but it would require redundant work for each bean and create a non-portable solution.

The Enterprise JavaBeans model supplies this missing framework; it makes multi-tier, server-oriented component development a reality. Designed for distributed, enterprise computing, this model has five elements:

- The server which provides transaction management and security.
- The container in which the enterprise beans will execute.
- A home interface that lists the methods for locating, creating, and removing instances of EJB beans.
- A remote interface that lists the business methods in the EJB class.
- An interface to databases and back-end systems.

At run time, the model looks as follows:



There are two types of EJB beans: session beans and entity beans. Session beans do not have a persistent state. In other words, session beans do not outlive their process. Entity beans have a persistent state, and their state exists beyond the current process.

The EJB Development Environment

The EJB Development Environment of IBM VisualAge for Java is a specialized environment that you can use to develop and test enterprise beans for use in your Enterprise applications. Enterprise beans are specialized non-visual beans that are deployed in EJB containers and run on an EJB server. They can be customized by changing their deployment descriptors and assembled with other beans to create applications. EJB containers manage classes of EJB objects. Specifically, an EJB container manages the life cycle of an EJB object, coordinates distributed transactions, and implements object security.

An EJB server, as defined by the Sun Microsystems EJB specifications, provides a run-time environment that supports the execution of applications that use enterprise beans. The EJB container manages and coordinates the allocation of resources at the server used by the enterprise beans. The EJB Development Environment provides an implementation of Sun's EJB server that enables you to test enterprise beans before installing them on a production EJB server.

Enterprise beans provide several benefits for application developers:

- Enterprise beans make it possible to build distributed applications by combining components developed using tools from different vendors.
- Enterprise beans make it easy to write applications. Application developers do not have to deal with low-level details of transaction and state management, multithreading, resource pooling, and other complex low-level APIs. However, an expert programmer can gain direct access to the low-level APIs.
- Enterprise beans can be developed once and then deployed on multiple platforms without recompilation or source code modification.
- The EJB architecture is compatible with other Java APIs. It provides interoperability between enterprise beans and non-Java applications.
- Enterprise beans are compatible with CORBA.

The EJB Development Environment consists of multiple tools that can be categorized into the following groups:

- Tools for creating enterprise beans and associated components, such as access (adapter) beans and associations
- Tools for generating deployed code for a WebSphere EJB server
- Tools for testing enterprise beans before you install them on a production EJB server

All of the EJB Development Environment tools are accessible from the EJB page of the Workbench. The EJB page is the heart of the EJB Development Environment. This is where your EJB groups and individual enterprise beans reside, and it is where you accomplish almost all of your EJB development activities.

For more information on the EJB Development Environment, see the online help. It contains tutorials you can use to learn how to work in the EJB Development Environment.

Chapter 10. XML Development

XML Generator

Creating test cases for Extended Markup Language (XML) applications by hand from Document Type Descriptions (DTDs) can be tedious. The XML Generator is a Java program that you can use to edit a DTD and generate sample XML documents based on that DTD. You can use the sample documents to check that the DTD is working properly, to see what kind of XML documents are generated by the DTD, and to test applications designed to use those documents.

When you use the generator, you can set constraints to limit the size or customize the appearance of the output XML. For example, you can can limit the depth of the generated tree, limit the number of IDs an IDREFs attribute can contain, or choose whether or not implied attributes should appear. You can also indicate which entities should appear within PCDATA using a configuration file.

By setting constraints, you can generate random test cases for batch testing of your XML applications.

There are many situations you could use the XML Generator in. For example, suppose you are developing an application which processes MathML documents (MathML is very large). You have a number of valid test cases, but you recognize that these test cases may not cover many of the possible forms which may be allowed by the DTD. You could decide to run XMLGenerator to generate some number of random sample documents. Generating random samples gives broader coverage than is likely given contrived or real examples. The more samples you generate, the more complete the coverage.

The VisualAge for Java online help discusses the XML Generator in more detail.

XML Parser for Java

IBM's XML for Java parser is a high-performance, modular XML parser written in Java. The parser provides a way for Java applications to read and write XML data. A single JAR file provides classes for parsing, generating, manipulating, and validating XML documents.

The XML Parser for Java conforms to the XML 1.0 Recommendation and associated standards. The parser has a modular architecture so that you can customize it with only the features your application needs

The XML generation and validation capabilities of the parser enable you to do the following:

- Build XML-aware servers
- Create applications that will use XML as their data format
- Create XML editors that validate dynamically
- Ensure the integrity of e-business data expressed in XML
- Build internationalized XML applications

A sample application that would use the XML parser is a business application that accesses a database that stores its data in XML format. The application would retrieve XML documents and run them through the XML parser. The XML parser would output an XML object. The business application would then invoke methods on the XML object to retrieve or manipulate the XML data.

XMI Toolkit

Visual models (diagrams) help developers to recognize and analyze potential problems. If, for example, you are part of a team of developers building an application that allows customers to do transactions over the Internet, you might decide to create a visual model to help in the planning and architecting of the application. Models help to identify requirements, design cleaner applications, and encourage documentation at the earliest stages of development. Models also help break down complex problems; having a visual plan helps you to build complex applications more easily.

However, when you have created a model, it is beneficial to have the means of moving directly from the model to the code itself. Although you have a visual plan laid out before you, it will not help the end-user unless the plan develops into a real application. For this reason, it is important to have the means of moving easily from model to source code, from plan to application.

VisualAge for Java supports visual modeling by providing a tool, the **XMI Toolkit**, that allows you to roundtrip information from Rational Rose to VisualAge for Java and back to Rational Rose. Rational Rose, a visual modeling tool, allows you to perform analysis and design. Using Rational Rose, you can visually design and model Java constructs (for example, packages, classes, inheritance, interfaces, and so forth), and therefore plan ahead in building your Java application.

The XMI standard

The XMI Toolkit uses the XML Metadata Interchange (XMI) technology to perform conversions between object-oriented analysis models developed in Rational Rose and VisualAge for Java.

XMI specifies an open information interchange model that allows developers to exchange programming data over the Internet in a standardized way.

XMI is an accepted industry standard that combines the benefits of the Web-based XML standard for defining, validating, and sharing document formats on the Web with the benefits of the object-oriented Unified Modeling Language (UML). UML itself is a specification of the Object Management Group (OMG) that provides application developers with a common language for specifying, visualizing, constructing, and documenting distributed objects and business models.

By using an industry standard for storing and sharing object programming information, development teams using tools from multiple vendors can collaborate on applications. The XMI standard allows developers to leverage the Web to exchange object-oriented data among tools, applications, and repositories, and to create secure, distributed applications built in a team development environment.

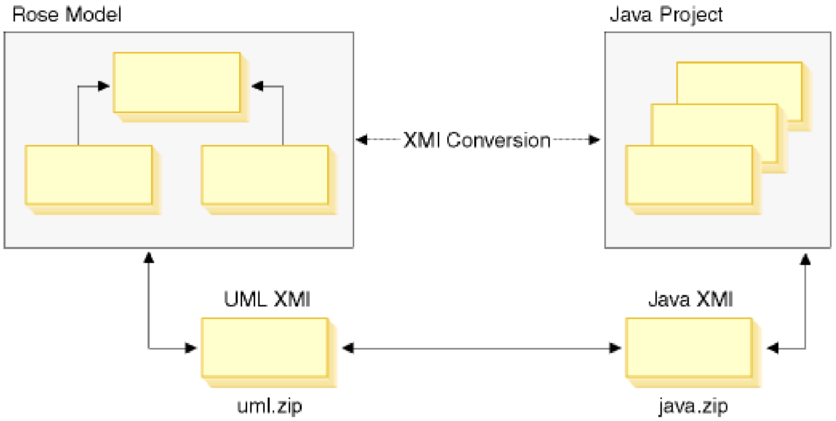
Using XMI Toolkit

A typical scenario would be as follows:

1. Develop an object-oriented analysis model for your application using Rational Rose.

2. Use the XMI Toolkit to generate Java code from the Rose model.
3. Update and modify the generated Java source in VisualAge for Java's IDE.
4. Generate a new version of your model to reflect the changes that were made in VisualAge for Java.

The diagram below shows the steps involved in the conversion process between a Rose model and the generated Java source. The broken line in the upper portion of the diagram shows how you can conceptually envision the conversion process between a Rose analysis model and the corresponding Java source files in a Java project. The solid lines in the lower portion of the diagram provide a more detailed flow of how the XMI conversion is actually carried out.



The steps are as follows:

1. The Rose model is converted into one or more XMI files that contain the UML XMI representation of the model's structure. The UML XMI files are packaged into a single file named uml.zip. This file is written to the same directory where the corresponding Rose .mdl file is stored.
2. The XMI Toolkit then maps the UML XMI representation into an equivalent representation in Java XMI. The corresponding Java XMI files are stored in a single file named java.zip. This file is written to the top level directory ("project directory") under which the Java project source files are to be written.
3. Finally, the XMI Toolkit generates the Java source files corresponding to the representation in Java XMI.

The process also works in reverse. If you have already implemented an application in Java, you can use the XMI Toolkit to derive a corresponding Rose model. You can then work with the generated model file to document or enhance the application's design.

The XMI Toolkit user interface

The XMI Toolkit user interface consists of two components:

- The XMI Toolkit Browser
- The XMI Toolkit SmartGuide

You can use the XMI Toolkit Browser to examine the XMI conversion mapping between a Rose model and a Java project. The XMI Toolkit Browser displays the UML XMI representation of your Rose model in the uml.zip file corresponding to the model, and it displays the Java XMI representation of the Java project in the java.zip file corresponding to the project. The XMI Toolkit Browser can also be used to show differences between successive versions of the UML XMI representation of a Rose model, or the Java XMI representation of a Java project.

The XMI Toolkit SmartGuide is launched from the XMI Toolkit Browser whenever you initiate an XMI conversion between a Rose model and Java project. Use the XMI Toolkit SmartGuide to organize and specify the details of the conversion you want to perform.

The VisualAge for Java online help discusses the XMI Toolkit in more detail.

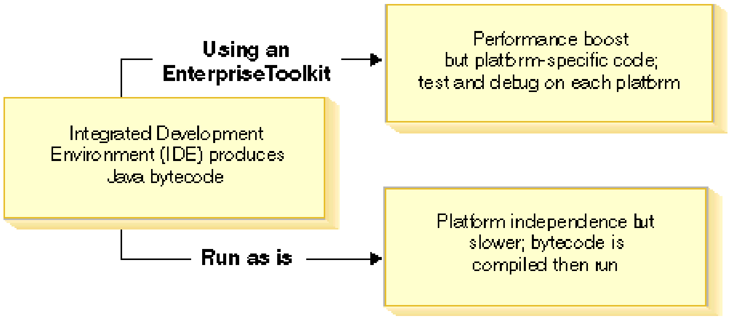
Chapter 11. Optimizing Java code for target performances

Enterprise Toolkits

VisualAge for Java, Enterprise Edition, provides Enterprise Toolkits that compile Java bytecode into code that is optimized for a particular target execution platform. The code that you develop with each toolkit can be debugged using the Distributed Debugger. Toolkits are available for OS/390 and AS/400 platforms.

Advantages of using an Enterprise Toolkit
Many people choose the Java language because it is platform-independent; that is, you can code and test an application once and run it anywhere. However, platform independence has a price: the bytecode that is produced is interpreted code, so it usually runs more slowly than compiled code. In small programs the performance difference will be negligible, but in large, computationally-intensive programs the difference may be significant. The Enterprise Toolkits provided with VisualAge for Java resolve this problem by generating compiled code that is optimized for the target platforms where the application will run.

Using a toolkit to boost performance involves a trade-off, as shown below.



Compiled code usually runs faster than bytecode, but it is platform-specific. If you process your application through a toolkit, but do not make any other changes to the code, you will have increased the speed at which your code runs, but still retained platform-independence. If you change your code to suit a particular target platform, after using a toolkit, however, you will no longer have platform-independent code.

When to use an Enterprise Toolkit
You should consider using an Enterprise Toolkit if you have a large, complex, or computationally-intensive application that will run on OS/390 or AS/400, and has one more of these characteristics:

- Performance is critical to your application.
- Your application is used repetitively. With compiled code, your application is not continually recompiled before execution, as it would be with a just-in-time compiler.
- Your application requires access to AS/400 databases or programs.

Enterprise Toolkits installation
The toolkits are optionally installed when you install VisualAge for Java, Enterprise

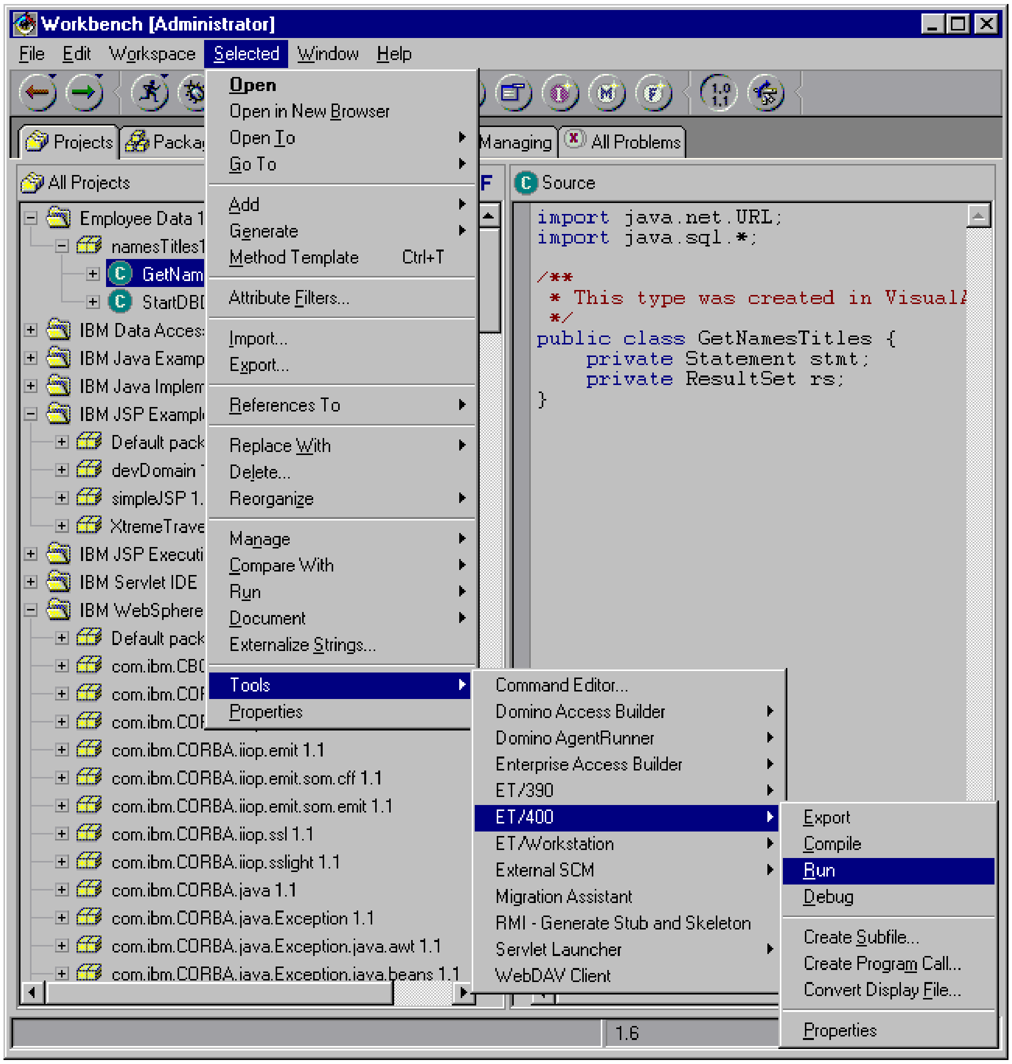
Edition. After installation, add each toolkit to the workspace before you can use it. For example, to add the Enterprise Toolkit for AS/400, in the Workbench, select **File > Quick Start > Features > Add Feature > IBM Enterprise Toolkit for AS/400**.

In the case of the Enterprise Toolkits for AS/400 and OS/390, some additional environment configuration is required on the AS/400 and OS/390, respectively. See the AS/400 and OS/390 README files, respectively, for more information.

Enterprise Toolkit startup

Each Enterprise Toolkit is started from the Workbench window. Select the project, package, or class that you want to work with and from the pop-up menu, select **Tools > Enterprise Toolkit.**

In the example below the Enterprise Toolkit for AS/400 has been selected, so you can see its functions (Export, Compile, Run, and Debug) and SmartGuides (Create Subfile, Create Program Call, and Convert Display file). You can select any of these functions or SmartGuides to start using the toolkit.



```
import java.net.URL;
import java.sql.*;

/**
 * This type was created in Visual
 */
public class GetNamesTitles {
    private Statement stmt;
    private ResultSet rs;
}
```

The Enterprise Toolkit for AS/400 (ET/400)

The native compiler and Java Developer Kit for AS/400 that are provided with OS/400® allow you to compile Java bytecode into classes that are optimized for the AS/400. By contrast, the Enterprise Toolkit for AS/400 (ET/400) that is provided with VisualAge for Java allows you to deploy your Java programs to the AS/400 from the IDE. In addition, ET/400 provides you with tools for accessing AS/400 resources from your Java client applications.

You can use ET/400 for tasks such as exporting Java class and source files to the AS/400 (source files are required to debug AS/400 Java applications from VisualAge for Java), and compiling Java code optimized for the AS/400. As well, you can use ET/400 to run Java code as interpreted bytecode in the AS/400 Java Virtual Machine (JVM), or as compiled programs, and to run and debug AS/400 Java applications from the VisualAge for Java IDE. ET/400 also provides you with two SmartGuides (wizards) that you can use to convert an existing display file object into Java code and call any AS/400 program object. ET/400 also provides a set of beans which can be opened in the Visual Composition Editor and used for accessing and formatting your AS/400 data.

You must install the Distributed Debugger before you can debug ET/400 applications. You use the Distributed Debugger to debug Java programs that you have exported to the file system, whereas you use the IDE debugger to debug applets and applications running in the IDE. With the Distributed Debugger, you can perform remote debugging (that is, you can run a program on one machine, and debug it on another machine) and debug optimized code.

The Enterprise Toolkit for AS/400 has four main functions:

Export Sends your code to the AS/400.

Compile
Compiles your code.

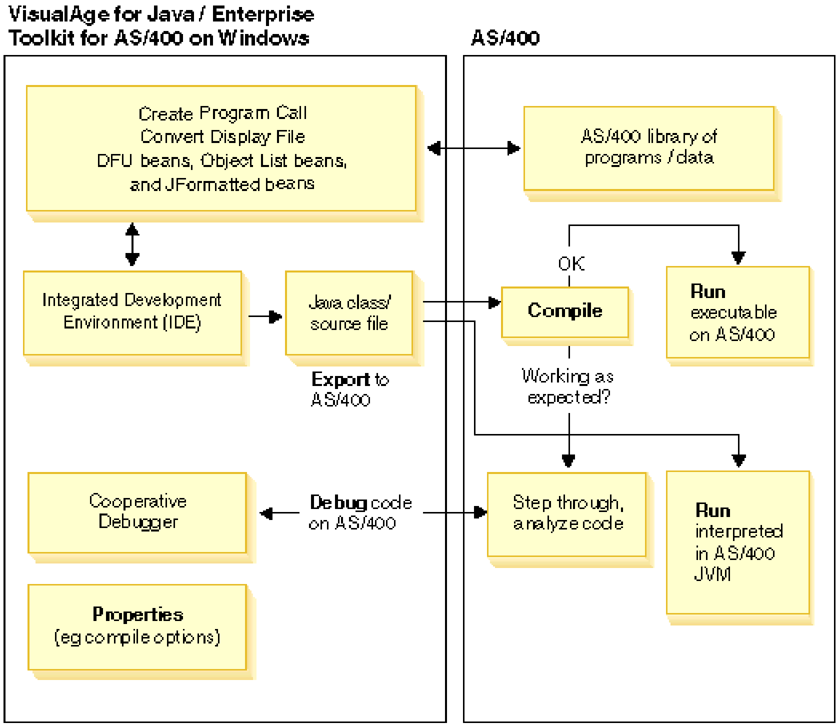
Run Run your code as a native AS/400 program, or as interpreted bytecode in the AS/400’s Java Virtual Machine (JVM).

Debug
Starts the Distributed Debugger tool so that you can debug code. You cannot debug applications unless you have installed the Distributed Debugger. For instructions on installing the Distributed Debugger, refer to the Installation and Migration guide, which can be found on the product CD. For instructions on using the Distributed Debugger, see the online help.

ET/400 also has two SmartGuides that you can use to work with AS/400 resources:

Create Program Call
Creates a Java class which will call any AS/400 program object

Convert Display File
Converts an AS/400 display file into Java code.


VisualAge for Java / Enterprise Toolkit for AS/400 on Windows
AS/400
Create Program Call
Convert Display File
DFU beans, Object List beans, and JFormatted beans
AS/400 library of programs / data
Integrated Development Environment (IDE)
Java class/ source file
Export to AS/400
OK
Compile
Run executable on AS/400
Working as expected?
Cooperative Debugger
Debug code on AS/400
Step through, analyze code
Run interpreted in AS/400 JVM
Properties (eg compile options)

ET/400 also has the following AS/400-specific beans:

DFU beans
The DFU beans, FormManager, ListManager, and RecordIOManager are a set of classes that extend the support of code to access one or more AS/400 database files. These beans allow you to map GUI forms, tables, and lists to AS/400 databases and manipulate (retrieve, add, update, and delete) database records.

Object List beans
The Object List beans, ET400List and AS400eList, provide a method for accessing AS/400 object names. These beans allow you to set listing properties for selecting the desired type of object list. Object list types include a list of libraries, a list of files within a library, or a list of user IDs on the system. The property settings can be used to access many different type of object lists.

JFormatted beans
The JFormatted beans include a set of utility classes that extend the support of code to convert AS/400 fields and attributes. Included in this set of beans are the following: JFormattedTextField, JFormattedLabel, JFormattedComboBox, DefaultFieldModel, AS400FieldModel, JFormattedTable, JFormattedTableColumn, JFormattedTextFieldCellEditorRenderer, JFormattedComboBoxCellEditorRenderer, and JFormattedLabelCellRenderer.

The VisualAge for Java online help discusses ET/400 in more detail and provides samples.

ET/400 properties

Before you can export, compile and run your code, you must set the properties for these functions in the Properties window. For example, you can set the optimization level for compiling, or select what type of files you want to export.

In ET/400, properties are inherited in a hierarchical manner. For example, if you set compiler options for a project and then you compile a class in that project, by default the class is compiled with the options that you set at the project level. To override this default, you would explicitly set compiler options at the class level.

To access the Properties window, select a project, package or class, then, from its pop-up menu, select **Tools > ET/400 > Properties**. You can find out more about the properties options by pressing F1 in the Properties window.

ET/400 and code exportation

You can export your code when you want to send Java files from the VisualAge for Java workspace to the AS/400. You can export source code (.java files) or bytecode (.class files); source code is necessary if you wish to use the Distributed Debugger. You can export entire projects or specific packages, classes, and interfaces. You can also export a GUI application, run it on the AS/400, and display it on an AS/400 client or a Network Station™.

The Enterprise Toolkit for AS/400 supports the standard Java interfaces. For example, an application targeted for AS/400 can use the Java Native Interface (JNI). Your applications can interoperate with AS/400 applications written in other languages such as C++. Similarly, an application can use Java Database Connection (JDBC) to access a database on the AS/400.

You must set the export properties in the Properties window before you can export your code. To export your code, select a project or package or class (all classes contained in the selected program element or elements will be exported), then, from its pop-up menu, select **Tools > ET/400 > Export**.

The files are exported to the workstation and then to the AS/400 Integrated File System. The exported code is still in the workspace and the repository.

Compiling, running and debugging an ET/400 application

When you compile your code, you are creating an AS/400 Java program from a Java class file. Before you can compile your code, you must export class files to an AS/400 Integrated File System directory and set your compilation properties in the Properties window.

If you decide to optimize your code, you will be creating an application for use outside the Java Virtual Machine (JVM). By contrast, an interpreted compile creates an application to be run in the JVM on the AS/400.

To compile your code, select a project or package or class, (all classes contained in the selected program element or elements will be compiled) then, from its pop-up menu, select **Tools > ET/400 > Compile**.

Run a compiled application
You can run your compiled application from the VisualAge for Java IDE. Before you can do this, you must export your code and set your run properties in the Properties window.

You can run applets, database, or Remote Method Invocation (RMI) applications. Standard input/output, that is, STDIN, STDOUT and STDERR, is routed from the AS/400 to the console.

To run your code, select a project or package or class, (if a package or project is selected, only the first class will be run.) then, from its pop-up menu, select **Tools > ET/400 > Run**.

Debug an application
The Distributed Debugger is a client/server application that enables you to detect and diagnose errors in your programs. This client/server design allows you to debug programs running on AS/400 systems accessible through a network connection. The debugger server, also known as a debug engine, runs on the AS/400 system where the program you want to debug runs. The debugger allows you to step through and analyze your code as it runs.

The Distributed Debugger debugs the Java application remotely, as it runs on the AS/400, from your VisualAge for Java workstation. The debugger allows you to step through and analyze the code as it runs on the AS/400. To use the debugger, you must export your source code, not just your class files.

To debug your code, select a project or package or class, then, from its pop-up menu, select **Tools > ET/400 > Debug**. You must install the Distributed Debugger before you can debug applications.

ET/400 SmartGuides

Create Program Call SmartGuide
The Create Program Call SmartGuide generates a Java class to call an AS/400 program. The program can be in any of the programming languages that are supported by the AS/400, such as ILE RPG, ILE C, or OPM COBOL. Through the Java class, you can pass arguments to the called program.

To create a program call, select a project or package or class, then, from its pop-up menu, select **Tools > ET/400 > Create Program Call.** The SmartGuide opens.

Convert Display File SmartGuide
This Convert Display File SmartGuide takes an existing Data Description Specifications (DDS) display file from an AS/400 and converts it to Java Swing code; that is, it converts it to a Display File bean with a graphical user interface.

To convert a display file, select a project or package or class, then, from its pop-up menu, select **Tools > ET/400 > Convert Display File.** The SmartGuide opens.

AS/400 Toolbox for Java

The AS/400 Toolbox for Java is a set of classes that provide access to AS/400 data and resources. Using these classes, you can access databases through the Java Database Connection (JDBC) driver, read physical and logical file records, manipulate AS/400 print resources, use files in the integrated file system, run AS/400 commands, access keyed or sequential data queues, and retrieve AS/400 messages.

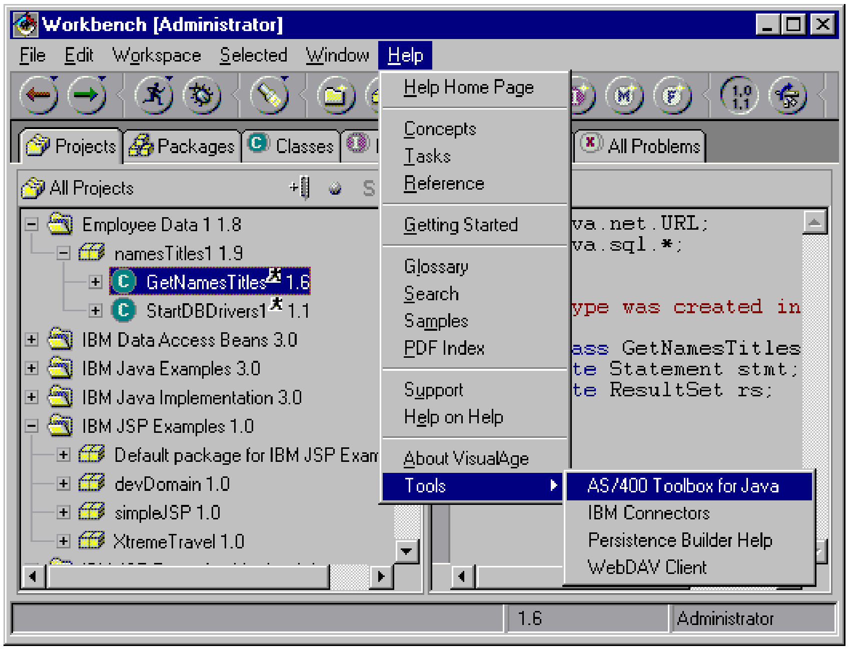
There are several types of classes offered in the toolbox:

- *Access classes* provide access to resources on the AS/400. For example, the JDBC classes provide access to relational databases on the AS/400.

- *Graphical User Interface (GUI) components* allow graphical manipulation of the resources. For example, the SQLStatementButton creates a button to execute an SQL statement.
- *Program Call Markup Language* (PCML) classes efficiently call AS/400 programs. You can call a program by writing a PCML script and then loading and deploying it from a Java program.
- *Security* classes make secured connections with the AS/400 and verify the identity of a user working on the AS/400 system.
- *HTML* classes quickly create HTML forms and tables.
- *Servlet* classes assist in retrieving and formatting data for use in Java servlets.

Many of these classes are themselves beans, and can therefore be visually manipulated through the Visual Composition Editor.

The AS/400 Toolbox for Java online help lists the specific classes in the access and GUI classes that are JavaBeans, and provides samples for writing applications with the AS/400 Toolbox classes. The AS/400 Toolbox help is accessible from the menu bar of the Workbench window. Select **Help > Tools > AS/400 Toolbox for Java**.



```
va.net.URL;
va.sql.*;

ype was created in

ass GetNamesTitles
te Statement stmt;
te ResultSet rs;
```

The Enterprise Toolkit for OS/390 (ET/390)

The Enterprise Toolkit for OS/390 (ET/390) provides support for developing OS/390 applications. Once you develop your Java bytecode using the IDE, you use ET/390 to export the bytecode to OS/390, and to invoke the binding process that creates optimized object code that runs in the OS/390 shell or CICS environment. The optimized code can be in the form of Java executable programs or dynamic link libraries (DLLs).

ET/390 also includes a Performance Analyzer to fine-tune your compiled Java application. The Performance Analyzer can help you understand and improve the performance of your Java programs. The Performance Analyzer traces function calls and returns, and collects timing data and call counts for each function called.

You must install the Distributed Debugger before you can debug ET/390 applications. You use the Distributed Debugger to debug Java programs that you have exported to the file system, whereas you use the IDE debugger to debug applets and applications running in the IDE. With the Distributed Debugger, you can perform remote debugging (that is, you can run a program on one machine, and debug it on another machine) and debug optimized code.

A sample ET/390 application would be a client/server application that invokes a CICS transaction to retrieve customer data stored on an OS/390 system. In this scenario, the client part runs on a workstation, and the server part runs on OS/390. The client code displays the user interface. The server part contains the code that invokes the CICS transaction.

A programmer would develop the server Java code on his workstation, then use the export and bind functions of ET/390 to create the executable that will run on the OS/390 system. The programmer would export the bytecode package that contains the server code to OS/390 and invoke the binding process. During the binding process, the OS/390 high-performance compiler compiles the Java bytecode into optimized object code, and binds it into executables or DLLs.

During run-time, the user enters a customer name in the user interface displayed by the client. The client invokes the server code which in turn invokes the CICS transaction to retrieve the customer data.

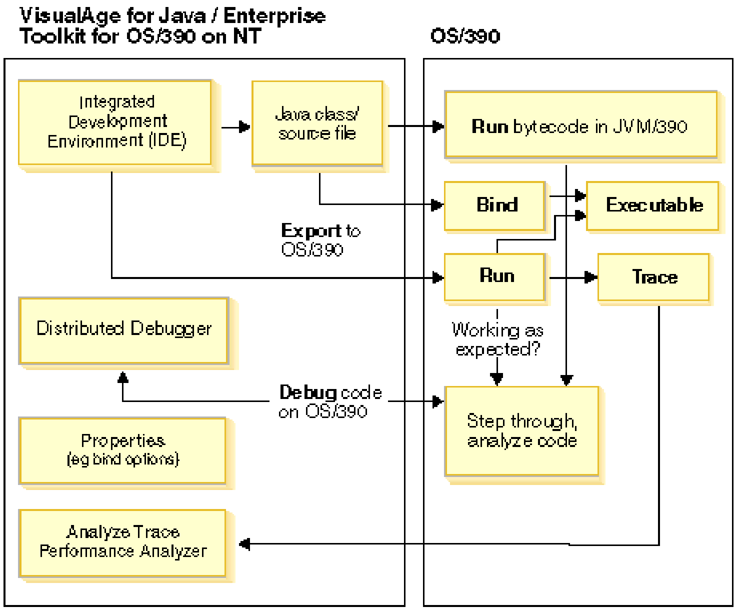
The Enterprise Toolkit for OS/390 has four main functions:

Export and bind
Sends your selected code to the OS/390 operating system and creates an executable file or DLL

Run Runs your code on the OS/390 operating system.

Debug
Starts the Distributed Debugger. You cannot debug applications unless you have installed the Distributed Debugger. For instructions on installing the Distributed Debugger on Windows, refer to the Installation and Migration guide.

Analyze Trace
Starts the Performance Analyzer, which you can use to analyze the performance of your compiled Java application.


VisualAge for Java / Enterprise Toolkit for OS/390 on NT
OS/390
Integrated Development Environment (IDE)
Java class/ source file
Run bytecode in JVM/390
Bind
Executable
Export to OS/390
Run
Trace
Distributed Debugger
Working as expected?
Debug code on OS/390
Step through, analyze code
Properties (eg bind options)
Analyze Trace Performance Analyzer

Host environment setup for the ET/390

After you install ET/390, you need to set up the host environment before you can build and run Java applications:

- Ensure the correct daemons and server processes are running on the host.
- Ensure that the ET/390 Java Install Data file on OS/390 contains the correct information. This file provides the host session's ET/390 Java installation information to VisualAge for Java, and is usually prepared by the system programmer.

Defining host sessions

Before using ET/390 to bind and run your OS/390 Java applications, you have to provide the OS/390 host session's information to VisualAge for Java. When you define the host session, ET/390 downloads a copy of the ET/390 Java Install Data file to the workstation and permanently saves a copy of it for your ET/390 development work.

You must define the hosts that you wish to work with before you can select them.

To define the host session options select **Workspace** > **Tools** > **ET/390** > **Host Sessions**. This option can only be reached from the Workspace menu because you are setting hosts for the entire workspace, not for just one project, package or class.

Storing host logon data

Before using ET/390 to bind and run your OS/390 Java applications, you have to provide OS/390 logon information. You can select to store host logon data, such as the userid and password, to be used when you log on to a selected host.

To define the host session options select **Workspace** > **Tools** > **ET/390** > **Logon Data.** This option can only be reached from the Workspace menu because you are setting logon data for the entire workspace, not for just one project, package or class.

ET/390 properties

Before you can export, bind, run, or debug your code, you must set the properties for these functions in the Properties window.

Every project, package, and class in the IDE can have its own set of ET/390 properties. You can use ET/390 Properties window to set bytecode binder options, run-time options, OS/390 environment variables, and Java command options. For example, you may want to automatically export your source code to the host system whenever you perform a bind action. Or, you may want to request trace information for use with the Performance Analyzer.

To access the Properties window, select a project, package or class, then, from its pop-up menu, select **Tools > ET/390 > Properties.**

Export and bind Java files with the ET/390

Exporting sends Java files from the workspace to the OS/390 operating system. You can export source code (.java files) or bytecode (.class files), either of which can be used to bind your application into executable files or DLLs. You can also export code to run in the OS/390 JVM. You can export entire projects or specific packages, classes and interfaces.

The Enterprise Toolkit for OS/390 supports the standard Java interfaces. For example, an application targeted for OS/390 can use the Java Native Interface (JNI). Your applications can interoperate with OS/390 applications written in other languages like C++. Similarly, an application can use Java Database Connection (JDBC) to access a database on OS/390.

Binding is the action that creates the executables or DLLs from the class that you exported; that is, it creates an application for use *outside* the Java Virtual Machine. Bind options are set in the Properties window. If you have development and production environments, you can specify options for each.

To export and bind your code, select a project or package, then, from its pop-up menu, select **Tools > ET/390 > Export and bind**.

Running, debugging, and analyzing ET/390 applications

You can also run your OS/390 applications from the VisualAge for Java IDE, whether in the form of executable files or DLLs.. Like bind options, run-time options are set in the Properties windows.

There are two Run functions: **Run executable** and **Run main.**

You should select **Run executable** if you want to run your application on the host machine. To do this, select a project or package (this option is not available with a class), then, from its pop-up menu, select **Tools > ET/390 > Run executable**.

You should select **Run main** if you want to run your application on your own workstation. To do this, select a project or package (this option is not available with a class), then, from its pop-up menu, select **Tools > ET/390 > Run main.**

Debug an OS/390 application
The Distributed Debugger is a client/server application that enables you to detect and diagnose errors in your programs. This client/server design allows you to debug programs running on OS/390 systems accessible through a network

connection. The debugger server, also known as a debug engine, runs on the OS/390 system where the program you want to debug runs. The debugger allows you to step through and analyze your code as it runs.

To debug your code, select a project or package or class, then, from its pop-up menu, select **Tools > ET/390 > Debug**. You must have installed the Distributed Debugger before you can debug applications.

Analyze trace
The Performance Analyzer traces function calls and returns, and collects timing data on call counts for each function called. It has two components - the host component and the workstation component.

The host component creates a trace of your Java program's execution in the OS/390 UNIX environment. The trace file is written to HFS or a sequential data set. The workstation component allows you to analyze the trace file that you have created on the host. VisualAge for Java provides graphical and textual diagrams to assist you with the analysis of the trace data.

To start the Performance Analyzer select a project or package or class, then, from its pop-up menu, select **Tools > ET/390 > Analyze Trace.**

Chapter 12. Accessing non-Java resources

Access Builders

VisualAge for Java, Enterprise Edition, provides *Access Builders* that you can use to access enterprise servers, data, and programs that are outside your Java programs. Access Builders offer a consistent programming interface for accessing multiple systems from a single Java application. They generate JavaBeans, which you can use in the Visual Composition Editor to connect to your user interfaces.

Access Builders are only available with the Enterprise Edition of VisualAge for Java.

VisualAge for Java, Enterprise Edition, Version 3.5 includes the following Access Builders:

Enterprise Access Builder for Transactions
Provides frameworks and tools that allow you to create Java applications that access existing host applications and data.

C++ Access Builder
Generates beans and C++ wrapper classes that let your Java programs access C++ DLLs.

Access Builder for SAP R/3
Creates Java applications, applets, and beans capable of accessing SAP business objects and data.

IDL Development Environment
Provides an environment for working with definitions written in the Interface Definition Language (IDL). It lets you manage and version IDL files like a VisualAge for Java project, and it can be configured to work with an IDL-to-Java compiler to facilitate compiler invocation and import IDL-to-Java compiler generated Java code.

Persistence Builder
Provides scalable persistence support for object models (Object models, represented by class hierarchies, are said to be persistent when instances created from these classes can be stored to an external data store such as a relational database).

Enterprise Access Builder for Transactions

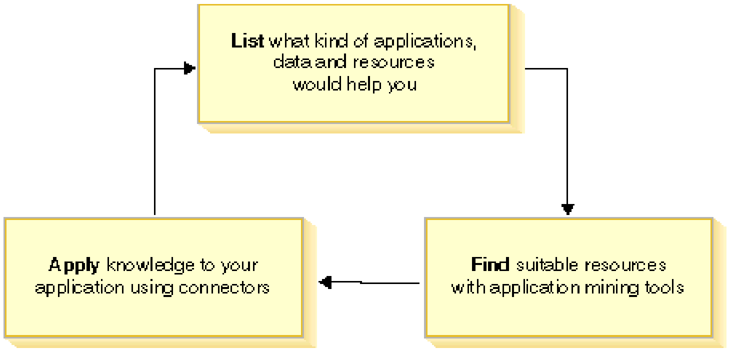
The Enterprise Access Builder for Transactions (EAB) can be used to create Java applications that access existing applications, data and resources. EAB lets you migrate to object-oriented technology while making full use of your traditional (non object-oriented) programs.

For example, you can use EAB to build a Java application that accesses existing CICS or MQSeries transactions.

The applications that you create using the Enterprise Access Builder can be exported to other development environments, such as Component Broker's Object Builder, or the WebSphere environment.

The application mining cycle

The application mining cycle consists of three actions: listing what resources would help you build your current application, finding those resources, and applying your knowledge using EAB connectors to access those resources.

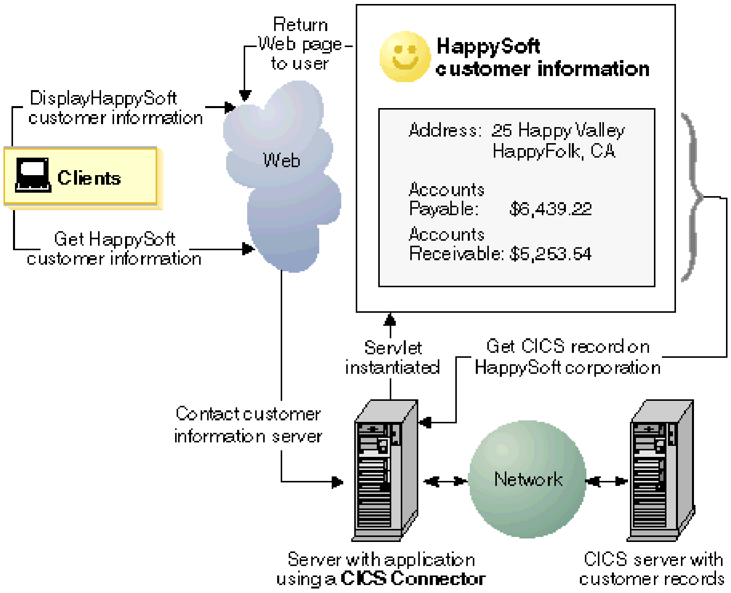

List what kind of applications, data and resources would help you
Find suitable resources with application mining tools
Apply knowledge to your application using connectors

Listing your resources involves listing resources critical to your business that you want to reuse to save you development time and limit the actual code produced. To **find suitable resources** you would examine resources such as applications, data, and servers to see how you can reuse them. **Applying your knowledge** involves reusing and further developing existing applications using connectors.

EAB Connectors

Connectors connect your application to existing resources; for example, a set of CICS customer records. The Common Connector Framework (CCF) is a set of interfaces and classes that provide a consistent means of connecting to and interacting with enterprise resources. Implementations of the CCF framework create a specific connector. For example, IBM has created a CICS Connector which accesses CICS records.

At run time the connectors are transparent to the users of your application. In the following diagram, a user gets customer information from a Java application developed with the CICS Connector. It displays a CICS customer record based on the corporation requested by the user.


Return Web page to user
DisplayHappySoft customer information
Clients
Get HappySoft customer information
Web
HappySoft customer information
Address: 25 Happy Valley HappyFolk, CA
Accounts Payable: $6,439.22
Accounts Receivable: $5,253.54
Servlet instantiated
Get CICS record on HappySoft corporation
Contact customer information server
Network
Server with application using a CICS Connector
CICS server with customer records

All connectors follow a common process and use the same visual builder tools. The common process and the same tools mean that whether you want to access CICS records, an Encina server, or MQSeries messaging software, you follow the same path and use the same tools when working with connectors.

The following is a list of the connectors provided in the Enterprise Access Builder:

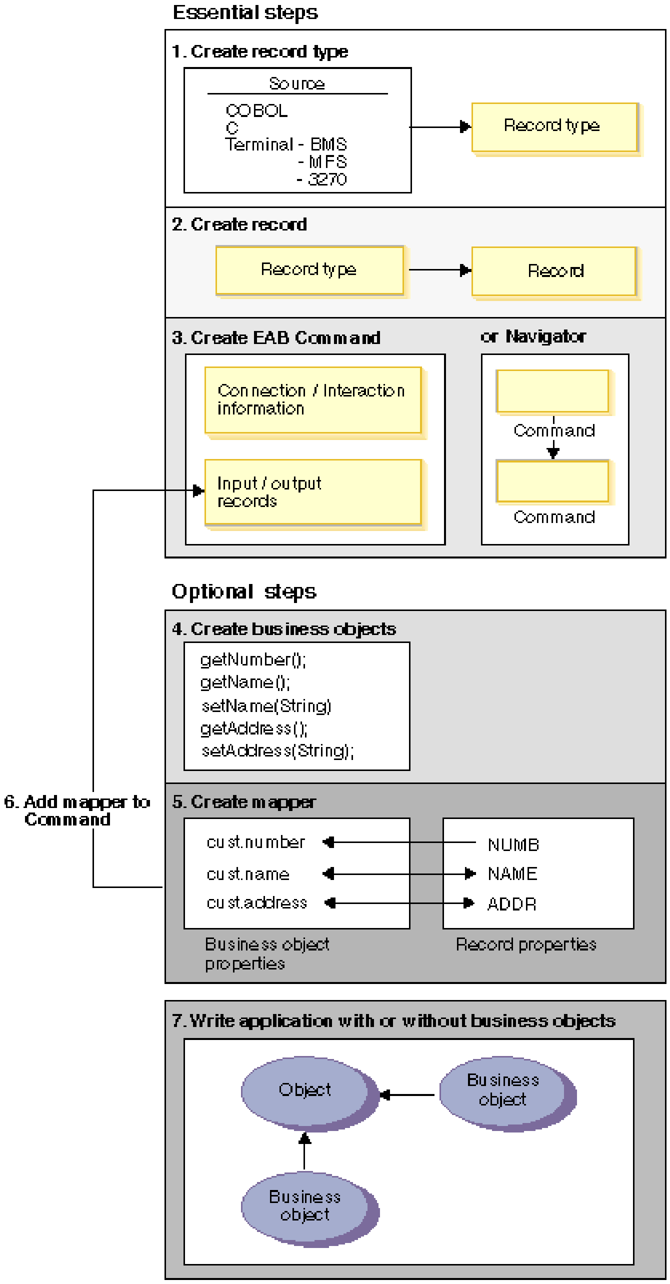
- **CICS Connector** - lets you develop applications that can access CICS transactions.
- **Encina Connector** - lets you develop applications that can make requests to a Distributed Computing Environment (DCE) or Encina server.
- **MQSeries Connector** - lets you develop applications that can access MQSeries messaging software on MQSeries servers.
- **IMS Connector** - lets you develop applications that can access IMS transactions.
- **Host-on-Demand (HOD) Connector** - lets you develop applications to access 3270, 5250, CICS and VT hosts from the Internet.
- **SAP Connector** - lets you develop applications to access the Business Object Repository in SAP R/3.

EAB components to build applications

The Enterprise Access Builder is based on the following constructs.

Enterprise Access Builder construct	Function
Records	Are used as input and output for Commands.
Commands	Send input data from Java applications to host applications for processing, and receive output data afterwards.
Navigators	Encapsulate a sequence of Commands for cases of complex interactions.
Business Objects	Represent entities in the Java application that have associated properties.
Mappers	Define the relationship between commands or navigators and the business objects by mapping record properties within commands or navigators to the business objects.

Applications that use connectors follow a common process and use the same EAB constructs. The following diagram shows the steps involved in creating a Java application using connectors and EAB.



Records

The first step in creating a Java application using an EAB connector is to create a record bean. Record beans are generated from record types. A dynamic record type is a representation of the field content of a record in an application. A field can be another dynamic record type, an array, or a simple field. You can use the SmartGuide corresponding to your source to parse a local copy of a source file and generate a dynamic record type.

Dynamic record types can be modified in the Record Editor. Once you have defined the fields of your record type, you can use the Create Record from Record Type SmartGuide to generate a record bean or class. Record beans define the fields in the server application that a Java client application will interact with. They can be used as input or output to a Command.

Record beans represent the fields in the server application that a Java client application will interact with. They can be used as input or output to a Command. The input and output data used by the Command is a Java class called a record. The record is a class within the Record Java Framework.

Refer to the VisualAge for Java online help and samples for more information about creating Records in the Enterprise Access Builder.

Commands

The second step in creating a Java application with EAB is to create a Command or a Navigator (A Navigator is a set of Commands).

A Command provides a single interaction between a Java application and a host system. When executed, a Command wraps a single interaction with a host system, as follows:

1. It takes input data and sends the data via a connector to the host system. A connector is a link to another system.
2. It then sets, as its output, the data returned by the host system.

The Command itself is a composite bean that follows a specific construction pattern.

You construct a Command using a SmartGuide. In a Command's creation, you provide the following information:

Information	**What it does**
Input Data	The input data is required by the Command to perform the execution.
Output Data	The output data is the result of the Command's execution.
Connection Information	This information specifies which set of classes will be used to communicate with the host system. A *connection specification* and an *interaction specification* define how a Command communicates with a host system. Use the ConnectionSpec interface to specify the connection to the host system. Use the InteractionSpec interface to specify to the Command which program to call through the connection derived from ConnectionSpec.

The input of a Command is defined by a record bean. The output of a Command is also defined by a record bean, though there may be multiple output record beans, which are defined as candidates. A Command can show an entire record bean in its interface, or only selected record bean properties.

Refer to the EAB online help and samples for more information about creating Commands.

Navigators

A Navigator consists of multiple interactions with a host system. Externally, a Navigator appears like a Command. However, a Navigator consists of Commands and other Navigators strung together to form a more complex interaction with the host system. When you execute a Navigator, the Navigator provides its input to the Commands and Navigators that compose it. Each Command in the Navigator is executed in the order specified by the Navigator. After the final interaction is complete, the output of the individual Commands and Navigators is available as the output of the Navigator.

You can specify connection information for a Navigator to override the connection information that you supplied with the Command.

You can construct a Navigator in the Visual Composition Editor. Refer to the EAB samples and online help for more information about creating Navigators.

Business objects

The next step in creating a Java application is to create one or more business objects. Business objects can be incorporated within your application just like any other set of objects, and provide an object-oriented abstraction of the data you are accessing.

A BusinessObject is derived from the com.ibm.ivj.eab.businessobject.BusinessObject class. A BusinessObject can also have a key which is derived from the com.ibm.ivj.eab.businessobject.BusinessObjectKey class. You can use them in your application to access your Commands and Navigators.

A business object makes core logic portable to other execution environments. You can use a business object to map many different host interactions with many different host resources. A business object can also contain any number of Java properties. There are two types of business objects: managed and unmanaged.

You can use the new Business Object SmartGuide to create business objects.

Refer to the EAB online help and samples for more information about creating business objects.

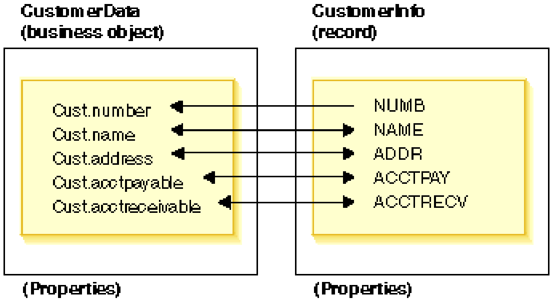
Mappers

The next step in creating a Java application with EAB is to create a mapper. To exchange data between a business object and a record used in the construction of a Command or Navigator, the Enterprise Access Builder uses objects called mappers.

A mapper is a class that maps the properties of one bean, such as a record, to or from the properties of another bean, such as a business object. A mapper allows the input record of a Command to get and set data automatically to and from a business object. The data can be set before, during, and after a communication interaction with the host application. You can also map from a record object to a business object or to multiple business objects. You create a mapper for each record that you need to access.

For example, you may have a business object called Customer and a Navigator that contains several Commands. Customer contains a property called address, used to store each customer’s address. During the execution of the navigation, one of the Commands results in an output record containing a field called address. A mapper between the Customer business object and the output record of this Command (mapping the address field in the output record to the address property of the Customer business object) would cause the address property of the Customer business object to be set to the output record address just prior to this Command’s completion. In this case, the mapping happens from the record object to the business object.

In the following diagram, the properties of the CustomerInfo record are mapped to the properties of the CustomerData business object. The mapper itself works in a similar way to this diagram. Using arrow icons, you connect a property from a record bean to a business object.



VisualAge for Java provides a Mapper Editor to map between a business object and records used in a Command or Navigator.

Lastly, you write your application using the business objects created earlier to interact with the resources you defined. After you have built your application, you also need to consider setting the run-time context and deploying it.

Refer to the EAB online help and samples for more information about creating Mappers.

Setting the run-time context and deployment

After you have built your application, you also need to consider setting the run-time context and deployment. When you deploy an an application, you are exporting it (along with any supporting run-time JAR files) to another machine. For example, you could deploy an application to a web application server (such as WebSphere) where users could access it over the Internet.

The run-time context refers to connector-related items at run-time. For example, managing the number of connections to a server at run-time in an application with many concurrent users is a critical design consideration. In such cases, you can override the default run-time context settings. Refer to the online help for more information about setting the run-time context and deploying your your completed application.

C++ Access Builder

VisualAge for Java, Enterprise Edition, includes a C++ Access Builder that enables your Java applets and applications to access C++ shared libraries.

The advantages of accessing C++ code from a Java client program include:
- You do not need to rewrite your existing C++ code in Java.
- You can take a phased approach to porting your C++ applications
- You can code features that are not currently available in the Java language.

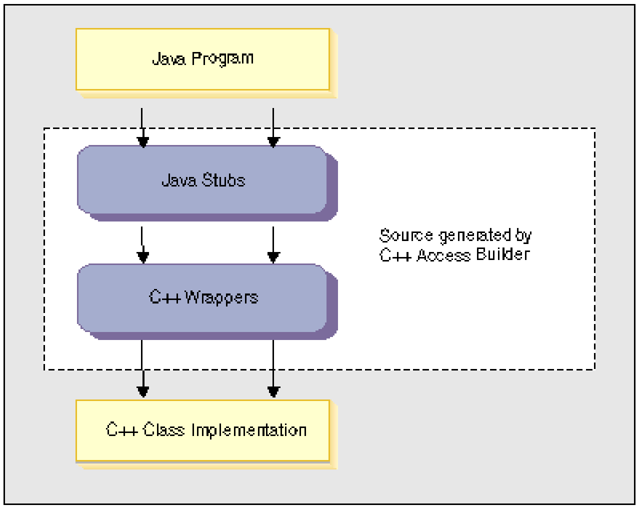
It can also be used if, for certain server applications, the performance benefits of binary code outweigh the platform independence offered by interpreted Java code. (In this case, another alternative would be to use the high-performance Java compiler that is included with VisualAge for Java, Enterprise Edition).

For example, you can use the C++ Access Builder to build Java clients that can locally access the services of a C++ server.

Generating the C++ wrapper and stub beans

The C++ Access Builder creates a bridge between Java and C++ by creating:

- A C++ wrapper for each accessible class in your C++ shared library
- A JavaBeans stub class that corresponds to each C++ wrapper
- A makefile that you can use to compile both the generated C++ source and Java source



To construct the C++ wrapper, you need to provide two essential pieces of information to the C++ Access Builder:

- A base name, used to specify the name of the shared library and other files used to build the library
- Header files, which contain definitions of all C++ classes that you wish to access from Java. For example:

```
class Rect: public Shape

{
public:
Rect(double width, double length);
virtual double area();

private:
double _width;
double _length;
};
```

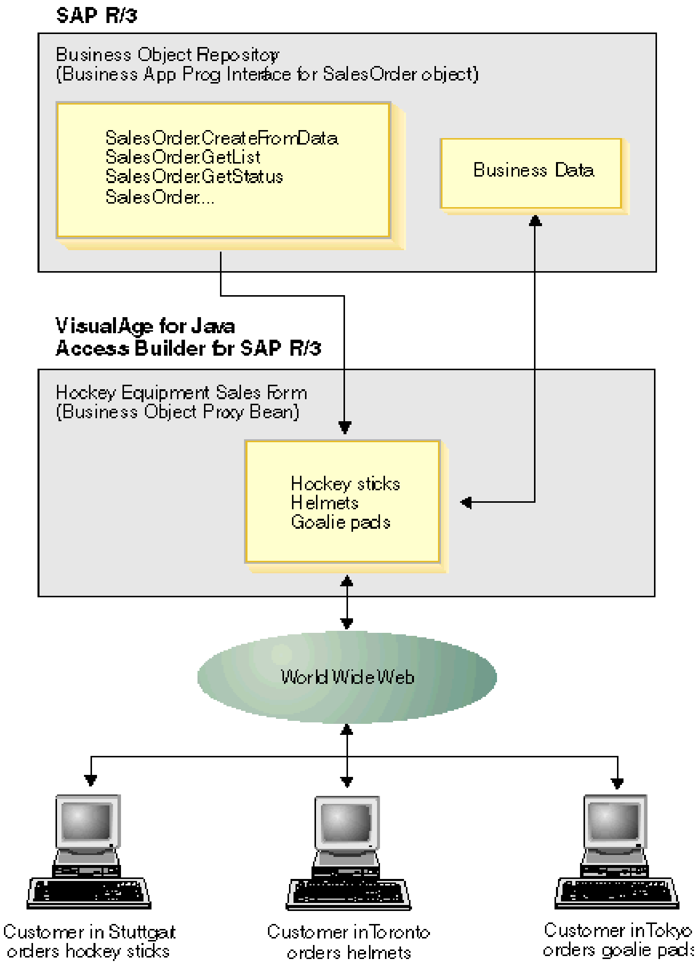
The name of the shared library is not required. In fact, the shared library does not even need to exist; the generated makefile includes commands to create the shared library based on the header file information.

The VisualAge for Java online help discusses the C++ Access Builder in more detail and provides samples.

Access Builder for SAP R/3

The Access Builder for SAP R/3 creates Java applications, applets, and JavaBeans capable of accessing an SAP system. Using the SAP Business Objects technology, which includes a Business Application Programming Interface (BAPI), applications can be quickly created in Java to access business data.

You begin using VisualAge for Java to create proxy beans which are derived from SAP business objects. Then you or your customers use the beans to access the business data in SAP, as shown in the diagram below.


SAP R/3
Business Object Repository
(Business App Prog Interface for SalesOrder object)
SalesOrder.CreateFromData
SalesOrder.GetList
SalesOrder.GetStatus
SalesOrder....
Business Data
VisualAge for Java
Access Builder for SAP R/3
Hockey Equipment Sales Form
(Business Object Proxy Bean)
Hockey sticks
Helmets
Goalie pads
World Wide Web
Customer in Stuttgart orders hockey sticks
Customer in Toronto orders helmets
Customer in Tokyo orders goalie pads

As shown at the top of the picture, the SAP system usually resides on its own server. It includes a Business Object Repository (BOR) plus the actual business data from a corporation. In the Business Object Repository are objects with associated methods typically used in business. Interfaces are already defined for them. Collectively, these interfaces are referred to as the Business Application Programming Interface (BAPI). In the diagram, the SalesOrder business object is shown along with its CreateFromData, GetList, and GetStatus methods.

Using the Access Builder, as shown in the center of the diagram, a Hockey Equipment Sales Form is created. This form is a proxy bean; specifically, it is a business object proxy bean. In the Hockey Equipment Sales Form are listed the items sold by the corporation: hockey sticks, helmets and goalie pads. The sales form is now ready to be used by the customers.

At the bottom of the diagram, the Hockey Equipment Sales Form is accessed by customers using the World Wide Web. The sales form passes their requests back to the SAP system, updating the business data controlled by SAP accordingly.

The online help discusses the Access Builder for SAP R/3 in more detail and provides samples.

Using SAP business objects and BAPIs

You use the Access Builder tool to browse meta information about SAP business objects and BAPIs of the SAP R/3 system. You can read the full reference of SAP business objects and BAPIs, disconnected from R/3, with any Web browser.

The Access Builder allows you to do these things:

- Retrieve complete meta information from the Business Object Repository (BOR) within R/3
- Keep meta information of multiple R/3 systems
- Access meta information locally without R/3 connection

The Access Builder for SAP R/3 generates proxy beans for SAP business objects, their BAPIs, and RFC modules. You can use these beans to design your application visually or to code an application or applet.

You will use the Access Builder to:

- Generate HTML documentation for specific SAP business objects and RFC modules
- Generate proxy beans for SAP business objects
- Generate proxy beans for RFC modules

Accessing the SAP system

The Common R/3 Access Interface defines a middleware-independent layer. You can leverage different ways to communicate with R/3 in your application without recoding. All generated beans are based on this interface and provide you with the same flexibility. The Access Builder for SAP R/3 includes an actual layer via Java Native Interface (JNI). This communication layer is best suited for the development environment, but can also be used for server-side Java or Java applications that are installed on fat clients.
You can also use another communication layer: CORBA/IIOP. This communication layer is often used for Internet/intranet environments with client-side Java.

The R/3 Access Classes build the basic run-time access to R/3 Business Objects and BAPIs. They are used by all other components of the Access Builder package and by the generated JavaBeans. You can use them, to dynamically access SAP Business Objects and BAPIs.

Logging on to an SAP system

To implement the SAP logon, you integrate the Logon bean into your application. You can take the following approaches:

- Include the user interface of an SAP Logon in your application
- Visually customize the logon panel with your default values using the Visual Composition Editor of VisualAge for Java
- Set the national language of the logon panel to your local needs

The online help shows you specifically how to log on through the VisualAge for Java interface.

IDL Development Environment

The Interface Definition Language (IDL) is a language-independent way of defining interoperable objects. Some IDL-to-Java compilers can use IDL source files as

input to generate Java bindings that can be used in Java programs. When properly used with a compatible ORB run-time, a Java program can be made to communicate with other ORB-compliant objects, regardless of the compiler technology with which the object was created. For example, a Java program may be able to communicate with an object that is written to the Common Object Request Broker Architecture (CORBA) specification.

VisualAge for Java, Enterprise Edition, includes an IDL Development Environment that works with an IDL-to-Java code generation tool, commonly known as an IDL-to-Java compiler. Using the IDL Development Environment, you can build support for the construction of CORBA-compliant applications over the Internet Inter-ORB Protocol (IIOP).

In the IDL Development Environment, you can work with your IDL source as well as the generated Java code in one convenient browser page. You can also leverage the built-in team and versioning capabilities of VisualAge for Java to help you maintain both your IDL source code and generated Java code.

You can accomplish a wide range of tasks to help you with your development activities:

- Importing directories and IDL files as IDL groups and IDL objects
- Creating new IDL groups and IDL objects or adding existing IDL groups and IDL objects from the repository
- Editing IDL source code
- Defining your own choice of an IDL-to-Java compiler
- Setting compile options and generating Java source code from IDL source code
- Managing versions, releases, and editions of IDL groups and IDL objects in a team development environment
- Exporting IDL groups and IDL objects as directories and IDL files

Using the IDL Development Environment, a Java ORB, and an IDL-to-Java compiler, you can build a variety of distributed client/server applications. For example, you can create IDL objects that enable Java programs to communicate with distributed CORBA applications, such as those commonly found in the finance industry.

A Java Object Request Broker (ORB) is a CORBA-compliant ORB that is implemented in Java. When used with IDL-to-Java tools, it enables you to develop Java applications that work with other CORBA-compliant objects. A Java ORB and an IDL-to-Java compiler is found in the IBM JDK. Visigenic and Iona also provide Java ORBs and IDL-to-Java compilers that you can use with the IDL Development Environment.

Typically, you would follow these steps to use VisualAge for Java's IDL Development Environment:

1. Set up the development environment to work with your choice of a Java ORB and IDL-to-Java compiler
2. Select or create a project for your IDL development work
3. Create the IDL groups and IDL objects to contain your IDL source code
4. Edit the source code of your IDL objects (if required)
5. Generate your Java code
6. Version your IDL groups and IDL objects
7. Version related packages and the project (as needed)
8. Use the generated Java classes to build your distributed application in the IDE

Refer to the online help for an IDL Development Environment tutorial.

Enterprise Access Builder for Persistence (Persistence Builder)

Persistence Builder provides a framework for building robust, scalable persistence support for object models. Object models, represented by class hierarchies, are said to be persistent when instances created from these classes can be stored in an external data store such as a relational database. As well as support for persisting objects, the Persistence Builder feature of Visual Age for Java includes additional frameworks to support transacting objects, enforcing object integrity through associations, and visual programming with a special set of palette beans.

Though object models for applications are often reused, the translation between object-oriented and non-object-oriented representations of business models has been a costly development task. Given that the majority of databases in use today are not object-oriented, the task of mapping objects to relational database tables and data from various existing sources has been the missing piece in object persistence standards. Persistence Builder supplies this missing piece by providing the following services:

- Automated code-generation services for underlying frameworks
- Import and export facilities for working with database schemas
- Debug and monitoring tools for performance tuning where desired
- Beans to assist visual programming and the building of a GUI to support the transactional semantics of a nested transaction application

These Persistence Builder services manage the following elements:

- **Business objects.** Business objects are typically the objects in your application domain that you want to persist in a data store, such as a bank application’s Customer and Account objects.
- **Associations.** In Persistence Builder, an association is a relationship that exists between two persistent business objects. There are three main types of association: one-to-one, one-to-many, and many-to-many. A primary function of Persistence Builder is to maintain the integrity of objects in the run-time workspace and the corresponding persistent data store through the support of atomic transactions. For example, a Customer object could have a one-to-many relationship with a collection of Account objects such that each Customer knows its set of Accounts, and each Account knows its Customer. The transactions necessary to manage this association are transparent to you.
- **Transactions.** Transactions represent paths of code execution. For example, a bank application might have a transaction that updates a customer account. The code activity necessary to manipulate the persistent objects would take place in the transaction.

Benefits of using Persistence Builder

- **Minimal intrusion to object and database design.** Close coupling between the object and database models is no longer necessary when you use Persistence Builder. Object models only need to represent their application domain, not the database design. This enables you to maintain an object-centric view of persistence rather than a data-centric view of data stores. The number of persistence constructs required in the application code is very low, keeping the persistence application model relatively lightweight. Separating objects and database concepts is done by employing meta-information. The loose connection between object model and database is preserved through mapping schemes.

  Persistence Builder supports this meta-information layer through UML object models, mapping models, and data models (schemas). Persistence Builder is integrated with the visual aspects of programming as well, such that existing view components can work with the framework.

- **High performance.** System performance depends on many factors: server load, network throughput, client application response time, and more. Database access can be optimized according to these performance factors.

  To optimize the number of database round trips and the path length to data, Persistence Builder's cache and preload framework enables you to define your working set of data for any task. This working set determines the amount of data that will be retrieved for a given task.

  When the set of objects you want to retrieve is very large, performance degradation occurs when instantiating the objects all at once. A cache framework prevents this problem by keeping objects in their native format as long as possible.

  The set of data required to perform a task is often known ahead of time, or can be derived from application hints. Minimizing the number of database round trips is achieved by retrieving as much data as possible. The preload framework enables you to tune performance in this manner. One example of this would be joining the relational tables that form a composition tree.

- **Advanced transaction support.** The transaction framework enables you to support multiple users executing concurrent and nested transactions that update the same objects. You can manage multiple versions of objects in your workspace and determine appropriate policies for handling collisions and commits to the database.The transaction framework treats the application and database memory spaces as if they were one memory space, synchronizing them and managing collisions.

- **Advanced query support.** The underlying query model is expressed in object terms rather than in native database-specific terms. Several Persistence Builder mapping strategies encourage this loose coupling.

  Persistence Builder produces SQL statements customized for your object model. The SQL statements are consumed by service classes that manage your data. The query generator supports:

  - Equi-joins for loading chains of objects (no branches)
  - Unions and set differences for loading complex composition trees
  - Left-outer-joins for loading trees that allow missing leaves

- **Association support.** Associations between persistent objects are implemented using first-class link objects that are hidden behind generated accessors. Links automatically manage:

  - Object referential integrity
  - Database key referential integrity
  - Persistent state of the relationship (lazy retrieval)

- **Seamless support for various database paradigms.** Large, commercial applications typically need to access data from multiple data stores. The relational and procedure-call generation embedded in the framework provides the linkages between the object model and the data store. These linkages are established through generated service classes and include generated queries or data access statements. All JDBC 1.1x-compliant drivers are supported.

  Table definitions can be read from an existing database schema.

  For each business object class, there is a set of pluggable data access services. There can be more than one services set per business object class. For example, a business object can have both a relational service set and a procedure call service set. Information sent during activation of the data store determines which service set a business object class uses.

For more detailed information on Persistence Builder and samples, see the online help.

Index

Notices

Note to U.S. Government Users Restricted Rights — Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

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

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

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

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106, Japan

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

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

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

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

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:

Lab Director
IBM Canada Ltd.
1150 Eglinton Avenue East
Toronto, Ontario M3C 1H7
Canada

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

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

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

COPYRIGHT LICENSE:

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

Each copy or any portion of these sample programs or any derivative work, must include a copyright notice as follows:

© (your company name) (year). Portions of this code are derived from IBM Corp. Sample Programs. © Copyright IBM Corp. 1997, 2000. All rights reserved.

Programming interface information

Programming interface information is intended to help you create application software using this program.

General-use programming interfaces allow the customer to write application software that obtain the services of this program's tools.

However, this information may also contain diagnosis, modification, and tuning information. Diagnosis, modification and tuning information is provided to help you debug your application software.

Warning: Do not use this diagnosis, modification, and tuning information as a programming interface because it is subject to change.

Trademarks and service marks

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

- AIX
- AS/400
- DB2
- CICS
- CICS/ESA
- IBM
- IMS
- Language Environment
- MQSeries
- Network Station
- OS/2
- OS/390
- OS/400
- RS/6000
- S/390
- VisualAge
- VTAM
- WebSphere

Lotus, Lotus Notes and Domino are trademarks or registered trademarks of Lotus Development Corporation in the United States, or other countries, or both.

Tivoli Enterprise Console and Tivoli Module Designer are trademarks of Tivoli Systems Inc. in the United States, or other countries, or both.

Encina and DCE Encina Lightweight Client are trademarks of Transarc Corporation in the United States, or other countries, or both.

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

ActiveX, Microsoft, SourceSafe, Visual C++, Visual SourceSafe, Windows, Windows NT, Win32, Win32s and the Windows logo are trademarks or registered trademarks of Microsoft Corporation in the United States, or other countries, or both.

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

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

Other company, product, and service names, which may be denoted by a double asterisk(**), may be trademarks or service marks of others.