IBM VisualAge for Java

by L. A. Chamberland
S. F. Lymer
A. G. Ryman

This paper introduces the IBM VisualAge® for Java™ product, a robust, visual suite of tools designed for rapid prototyping and enterprise application development. The paper outlines the development-time benefits of using VisualAge for Java.

Although the race to develop end-to-end Java development environments has just begun, the pace is accelerating. When tools first appeared, Java** developers were reluctant to move to them and away from coding directly with the Java Development Kit (JDK**). The criticisms of the early tools were predictable: for example, support for project management was clumsy, architectures were constrained, tools were not integrated, had limited code-generation facilities, and were "buggy" and slow. Tools or third-party beans started appearing for such niche markets as multimedia applets, database access, and remote connectivity. However, a hefty investment in Java architecture and code was still required to piece together these disparate parts.

In the summer of 1997, IBM released Version 1.0 of VisualAge* for Java, a robust, visual suite of tools intended to address these criticisms. The designers of VisualAge for Java had these goals in mind:

- Rapid prototyping. The programmer needs to experiment with fragments of code, developing applications more iteratively.
- Visual programming. Consistent with the industry's move toward visual design, programmers need to be able to visually create and manipulate all components.
- Open architecture. Generated code should fully support the JavaBeans** component model, allowing the programmer to create components that can be reused in other tool environments.
- Tool integration. Tools that coexist in an environment should be aware of each other and be able to leverage each other's strengths. Better integration of tools helps to shorten the development cycle.
- Robust code generation. Where possible, the development environment should help the programmer with code generation. Common types of code should be automatically generated.

This paper outlines how VisualAge for Java meets these goals, primarily through the following features:

1. Dynamic and iterative development, with simplified source control, interactive execution, incremental compilation and linking, and debugging support
2. Construction-from-parts paradigm using visual programming with JavaBeans support
3. Code generation by wizards, called "SmartGuides." The generated code can provide access to data and resources on enterprise systems as well as on Application System/400* (AS/400*) systems.
4. Enhanced context for development, with powerful browsing features and API (application programming interface) documentation framesets

VisualAge for Java was designed for complete, end-to-end Java development. Using its tools, programmers can achieve productivity gains that rival those

0018-8670/98/$5.00 © 1998 IBM

of other tool suites. Programmers can purchase VisualAge for Java Professional, which includes the core integrated development environment (IDE) and Visual Composition Editor, or VisualAge for Java Enterprise, which in addition includes the enterprise access builders.

Dynamic and iterative development

The traditional edit-compile-debug development cycle can take too long in today's competitive environment. By breaking down the sequential nature of this cycle, IBM's VisualAge paradigm makes the entire development cycle more iterative.

Simplified source control. With VisualAge for Java, the programmer no longer needs to be concerned about file management and frequent back-ups. As program elements are created they are automatically stored in the repository. When first created, the program element is also made available in the IDE workspace. Previously created program elements may be added to the workspace as needed. While the workspace contains only a single edition of a program element (called the current edition), the repository contains all editions. In fact, all the user source code is contained in the repository, including the current edition. When a program element is added to the workspace, the source is compiled by the IDE into Java bytecodes. Similarly, when a program element is created, the first "save" places the source in the repository and compilation is automatically initiated.

As well as the automatic creation of editions, it is possible to explicitly create versions of a class, package, or project. Open editions, denoted by time stamps, can be changed. Versioned editions, which may be given specific names or be automatically numbered, are fixed baselines of code that cannot be edited. Typically, the programmer works with an open edition until satisfied with a particular portion of code, then preserves it as a version. The version can then be used to create another open edition. This makes it possible to identify the completion of each stage of a project and ultimately to identify the contents of a particular release.

VisualAge for Java provides a simple way to revert to previous editions. If the user decides that the last code change is unnecessary or incorrect, selecting the "Revert to Saved" edit action causes the previous edition of the program element to be reloaded into the workspace. This maintains code integrity during development of the next release.

The IDE also provides a powerful comparison tool for objects in the repository. This tool is especially useful for comparing different editions of the same object. For example, when editions of the same class are compared, any differences in the class definition are shown, as well as a list of differing methods and the type of difference: source changed, added, or deleted. When a list element is selected, the source for each edition is displayed. Changes can be merged by simply copying and pasting from one source view into another.

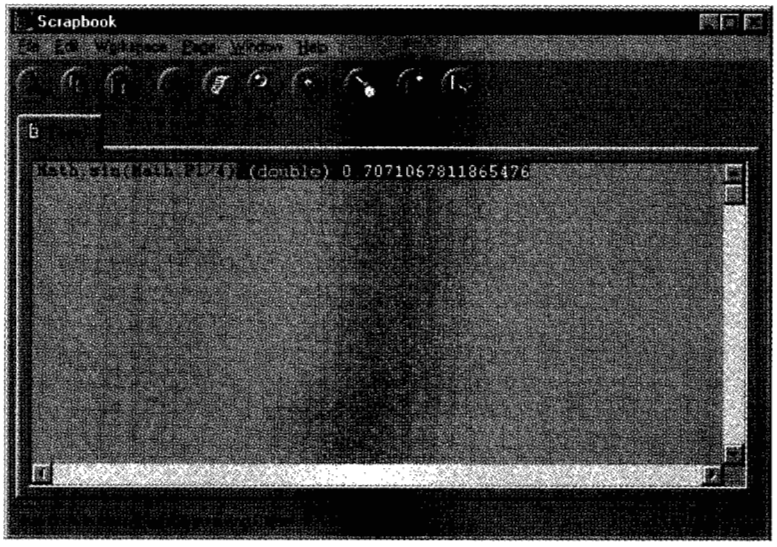
In VisualAge for Java Version 1.0, it is not easy for developers to share code. The user must either explicitly export the source or create and export an interchange file that also contains project and visual programming information. In this first release, it is also impossible to remove or purge program elements from the repository. (The user has to create and export an interchange file of the desired projects, then import this information into a clean repository.) These shortcomings are addressed with the team development capabilities available in a beta version of the next release of VisualAge for Java Enterprise. Programmers can share code repositories that reside on a server, and team server administrators can create new repositories based on existing repositories.

Interactive execution. A development environment that allows programmers to enter and immediately run code snippets, without the need for batch compilation and linking, is said to support *interactive execution*. Users of interpretative languages, such as BASIC, Smalltalk, Prolog, and APL, often claim greater development productivity than users of compiled languages, such as C and C++, because they use development environments that support interactive execution.

Interactive execution allows a program to be developed and tested incrementally, from the bottom up. The programmer creates new code and tests it interactively without having to develop special test programs. The productivity of users of compiled languages is constrained by the edit-compile-debug cycle, which can often take several minutes. In contrast, users of interpreted languages simply enter and execute statements with no perceivable delay.

In general, a program written in any language can be either interpreted or compiled. For example, BASIC can be compiled and C++ can be interpreted. The way a program is executed is determined by the

Figure 1 Scrapbook window



development environment. However, certain languages lend themselves to interpretation and others to compilation. In general, languages that allow very dynamic run-time behavior, such as self-modifying programs and automatic memory management, are easier to interpret, while those that have strong type systems are easier to compile. Java was designed to be interpreted, but its strong type system allows it to be effectively compiled.

Java programs are compiled into *bytecodes*, which are instructions to a virtual machine. A Java program can be executed by interpreting its bytecodes, one instruction at a time, in a software or hardware implementation of the virtual machine. Java can therefore be an interpreted language. Alternatively, a Java program can be executed by translating its bytecodes into machine code and running them directly on the target processor. Java can therefore also be a compiled language.

Bytecodes are typically much more compact than machine code, but execute 15 to 20 times slower. The translation to machine code can occur in batch mode, as is usual for languages like C++, or on demand as each method is invoked, as with languages like Smalltalk. On-demand compilation is referred to as dynamic compilation in Smalltalk and just-in-time (JIT) compilation in Java. Batch compilation is potentially more effective than per-method compilation because intermethod optimizations become possible. JIT compilation is used in Web browsers for executing applets, while batch compilation is useful for stand-alone applications or server programs (servlets, for example) where high performance is important.

The Scrapbook window. In VisualAge for Java, interactive execution is supported in the Scrapbook window. The Scrapbook consists of one or more pages, which can each contain code snippets. Pages can be saved as text files and reloaded, allowing the programmer to build up a library of useful snippets that can be shared and reused.

To work with a code snippet, the programmer highlights its text and then "pops up" a menu that contains commands to run, display, or inspect the snippet. Figure 1 shows the Scrapbook window after a user has entered an expression, highlighted it, and selected "Display" from the pop-up menu. The expression Math.sin(Math.PI/4) was evaluated and the result value (double) 0.7071067811865476 was displayed.

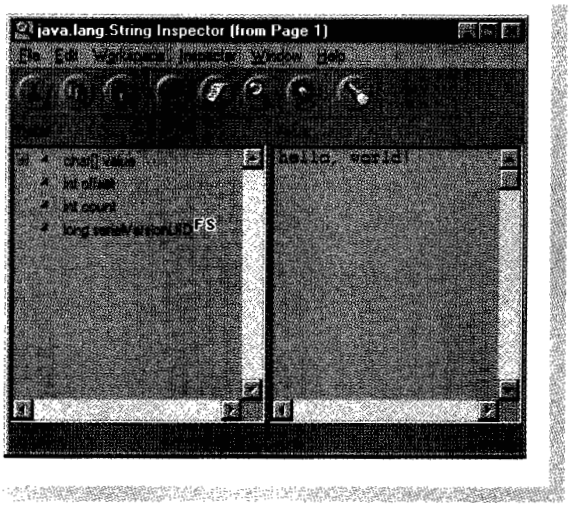
The run command simply executes a code snippet, which may include variable declarations, loops, and method invocations. The display and inspect commands are used for snippets that evaluate to a Java object. The display command evaluates the snippet and prints the result value as a text string in the page, after the snippet. The inspect command also evaluates the snippet, but rather than print the result value in the page, it opens an Inspector window for the object. The Inspector window allows the object to be explored in detail. Figure 2 shows the result of a user selecting a string ("hello, world!") in the Scrapbook window, then selecting "Inspect" from the pop-up menu. An Inspector window appears, displaying the String object and its characteristics.

Each page runs in the context of a Java class. The default class is java.lang.Object, but the page can be set to any class. Code snippets are evaluated in the context of this class. For example, in the context of the default class, the snippet Math.PI refers to a static variable defined in the Math class, but if the page is set to run in the context of the Math class, then the variable can simply be referred to as PI.

VisualAge for Java emulates multiple virtual machine instances. Whenever an instance of a class is run, either via its main method or as an applet, a new virtual machine instance is created for it. Each virtual machine instance behaves as a separate virtual machine. For example, each instance has its own set of static variables. The same considerations apply to code snippets that are executed from the Scrapbook window. Each snippet executes in its own virtual machine instance, providing a powerful and flexible development environment.

Consider a complex client/server application in which several clients can interact with each other via a server. This architecture is not limited to multiperson game applications; it is also useful in business applications, such as providing customer assistance over the Internet using a chat-like mechanism. For example, consider a team of customer service specialists that log on to a customer assistance server and work with customer requests. Web pages in the business application could contain links to the assistance server. When a customer needs help, selecting a link connects the customer with an available specialist. The customer and specialist hold a conversation using text, audio, or even video to resolve the problem.

Figure 2 Inspector window

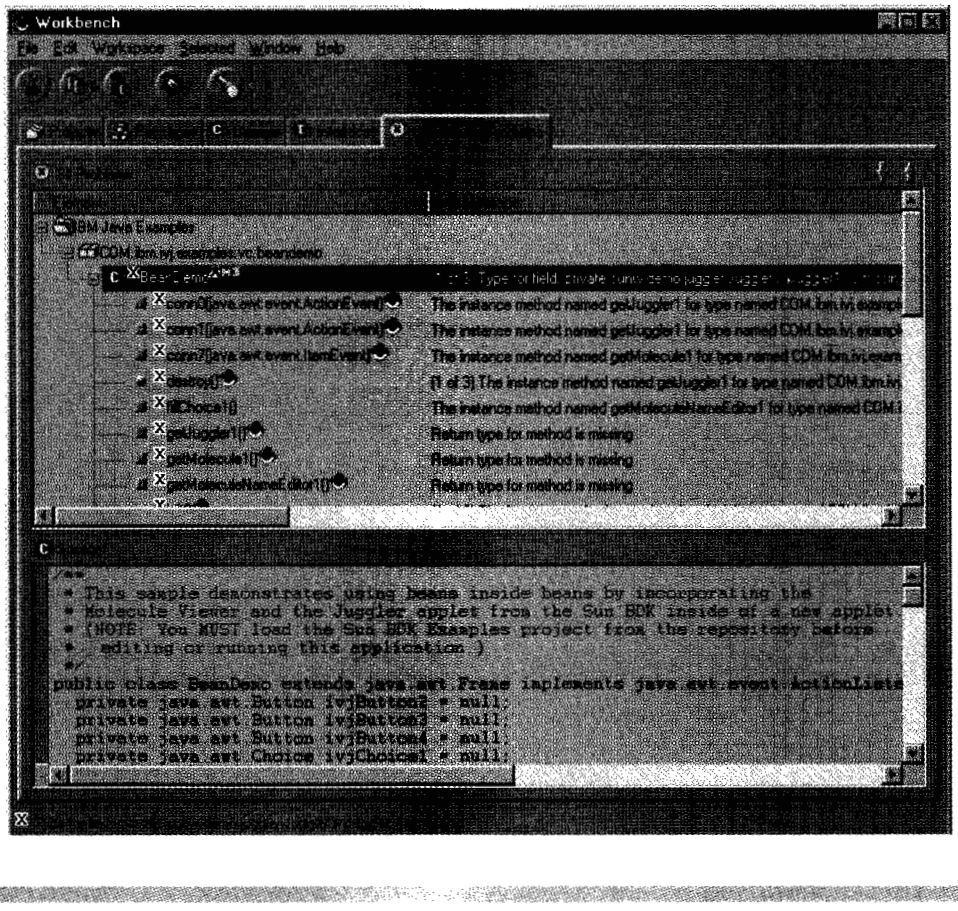


How would a programmer develop such an application? Using VisualAge for Java, the programmer simply runs an instance of the server class, then creates separate Scrapbook pages for each customer and specialist. A test scenario could have any combination of pages running concurrently in separate threads. Each customer and specialist, and the server, has a virtual machine instance. Using the debugger, the programmer can trace through the execution and resolve any interaction problems. Without this capability, the programmer would have to start multiple physical virtual machines and debug them remotely—a much more difficult task.

Incremental compilation and linking. Compiled languages like C++ require that the physical layout of objects be known at compile time. This allows efficient code to be generated, but whenever a class definition is changed, any class that uses it must be recompiled. If a low-level class is changed, virtually the entire application may need to be recompiled. In addition, linkers typically link an entire application as a batch operation; even though only a single object file may have changed, the entire application gets relinked. The compile and link time for a complex application can span several minutes or even hours. This time delay can seriously hamper development productivity because the effect of a small change to the source code cannot be quickly assessed.

Java was designed to avoid unnecessary recompilation. Java bytecodes do not contain information

Figure 3 The Unresolved Problems page of the Workbench



about the physical layout of the other Java classes they use. Instead, a Java bytecode file contains symbolic references to the methods and fields of other classes. These symbolic references are resolved at run time, when the class is loaded and executed. However, a change to a class may still affect classes that use it. For example, if a method is deleted, any class that uses it becomes invalid. Therefore most Java development environments recompile a class whenever any class used by it has changed. In addition, most Java Virtual Machines cannot incrementally load a changed class.

VisualAge for Java takes the inherent incremental capability of Java to its logical conclusion: rather than use a file-based compilation model, VisualAge for Java compiles source code incrementally whenever a field or a method of a class is edited and saved. The compiler checks the syntax of the source code and builds a list of methods and fields that it depends on. When these program elements are changed in the future, the compiler determines the impact and marks any invalidated methods. The Unresolved Problems page of the IDE (see Figure 3) lists all of the current problems in the code. The developer can navigate to the problem area and correct it. It would be very inconvenient if a developer had to ensure that any change to a method or field did not cause problems elsewhere. VisualAge for Java allows the developer to introduce inconsistencies in the code but still save the class and execute its instances. However, if an invalid method is invoked, the executing

thread is suspended and the Debugger window appears.

Permitting inconsistencies in Java code allows greater flexibility in development and is in sharp contrast with typical C++ environments, where a compilation error prevents object code from even being generated. VisualAge for Java allows inconsistencies to be introduced in other ways. When a class is imported, it may cause problems by referring to missing classes or by referring to methods or fields that have type mismatches with, or are missing from, loaded classes. Similarly, when a class is deleted, it will cause problems with program elements that use it.

Recall that VisualAge for Java emulates multiple virtual machine instances. This means that code may be changed during execution. If a method on the call stack of any thread is changed, the stack is popped back to that method and the new version is executed. If the fields of a class are changed and any live instances of the class exist, they are modified appropriately. In many cases, execution can proceed in a reasonable way. Of course, some changes will be so drastic that execution must be halted and restarted to obtain meaningful behavior. However, many typical changes to methods and fields can be incorporated into executing code in a satisfactory manner. This makes VisualAge for Java a powerful environment for developing long-running programs, such as servers. For example, suppose an error condition arises only after the server has serviced many requests. The program state may be very complex at this point. However, with VisualAge for Java, the code can be modified without restarting the server. This saves the developer from having to recreate the complex program state that caused the error condition each time the code is changed, resulting in greater development productivity. The next section discusses a particularly relevant example of this capability.

Debugging. Debugging support is a critical function for any development environment. A good debugger not only helps developers eliminate defects from a program; it can also help them understand the behavior of the program. There are three categories of debugging technologies, each having its use in practice:

- *Invasive debugging*—inserting print statements in the code and directing the output to the console or log files
- *Remote debugging*—attaching a debugger to another, possibly remote, Java Virtual Machine process and stepping through the code
- *Local debugging*—debugging multiple virtual machine instances running within the IDE, and modifying the code during execution

In the absence of a good debugger, developers resort to invasive techniques such as inserting print statements into their code. In addition, because Java code is often executed in target environments that do not enable remote debugging, developers are forced to use such invasive techniques to fix obscure problems that occur only in the target environment.

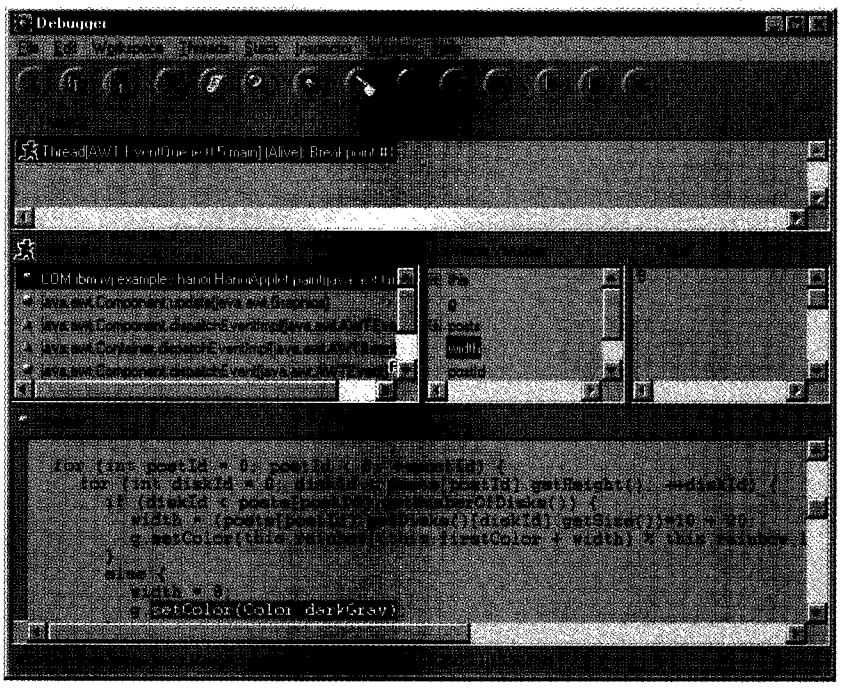
The Java programming system supports remote debugging. A Java Virtual Machine can be started in debug mode and allow an external process, such as another Java Virtual Machine, to control its execution. Unfortunately, the remote debugging support in JDK 1.0 had many problems that prevented the development of stable and robust remote debuggers. These limitations are being fixed in future releases. JDK 1.1.6 provides a much more usable remote debug API, with major changes coming in JDK 1.2.

Version 1.0 of VisualAge for Java does not support remote debugging, although this is currently under investigation for an upcoming release. Still, the VisualAge for Java support for local debugging goes well beyond the capabilities of the remote debugging specification, because it is fully integrated into the incremental compilation and linking capabilities of the environment. Java application systems that use several virtual machines and that would normally require remote debugging support (for example, client/server applications) can be executed within VisualAge for Java using its ability to emulate multiple virtual machine instances. Thus local debugging can emulate remote debugging.

The VisualAge for Java debugger can be activated in several ways:

- The developer can insert breakpoints in the code, which cause the debugger to be activated when they are reached.
- The debugger can be invoked programmatically using a support library.
- The debugger is activated when an invalid method is invoked, or when an uncaught exception is thrown.
- The developer can interactively open the debugger window and add any available thread.

Figure 4 The Debugger window



Individual threads can be selected for examination. When a thread is selected, its call stack is displayed. The developer can then select any stack frame and view the source code for its method and its list of local variables. While the source code for the method is displayed, the developer can edit it and save it, which causes the method to be automatically compiled and linked into the running program. Any local variable can be selected and its value displayed, or an Inspector window can be opened to work with it. The developer can step through the code as usual, resume execution, or kill the thread. The developer can also prune the call stack back to any method on the stack, and restart execution from that point.

Figure 4 shows the Debugger window. A breakpoint was set in the paint( ) method of the HanoiApplet class. While stepping through the code, the programmer can monitor variables. Here, the variable *width* is currently set to "8."

We now consider how these features come together in the important case of debugging servlets.

Debugging servlets. Just as applets are Web browser extensions, servlets are Web server extensions. Java servlets are analogous to Common Gateway Interface (CGI) programs, with the added advantages of improved performance and state management. Unlike CGI programs, servlets run in the same process as the Web server, eliminating process creation overhead. Each servlet is initialized once and remains active in the virtual machine, where it can respond to many HTTP (HyperText Transfer Protocol) requests. Each request is handled in a separate thread, which requires the programmer to write servlet code that synchronizes access to resources.

Suppose that a servlet is being developed in a Web server that uses a standard Java Virtual Machine, and that after many requests, a problem occurs. Because standard virtual machines cannot incrementally load a modified class, the Web server must be stopped, restarted, and retested to recreate the conditions under which the error occurred, every time a code change is made. Moreover, the developer

must work with the limited capability of a remote debugger or use invasive print statements.

Contrast the above scenario with development using VisualAge for Java. The developer:

1. Turns the development environment into a Web server by simply running the HTTPServer class that comes with the Java Servlet Development Kit (JSDK) from JavaSoft
2. Uses a Web browser to create the conditions under which the error will occur, and places a breakpoint in the servlet code that handles the HTTP request
3. Generates another request, which activates the Debugger window when the breakpoint is reached
4. Steps through the source code, inspecting variables as required, until the problem is found
5. Corrects the code, saves it (causing the running program to be incrementally compiled and linked), and resumes execution

A complex scenario can be simulated and debugged within the IDE, without the need for remote debugging.

Construction-from-parts paradigm

Component modeling and visual programming complement one another. Parts created based on a component model are manipulated in a visual environment.

Visual programming. VisualAge for Java offers a visual programming environment that is significantly different from others. Most visual building tools allow the user to define the visual layout of the applet or application by selecting "widgets" and control mechanisms from a palette. Some tools assist in generating code to handle particular events and relationships between these controls. This assistance typically takes the form of a wizard that generates code that is intended for a single use and is unable to handle arbitrarily complex behavior. Once the user customizes the code, it is difficult to modify or extend it using the wizard. There is also a complete separation of the visual composition or layout of the user interface and the logic that drives its behavior; that is, support for integrating nonvisual parts on a visual surface is missing.

However, the Visual Composition Editor in VisualAge for Java renders not only the visual elements of the user interface; it also gives the user the ability to visually program the applet or application behavior. Using visible connections between both visual and nonvisual elements, it is possible to define and integrate the logic for the user interface. Not only are visual elements displayed and manipulated; the user may add beans (components) with no visual elements to the canvas. These beans have a visual representation and may be manipulated like visual components.

The Visual Composition Editor (VCE) uses the JavaBeans component model[1] as the basis for composing and assembling beans. The VCE displays the features of each component the user adds to the canvas. Visual programming is accomplished by making connections between components to define behavior. All connections are made between features of beans: properties, events, and methods. The JDK 1.1 event model is fundamental to the operation of connections and the code that is generated by the VCE to provide the behavior represented in the visual design. This makes the Visual Composition Editor an ideal builder for creating beans and assembling applications.

Different types of connections provide different kinds of behavior. For example, an event-to-action connection may be used to invoke the method of another bean when a button is selected from the user interface. A property-to-property connection enables data to flow easily between visual elements and the underlying beans that drive the interface.

Another advantage of the connection paradigm is the ability to modify and manipulate connections. It is possible to have several connections originating from the same source component. These connections may be reordered to provide the proper sequence of events and actions. Connections are easily moved or even deleted. Connections can also be used to pass parameters and exceptions to other connections, or to return results.

Beans or applets can be composed in the VCE by adding AWT (Abstract Window Toolkit) components or the user's own beans or classes. A composite bean may even be created with no visual components. Components are added as either variables or instances. This allows the user to have complete control over the construction and instantiation of objects. On any event or action, VisualAge for Java can dynamically create any type of object with a *factory* bean, or let the VCE create the code to construct an object.

Once the object has the desired visual composition and behavior, the developer saves it to generate the source. The VCE provides a test facility, which executes the composed object from the Applet Viewer window of the IDE. As the behavior is being tested, it is possible to alter the underlying logic, change a connection, regenerate code, and save changes, and immediately test the result—all without closing the running applet.

It is apparent that the visual programming paradigm has many advantages, but it also poses some problems. Due to its visual nature, beans created using the Visual Composition Editor can be more difficult to maintain. Visual connections are not labeled, so a user may need to browse a list of connections or open individual connections to reveal source, target, and connection type information. Also, the sequence of connections is not obvious in the visual representation. Another weakness is the difficulty in creating conditional flow using the connection paradigm. The VisualAge for Java development team is aware of these problems and is considering enhancements to the Visual Composition Editor to address them.

To effectively use the visual programming paradigm, a good foundation in object-oriented design and the JavaBeans event model is essential. This may make it more difficult for users with a procedural background to get started with the Visual Composition Editor. However, users who take the time to understand these concepts and design specifically for this environment are rewarded. A good design that decouples the visual aspects of the application from the business logic and minimizes the number of connections can overcome many of the weaknesses of visual programming described here.

JavaBeans support. Java is widely known as a programming language for the Internet—"write once, run anywhere." However, Java is also a full-fledged object-oriented programming language. In keeping with the principles of sound object-oriented design, Java supports a component model that encourages code reuse. With the JavaBeans component model, a programmer can construct whole applications from reusable parts, with potentially large savings on the development cycle. What separates JavaBeans from other component models is its platform independence. Like the Java language itself, JavaBeans can be reused on any platform that supports a Java Virtual Machine.

One of the key design goals for the JavaBeans component model was to allow beans to be manipulated by visual application builder tools at design time. By using such tools, beans can be reused with minimal programming effort, which can quickly accelerate the development cycle. For example, consider a graphical user interface. Novice Java programmers avoid the complexities of coding and combining frames, panels, buttons, text fields, and layout managers by hand. Advanced programmers avoid the time invested in mundane widget manipulation, focusing instead on designing and coding the nonvisual, business logic.

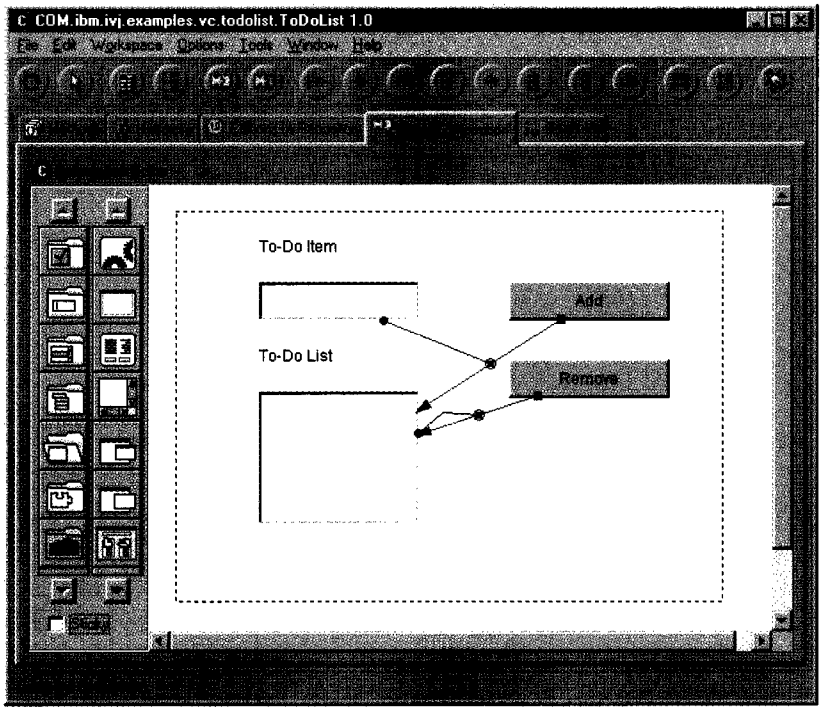
To help programmers prototype applications and beans quickly, VisualAge for Java provides visual support for several key JavaBeans features:

- Property editing
- Property sheets
- Introspection and customization

Preassembled beans. Any robust Java tool suite will include a set of preassembled beans, and VisualAge for Java is no exception. The Visual Composition Editor includes the beans palette, which provides beans that can be used to construct an application, an applet, or a more complex bean. The beans on the palette are organized in categories. The Data Entry category, for example, contains a TextField bean, a Label bean, and a TextArea bean. Beans selected from a category on the palette can be dropped on the *free-form surface*. As shown in Figure 5, the Visual Composition Editor provides a set of prepackaged beans that the programmer can work with. The left column of folder icons displays the bean categories. When a category is selected, the right column displays the icons of all beans contained in the category. (In Figure 5, the Containers category was selected and we see icons for Applet, Frame, Canvas, etc.)

In addition to the beans supplied by VisualAge for Java, programmers can add their own beans to the palette, as well as beans supplied by a vendor. Modifying the palette with additional beans can help increase productivity if these beans are used often. It also eliminates the need to know the exact class name of the bean. Once on the palette, these beans can be used in the same way that the built-in VisualAge for Java beans are used. For example, a programmer can import a third-party bean library into VisualAge for Java, and then add these beans to the

Figure 5 The Visual Composition Editor window. The beans palette is on the left.



palette. Customized bean palettes can be modified or reorganized as required.

Property editing. As instances are created from generic beans (by dropping them on the free-form surface, for example), the programmer will probably want to customize them. The simplest form of bean instance manipulation is property editing. VisualAge for Java provides visual property editing in the Visual Composition Editor. As illustrated in Figure 6, the *string* property of a button label can be changed without touching the code. Here, "Remove" replaces the previous value. When the bean is saved, the generated code is modified to reflect the new values. Note in Figure 7 the line of source code that reflects the change made from the free-form surface: ivjButton2.setLabel("Remove").

VisualAge for Java implements more detailed property editors in the form of dialogs. For example, the color editor shown in Figure 8 illustrates how powerful features can be simplified for the programmer through visual manipulation. The programmer can specify a color by selecting one of the preset basic colors, by matching a predefined system color (for example, the system desktop color), or by defining another RGB (red-green-blue) combination.

Figure 6 Editing the string property of a button label

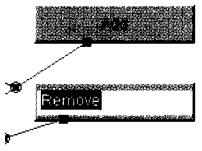
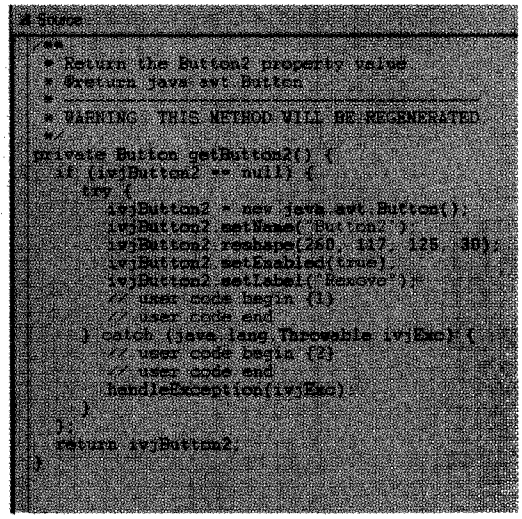


Figure 7 Generated source code for the button initialization



```
/**
 * Return the Button2 property value
 * @return java.awt.Button
 * ---------------------------------------------------
 * WARNING: THIS METHOD WILL BE REGENERATED.
 */
private Button getButton2() {
  if (ivjButton2 == null) {
    try {
      ivjButton2 = new java.awt.Button();
      ivjButton2.setName("Button2");
      ivjButton2.reshape(260, 117, 125, 38);
      ivjButton2.setEnabled(true);
      ivjButton2.setLabel("Remove");
      // user code begin {1}
      // user code end
    } catch (java.lang.Throwable ivjExc) {
      // user code begin {2}
      // user code end
      handleException(ivjExc);
    }
  }
  return ivjButton2;
}
```

The Visual Composition Editor can also be used to specify and modify the properties of connection parameters. Many of the built-in property editors in VisualAge for Java are accessible from bean *property sheets*.

Property sheets. Every bean instance includes a property sheet, which is an editable list of all public properties within the bean. A programmer uses a property sheet to set initial values. A property sheet can be thought of as a collection of all the property editors on a single bean instance. The property sheet of any bean instance that has been added to the free-form surface is available through the Visual Composition Editor.

Figure 9 shows a property sheet on a TextField bean instance. The value of any property can be edited by simply selecting the value field of the property. String properties can be changed on the sheet; for nonstring properties, a property editor can be invoked.

When developing beans in VisualAge for Java, the programmer may want to hide some of the bean's advanced properties from most users. By designating that these properties are to be used by an expert, the bean programmer protects the novice user from being overwhelmed by complexity. When the property sheet is brought up for an instance of a bean, the user can view and modify the basic properties, or select the "Show Expert Features" checkbox to see all of the public properties of the bean.

Especially for multiple instances of a bean, using property sheets can generate modifications much more quickly than working directly with the source code. In VisualAge for Java, the programmer can simply add the generic beans to the free-form surface and edit the property sheet for each bean instance. For example, the programmer adds several checkbox bean instances to a frame, and then edits the label of each checkbox in turn.

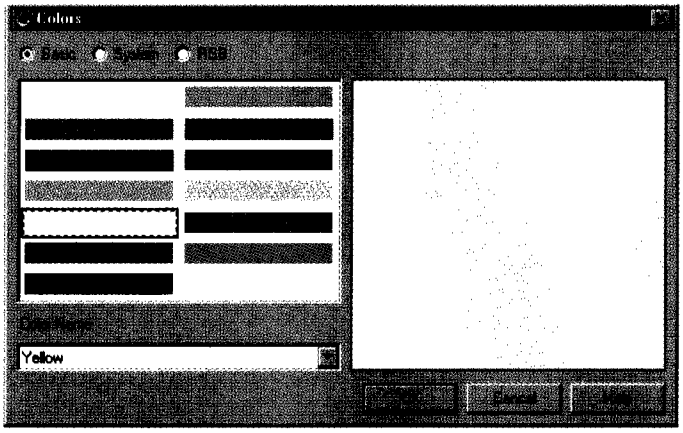
By default, the properties of the embedded beans are not visible in the property sheet of a composite bean. The bean designer can control which properties of the composite bean are visible (that is, public) by using the VisualAge for Java promotion feature. The bean designer can also promote methods and events of the embedded bean.

For very complex beans, then, the VisualAge for Java bean designer has strong control over a bean's public interface, through designating features as public or private and basic or expert, and through promotion.

Introspection and bean customization. For developers, the JavaBeans API includes facilities for seeing the internal structure of beans, generally called *introspection*. Low-level introspection services provide a wide variety of information that is useful to visual tools such as VisualAge for Java. High-level introspection services use the low-level services, but limit the information provided to the bean's public interface. The *BeanInfo* page in the class browser uses high-level introspection services to present views of the bean's public properties, events, and methods, as shown in Figure 10. Properties, methods, and event sets can be easily added or modified.

VisualAge for Java supports a variety of ways to generate and add code to an application or bean. Through the BeanInfo page, the programmer works with code from a bean perspective. Properties, methods, and event sets can be added in the context of the bean. The JDK 1.1 event model is directly supported: listener support can be added, and properties can be designated as indexed, bound, or constrained.

Figure 8 Color editor

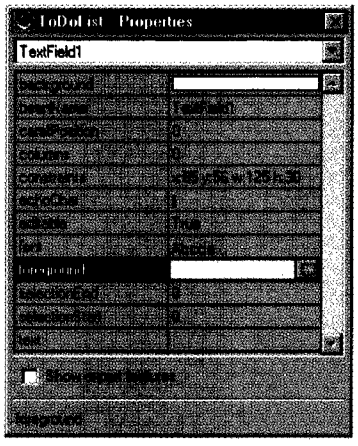


The JavaBeans API defines specific design patterns that support the JavaBeans reflection services. Low-level introspection facilities gather information about the bean's features by default if the standard design patterns are used. The programmer is free to follow a different convention, in which case a supporting BeanInfo class is highly recommended. While all beans generated by VisualAge for Java conform to the standard design patterns, the BeanInfo page provides a SmartGuide to help the programmer create a BeanInfo class.

Distributing JavaBeans. The standard for distributing JavaBeans is to package all related classes and resource files in a JAR (Java archive) file. As with zip files and other compressed file formats, the JAR format allows the programmer to package many files into one file and to significantly compress the size. Both of these features make download time from an HTTP connection much quicker. As well, the JAR file format allows the programmer to add a digital signature, specifically designed to make applet downloads safer through authentication. VisualAge for Java supports the importing and exporting of JAR files from the IDE. (See Figure 11.)

As discussed earlier, testing of beans is integrated in VisualAge for Java, thus a separate testing tool (for example, the Bean Development Kit [BDK] BeanBox) is not required. Beans can be edited, executed, and debugged, all within the IDE.

Figure 9 Property sheet of a bean

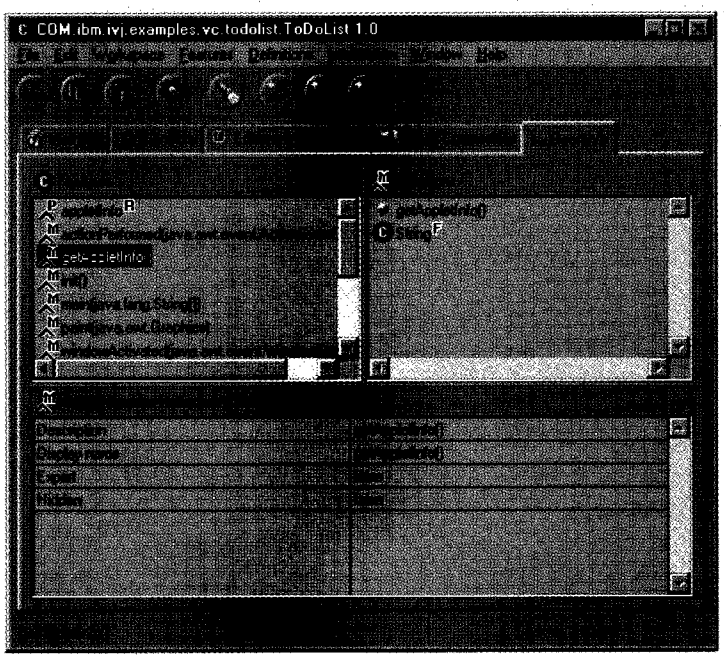


Code generation

The use of SmartGuides increases the pace of the development cycle and can simplify difficult tasks. Coding middleware access via SmartGuides makes access of legacy resources a less daunting task.

SmartGuides. The VisualAge for Java IDE includes several SmartGuides, or wizards, that provide guid-

Figure 10 The BeanInfo page of a class browser



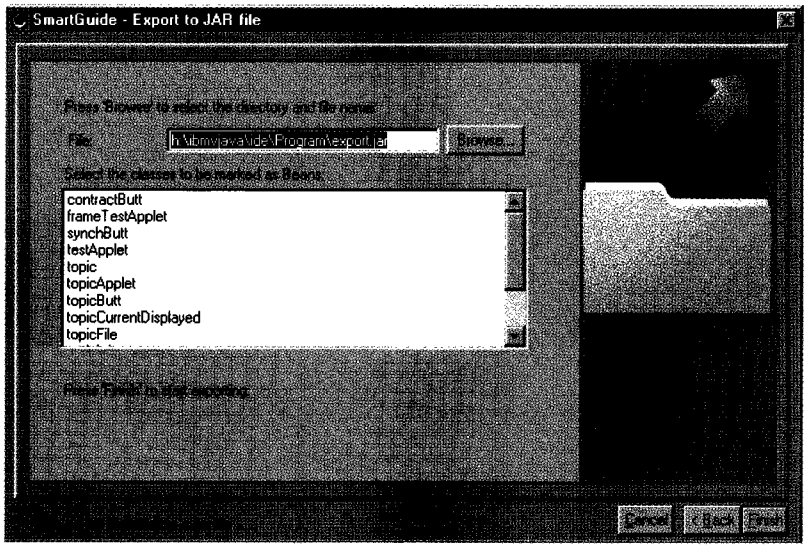
ance and assist the user in executing particular tasks. They replace conventional dialogs that provide little explanation and limited information as to how to complete the task, especially if there are different options possible. A SmartGuide is designed to lead the user through the completion of each supported task.

A SmartGuide consists of a series of panels. Users move forward through the panels to perform a task. Each panel presents information about choices and options, and guides the user through the necessary sequence of steps to get the task done. The user also has the opportunity to move back to previous panels to change or reference information already entered. Often default selections are provided, and choosing these defaults leads the user through the most typical usage scenario. It may not be necessary to visit all the panels in the SmartGuide. The SmartGuide automatically enables the "Finish" button as soon as the user has provided enough information to successfully complete the task.

SmartGuides are designed especially for the novice user, but with consideration for the expert user as well. The information about the task and the explanation of choices provided in each of these wizards teach the novice user about the task, so that the user quickly becomes more productive. For example, the Create Class, Create Method, and Create Field SmartGuides in the IDE allow the user to describe the characteristics of objects without needing to know the exact Java syntax. The SmartGuides also take care of dependencies between language features, so that the user is only able to select syntactically correct combinations. Appropriate Java code is generated by each SmartGuide.

SmartGuides assist the more experienced user by automatically creating or generating objects and code to meet particular requirements. It is then easy for the user to further customize the source to obtain the desired behavior. For example, the Create Applet SmartGuide allows the user to design an applet either by using the Visual Composition Editor or by hand-coding the source. If the applet is to be designed visually, the SmartGuide automatically creates the specified applet and opens a class browser directly to the Visual Composition Editor, with the

Figure 11 "Export to JAR file" SmartGuide helps package files for distribution



applet bean already instantiated on the free-form surface. The user can immediately begin to add visual elements to the applet as well as to create connections between both visual and nonvisual components.

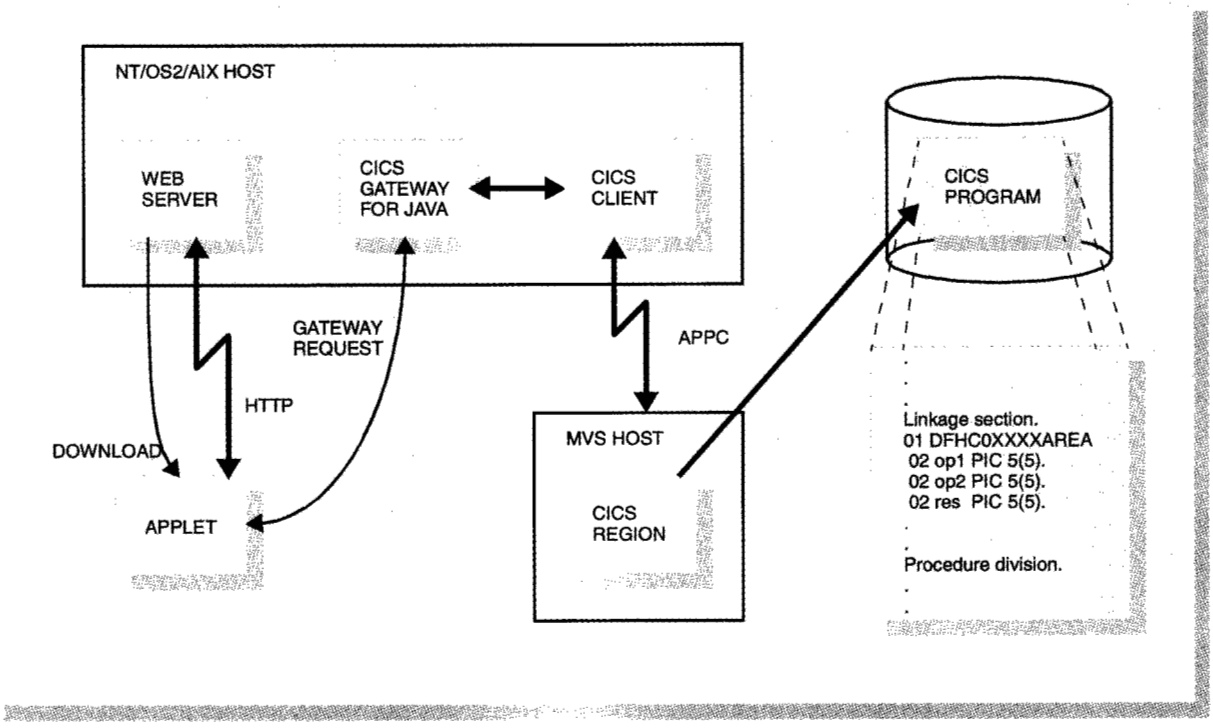
If the user chooses to hand-code the applet, the SmartGuide may still provide assistance. The SmartGuide provides the developer with choices to customize the wizard-generated applet skeleton. For example, the user may specify the kinds of events the applet will handle, the specific methods to be implemented, such as start, stop, or destroy, or the applet parameters to be generated. After the selected method skeletons are generated, the programmer adds business logic to this template to create the application.

Code generation for enterprise access. As Java enterprise access scenarios become more complex, programmers are under increased pressure to generate code that runs on a variety of middleware. Tools that help generate code for client/server access can greatly help productivity. The Enterprise Access Builders that are included with VisualAge for Java Enterprise generate access code, allowing the developer to quickly and easily access legacy applications, transactions, and data. Not only is the need to write middleware code eliminated; it is no longer necessary to rigorously test this highly error-prone code. The developer needs only to test that the generated code provides the desired behavior. In this way, the developer can focus on the application's business logic, rather than on generic access code.

Access to enterprise data and services is provided by a series of builders. Three key builders are: Customer Information Control System (CICS*) Access Builder, Remote Method Invocation (RMI) Access Builder, and Data Access Builder.

Each builder provides a SmartGuide, which leads the developer through the task of creating that particular type of enterprise access. The Data Access Builder also provides the user with editing capabilities for modifying the access code generated with the SmartGuide. All code generated by the builders supports the JavaBeans component model. These beans can be added to the client user interface during construction in the Visual Composition Editor. The following sections outline the capabilities of these three builders, and describe how the generated Java code may be easily utilized to provide access to enterprise data and services.

Figure 12 CICS access model



CICS Access Builder. The most difficult aspect of accessing most CICS transactions from Java is understanding and creating the code to convert between Java and COBOL types, and their representation at the system level. The programmer must write the code to marshal[2] and unmarshal the communications area (the data for the transaction) on and off the "wire." It is also necessary to understand and use the API available in the IBM CICS Gateway for Java libraries to write the access code.

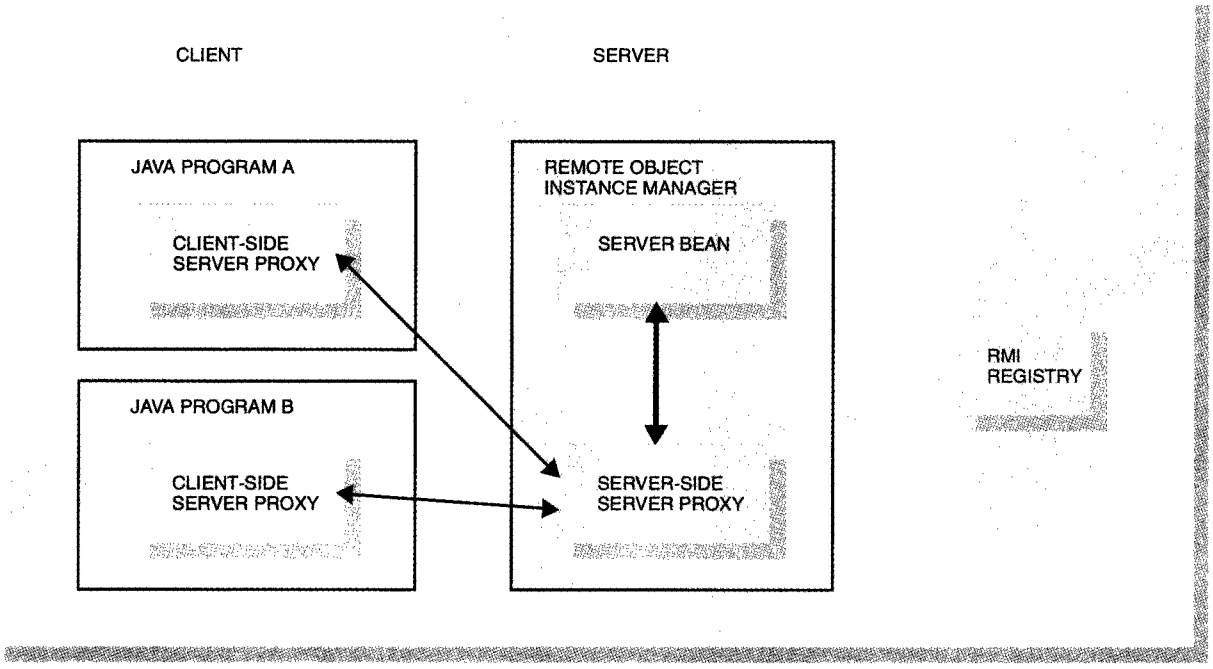
Figure 12 illustrates the interactions between a Java client and the CICS server. The flow of requests is:

1. The end user downloads the CICS access applet from an HTTP Web server.
2. The Java client sends Java ECI (external call interface) requests and transfers data for the COBOL communications area to the CICS Gateway for Java.
3. The CICS Gateway for Java forwards the Java ECI requests to the CICS client. Due to applet security restrictions, both the CICS Gateway for Java and the CICS client reside on the same host as the Web server.
4. The CICS client forwards information to the MVS* (Multiple Virtual Storage) CICS server, where the CICS transactions reside.

The CICS Access Builder makes it unnecessary to understand these interactions or to be skilled in writing this kind of code. The programmer only needs a copy of the COBOL program for the transaction to be accessed and the COBOL record that best describes the format of the communication area (COMMAREA) to be transmitted to the transaction. This may simply be the DFHCOMMAREA or another record in working storage.

The builder parses the COBOL data types in the COMMAREA and converts them to the appropriate Java types. The code to marshal and unmarshal the data is generated, and a bean is created with attributes for accessing each part of the COBOL record. This bean is used in conjunction with the CICS Unit of Work bean to fill the COMMAREA with the cor-

Figure 13 RMI access model



rect data, invoke the CICS transaction, and retrieve the results. The data elements of the COBOL record are represented as bean attributes, which allows easy connection to user interface elements.

The CICS Unit of Work bean is actually a class provided in the IDE called IVJCicsUOWInterface, used to access the CICS Gateway for Java from any CICS access program. This "wrapper" class manages all requests between the Java client and the CICS Gateway for Java. The Unit of Work bean allows the client program to:

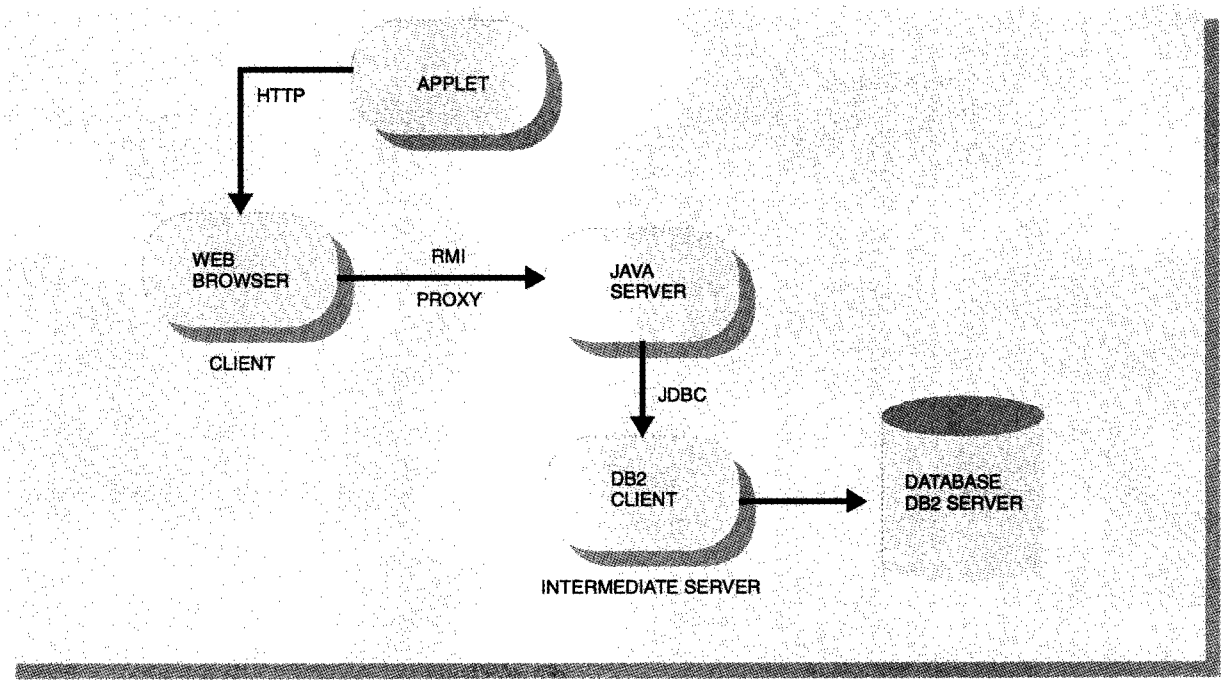
- Start and end a Unit of Work with the CICS Gateway for Java
- Commit or roll back changes introduced during this unit of work
- Pass the communications area bean to the host-based CICS program, to run the transaction both synchronously and asynchronously

The programmer adds the Unit of Work bean instance and the COMMAREA bean in the VCE. Then the properties on the Unit of Work bean need to be set for accessing the CICS gateway and MVS host. The COMMAREA can then be specified as a parameter to the Unit of Work bean. To facilitate testing, the IBM CICS Gateway for Java libraries are already loaded into the IDE workspace. These classes provide access through the CICS client to the transaction installed in the CICS environment on the MVS host. Thus, the full scenario implementation can be debugged in the IDE.

RMI Access Builder. The RMI Access Builder generates code for remote Java-to-Java access. Thus, understanding the RMI libraries provided with the JDK is no longer required. The programmer selects the class to be distributed and invokes the RMI Access Builder to create an instance of this class as a distributed object. The RMI Access Builder generates the server interface and a class representing an instance of the server. It also generates a client-side server proxy for the remote object in the form of a bean. The IDE generates the skeletons and stubs that provide access to the server. To connect this distributed code to the user interface, the programmer adds the client-side server proxy to the free-form surface and connects it to the visual beans.

Figure 13 illustrates the RMI access model provided by the generated code. The *server bean* is the bean

Figure 14 Thin-client view of three-tiered data access model



to be distributed and accessed remotely. Server bean methods can be invoked by a Java client program, and the server bean can generate events that are received by the client.

The *client-side server proxy* is a local representative of the remote server bean. The remote method access and event generation capabilities of the client-side server proxy allow the proxy to be treated as if it were the server bean itself. Because this proxy performs RMI initialization and the actual remote method invocation, the other code in a program does not need to deal with RMI code.

The *server-side server proxy* is a companion class to the client-side server proxy, and facilitates the communication of the client-side server proxy over RMI. The server-side server proxy is deployed on the server to access the server bean, and to relay events and exceptions from the server bean back to the client-side server proxy. In effect, server events are recreated on the client.

VisualAge for Java includes tools to test RMI code. The Remote Object Instance Manager is a general-purpose server that not only instantiates the RMI server objects, but also provides useful logs and statistics regarding access to the servers. When generating the code, the programmer can indicate that the Remote Object Instance Manager be started immediately after code generation and that the bean be instantiated. In conjunction with the RMI registry provided by the IDE, the Remote Object Instance Manager allows the user to completely test the client/server application within the IDE. Incremental development, testing, and debugging are thus possible.

Data Access Builder. In addition to a SmartGuide, the Data Access Builder provides integrated tools to manage both simple and sophisticated access to relational databases. The generated data access beans include code with the appropriate JDBC** (Java Database Connectivity) method calls, and also include a generated class that can be optionally used as a GUI (graphical user interface) prototype.

There are various ways to deploy a data access program. Figure 14 illustrates a three-tiered thin-client model. The following scenario describes its use:

1. The end user downloads a thin Java client from an HTTP server to a Web browser. The applet includes only distributed front-end code, not the data access business logic.
2. The Web browser communicates with an intermediate server through remote method invocation calls, using a TCP/IP (Transmission Control Protocol/Internet Protocol) protocol. The generated data access classes reside on this server. Because of applet security restrictions, the applet and data access methods must reside on the same machine.
3. The data access methods access the DB2* (DATABASE 2*) client through calls to JDBC, which also resides on the intermediate server.
4. The DB2 client connects to the JDBC-compliant database.

To assist the programmer in creating a schema representation of the database, the SmartGuide provides direction in selecting a database and mapping the database tables. SQL (Structured Query Language) statements can be used to query the database or perform table joins. Further refinement can take place by accessing user-defined methods for rows or collections of rows, either through SQL statements or database stored procedures. Unused methods and attributes can be deleted from the initial schema mapping.

When the schema mapping is complete, the programmer invokes the Data Access Builder to generate the data access beans. By default, the generated classes are named using the mapping name. For example, consider a schema called *Department*. The main Java classes generated for this schema would be:

- *DepartmentDatastore*, which represents and manages connections to the database
- *Department*, which represents a row from the mapping. This class contains database access methods, including user-defined methods, if defined.
- *DepartmentDataId*, which represents the set of columns that uniquely identify a row. This class is generated only if the mapping specifies at least one data identifier column.
- *DepartmentManager*, which allows a collection of rows to be selected and worked with. This class contains any user-defined manager methods.
- *DepartmentDataIdManager*, which allows a collection of data identifiers to be selected from the table and worked with. This class is generated only if the mapping specifies at least one data identifier column.
- *DepartmentAccessApp*, an executable GUI that integrates the basic features of the mapping
- BeanInfo classes that enable the Visual Composition Editor to use the generated data access classes by providing notification of changes to object properties
- Form support classes. These are prefabricated GUI components that reflect the basic features of the generated classes. These beans can be added in the Visual Composition Editor to help create the user interface that will access the data.

The graphical user interface class generated by the Data Access Builder, called an AccessApp (in the example, *DepartmentAccessApp*), includes a standardized layout that reflects the publicly accessible data and methods, as specified by the schema mapping. This simple GUI makes use of the generated classes to connect to data stores and manipulate data.

As shown in Figure 15, an AccessApp class is useful for demonstrating and prototyping database access or testing the generated classes before formal work on a user interface begins. An AccessApp class instance can be invoked as an applet or as an application. This class enables the developer to more rapidly develop a database access application. Its methods connect and disconnect from the database, and retrieve, update, and delete rows.

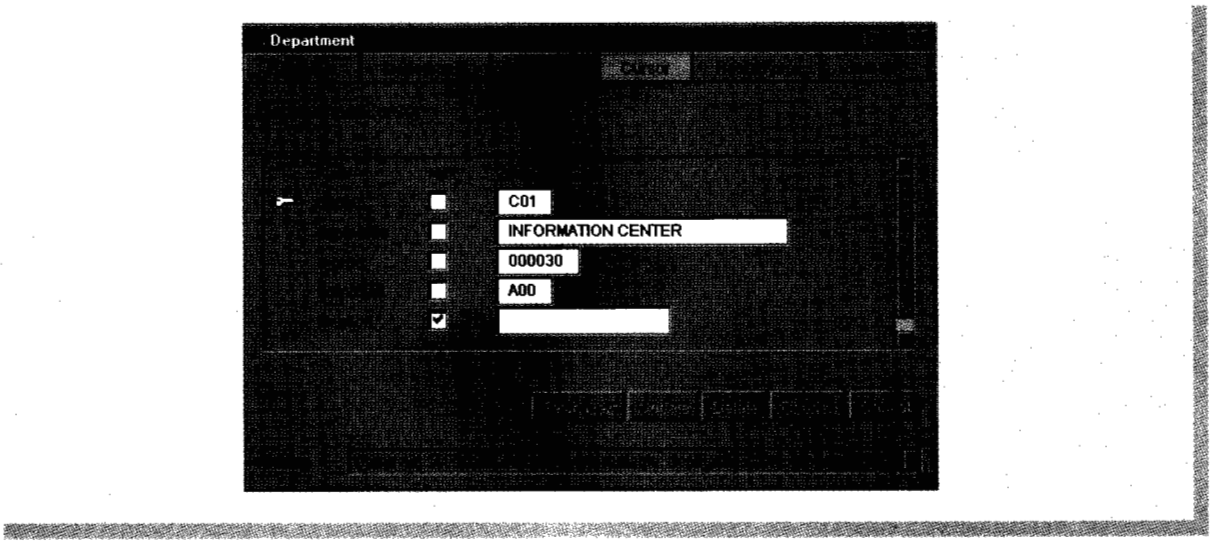
All of the generated beans are designed for use in the Visual Composition Editor. The developer can use the generated visual beans or create a more sophisticated graphical user interface using the data store, mapping, and manager beans.

Accessing the AS/400. One of the key additions to the VisualAge for Java 1.01 release is the support for accessing AS/400 data and resources. With this AS/400 support, Java clients can access AS/400 programs written in other languages, or Java programs can be created that run directly on the AS/400. A sound design approach for writing AS/400 Java programs is to do most programming using the workstation IDE, and then export the file out to the AS/400 for native compilation and debugging.

With this support, a programmer can:

- Convert AS/400 display file records to AWT classes. The Convert SmartGuide uses the Data Description Specifications (DDS) display files of existing

Figure 15 Generated AccessApp



language files (for example, COBOL or RPG files) to quickly generate AWT components.

- Generate Java code to access remote program calls to code written in other languages, such as RPG, COBOL, and C. From the AS/400 system name, program name, and parameters, the Create AS/400 Program Call SmartGuide generates the Java stub for the programmer, including handling of data conversions.
- Use the IBM Toolbox for Java API to access AS/400 resources and services. This set of 100 percent pure Java classes lets the programmer access such resources and services as AS/400 data, data queues, printers, and file systems. The API can be used for both client-side and server-side code. In addition, because this API is pure Java, no communications program (such as the AS/400 Client Access client) is required.
- Export Java source or bytecodes to the AS/400 integrated file system. Using the Export Java File SmartGuide, copies of Java source or class files can be exported from the workstation file system to the AS/400 IFS (Integrated File System). The AS/400-based Java Virtual Machine can then be used to execute the code.
- Compile Java bytecodes to AS/400 machine instructions. Through the Compile AS/400 Java Class File SmartGuide, exported class files can be recompiled into AS/400 native code. The optimization level can be set to alter the size and performance of the compiled Java program. The compilation can also provide performance data. While native code optimization makes the resulting code nonportable, it is ideally suited for server programs that require high performance. This SmartGuide provides a workstation client interface for the CRTJVAPGM command provided by AS/400 Version 4, Release 2.
- Remotely debug Java code that resides on the AS/400. This client/server debugger helps detect and diagnose problems in Java bytecodes or AS/400 native code. The debugger includes common features such as run, step, set breakpoints, and examine variables and call stacks. To use the debugger, both the compiled code and the Java source code need to be exported to the AS/400.

Enhanced context

At all times, the programmer needs a clear sense of where code is located in the environment. The powerful browsing features of VisualAge for Java provide this context.

Powerful browsing. As previously noted, the interpretive nature of the VisualAge for Java development environment helps the programmer transcend the limitations of the edit-compile-debug process that is so common among compilation environments. Still, a fundamental task of the Java developer is to search for or reference other code. For example, it

is necessary to review source code to resolve errors during compilation or to help isolate problems during debugging. It may even be necessary to review large amounts of code to learn about the program as a whole and how it behaves. This code may have been written by other programmers, and so may not be familiar to the programmer who reads it. To ensure that browsing does not become a bottleneck in the programmer's workflow, robust browsing facilities are critical.

The VisualAge for Java IDE includes a collection of browsers for viewing and manipulating projects, packages, and classes. The browsers are made up of panes of information. The contents of the panes may be lists, trees, properties, or source statements. The panes are linked together to allow information to flow from one pane to the next. This means that a selection in one pane populates the contents of other panes, using the context of that selection. This integrated flow of information facilitates browsing.

The IDE is an object-oriented environment, and each kind of object in the user interface has a dedicated browser (for example, projects, packages, classes, and methods). The main browser, the Workbench, provides a view of all the objects loaded in the workspace, while the Repository Explorer provides access to the complete contents of the repository. Each browser displays multiple tabs for different views of program elements.

For example, the class browser contains tabs labeled Methods, Hierarchy, Editions in Repository, Visual Composition, and BeanInfo. Each tab is devoted to a particular task associated with a class. The Methods tab provides a list of the methods for the class and a source pane enabled for editing. When a method is selected from the list, the method source code is automatically displayed and ready for updating. Similarly, the Hierarchy tab provides an extra pane that allows the user to browse the class hierarchy for this class. A selection from the hierarchy pane causes the methods for that class to be displayed in the next pane and the class definition to appear in the source pane. The Visual Composition tab provides access to the Visual Composition Editor so that the class can be defined using visual representations of beans. The BeanInfo tab allows the user to define and modify bean features of the class (properties, events, and methods). Finally, it is possible to browse other editions of the class by using the Editions in Repository tab.

With the ability to browse multiple objects concurrently, it is easy to reference other parts of the workspace and copy and paste information from one pane to another. The search facilities built into the IDE complement the browsers by enabling the programmer to search for any program element in the workspace. Searches can be based on definitions or references, and can be limited in scope to specific projects or packages.

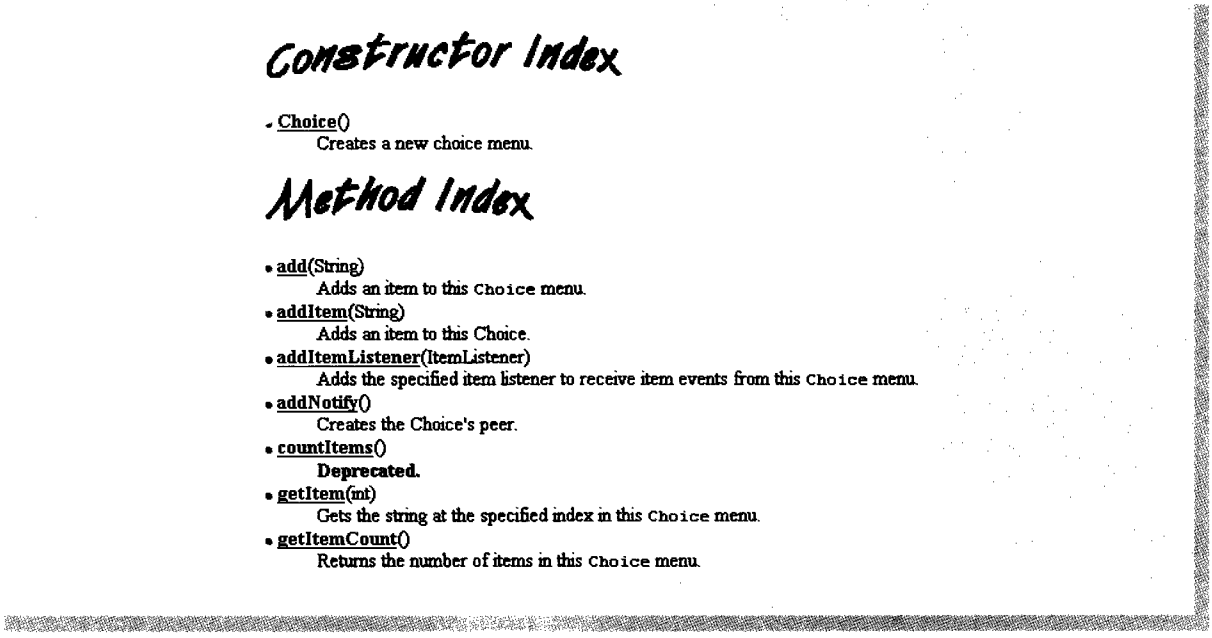
API documentation framesets. The Java language supports different ways to add comments to code. One form is known as a "javadoc comment," which is used to fold API reference information directly into the source code itself. These comments are placed in blocks at the tops of methods and classes, rather than being scattered throughout program elements. Javadoc comments are most commonly used to provide overview information and parameter descriptions for classes and methods. Other information about a method (for example, the class and package it belongs to) is not required because this information is already reflected in the structure of the source code. The comment writer does not need to document method signatures or inheritance hierarchies.

The JDK includes a command-line tool that extracts javadoc comments from the source code and generates a set of HTML (HyperText Markup Language) files that serve as support documentation. Hierarchy and index files are constructed, based on the structure of the code. Even if the programmer does not add javadoc comments, the javadoc tool generates a skeleton of the API or application that is specified as input. If the code is restructured or if new methods are added, a regenerated javadoc-based web reflects the changes. The JDK includes an HTML web that documents the JDK API.

While this web provides useful information, accessibility to the information can be improved. The single-page model produced by the javadoc tool directs the user to search for information in a linear manner; the user navigates up and down the API hierarchy with no contextual expression of sibling relationships. (See Figure 16.)

VisualAge for Java enhances the javadoc-based web by adding contextual information through a *frameset*. Using a javadoc-based web as input, an HTML postprocessor generates API framesets. As shown in Figure 17, when a package is selected, three panes are displayed:

Figure 16 Javadoc output from JDK



Constructor Index

- **Choice()**
  Creates a new choice menu.

Method Index

- **add(String)**
  Adds an item to this `Choice` menu.
- **addItem(String)**
  Adds an item to this Choice.
- **addItemListener(ItemListener)**
  Adds the specified item listener to receive item events from this `Choice` menu.
- **addNotify()**
  Creates the Choice's peer.
- **countItems()**
  Deprecated.
- **getItem(int)**
  Gets the string at the specified index in this `Choice` menu.
- **getItemCount()**
  Returns the number of items in this `Choice` menu.

1. The Class Index frame (upper left) lists all classes and interfaces in the selected package. The contents of this frame do not change until a different package is selected.
2. The Method Index frame (lower left) lists all constructors, methods, and variables in the selected class.
3. The Class Details frame (right) presents detailed information on the constructors, methods, and variables listed in the Method Index frame. This information is identical to that produced by the javadoc tool.

The Method Index frame presents a more concise list of the class constructs than is presented in the Class Details frame, letting the user scan it more quickly. If this is too little detail, the user can quickly select method names in the Method Index frame to bring up details in the corresponding Class Details frame. Context is never lost, and the user is not repeatedly moving up and down the hierarchy to find the desired method.

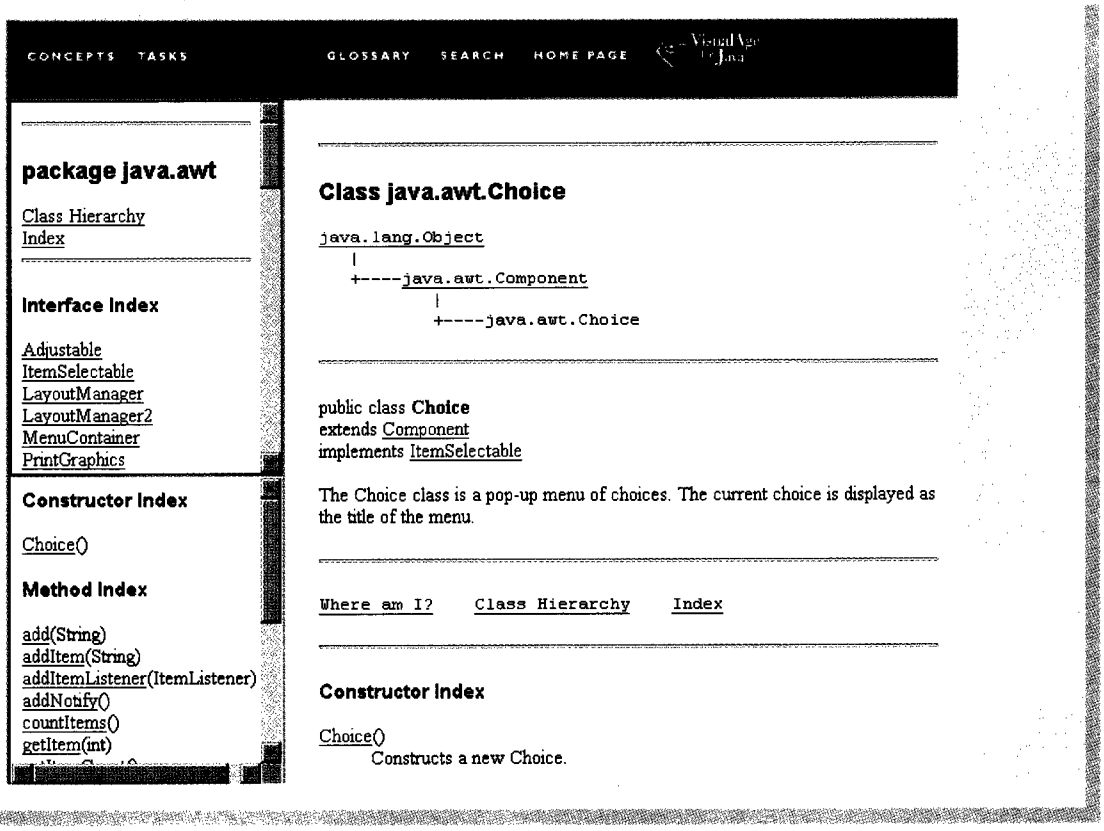
Selecting a class in the Packages frame refreshes the entire frameset. If the user explores other classes in the Class Details frame, selecting the "Where am I?" link refreshes the other two frames so that all three frames are synchronized.

Although the javadoc tool provides some context with its Class Hierarchy page, the VisualAge for Java framesets provide a more localized context that lets the programmer get to the required information more quickly without getting lost. For the user, reducing the number of mouse clicks directly translates into less context to remember. As the JDK grows and additional APIs are included with VisualAge for Java, these usability enhancements for browsing reference information will minimize the complexity for the programmer.

Conclusion

A customer had this positive remark about VisualAge for Java: "First, VisualAge for Java offered strong support for the environments we were dealing with, including NT**, AS/400, DB2, Lotus Domino**, and other technologies. Second, VisualAge for Java offered strong functionality for team development. When you have as many as four people working on the same software, it is important that you have a repository and strong versioning control.

Figure 17 API documentation frameset



VisualAge offers this over and above any other tool in the industry."

Another customer reported that he and his organization see "VisualAge for Java as an expedited route to transforming our core business applications into highly effective e-business-based services."

From both a technical and business perspective, strong market demand exists for powerful Java development tools. Software developers expect new tools to match existing tool suites in power and flexibility. Information technology managers require tools that exploit new information domains such as the Web, while at the same time leverage legacy enterprise data. IBM VisualAge for Java addresses these requirements, placing a premium on programmer productivity, ease of use, and robust code generation for middleware access.

For more information, see our Web site at http://www.software.ibm.com/ad/vajava. The "Getting Started" documentation for VisualAge for Java, Version 1.0, can be downloaded from http://www.software.ibm.com/ad/vajava/entry.htm. Also see Polan[3] for a discussion on using VisualAge for Java when working with the IBM San Francisco* frameworks.

*Trademark or registered trademark of International Business Machines Corporation.

**Trademark or registered trademark of Sun Microsystems, Inc., Microsoft Corporation, or Lotus Development Corporation.

Cited references and note

1. See http://java.sun.com/beans/docs/javaBeansTutorial-Nov97/javabeans/.
2. The marshaling process changes the language representation of abstract data into the corresponding bits and bytes, which are then "streamed" onto the network.

3. M. G. Polan, "Using the San Francisco Frameworks with VisualAge for Java," *IBM Systems Journal* **37,** No. 2, 215–225 (1998).

Accepted for publication April 15, 1998.

Luc A. Chamberland *IBM Software Solutions Division, Toronto Laboratory, 1150 Eglinton Avenue East, North York, Ontario, Canada M3C 1H7 (electronic mail: lchamber@ca.ibm.com).* Mr. Chamberland leads the technical writing teams for the VisualAge for Java and VisualAge e-business products at the IBM Toronto Laboratory. He has written several articles on Java and is the author of *FORTRAN 90: A Reference Guide*, Prentice Hall (1995).

Sharon F. Lymer *IBM Software Solutions Division, Toronto Laboratory, 1150 Eglinton Avenue East, North York, Ontario, Canada M3C 1H7 (electronic mail: lymer@ca.ibm.com).* Ms. Lymer is a professional engineer with over 13 years experience in software development. Currently, she is a solution designer on the user-centered design team at the IBM Toronto Laboratory. This team works with development organizations to optimize the usability of VisualAge for Java and other application development products.

Arthur G. Ryman *IBM Software Solutions Division, Toronto Laboratory, 1150 Eglinton Avenue East, North York, Ontario, Canada M3C 1H7 (electronic mail: ryman@ca.ibm.com).* Dr. Ryman is the solution architect for VisualAge for Java and e-business at the IBM Toronto Laboratory, where he has worked since 1982. Prior to working on Web application development tools, he worked on software engineering, image processing, and office systems products. He received a Ph.D. degree in mathematics from Oxford University in 1975. Dr. Ryman was a cofounder of the IBM Toronto Centre for Advanced Studies (CAS) and was its associate head from 1990–1994. His research interests are in software specification and design tools, and he is currently leading a collaborative university research project with Queen's University and York University in this area. He is a member of the IBM Academy of Technology, an adjunct professor of computer science at York University, and a Sun-certified Java programmer.

Reprint Order No. G321-5684.