Technical note

Using the San Francisco frameworks with VisualAge for Java

by M. G. Polan

This technical note describes the advantages of using the VisualAge™ for Java™ (VAJ) integrated development environment when working with the IBM San Francisco™ frameworks. It also discusses minimum system requirements, how to get started, and tips for using VAJ to exploit the frameworks. To fully utilize the material, the reader should be familiar with Java programming and with the basic concepts of integrated development environments.

VisualAge* for Java** (VAJ)[1] is an integrated development environment (IDE) tailored for designing and building Java programs.[2] It provides visual project and class navigation, incremental compilation, source-level debugging, visual program design, and version control. VAJ is a truly integrated development environment that maximizes productivity while easing the learning effort, and is ideal for developing large, complex programs or harnessing large frameworks such as the IBM San Francisco* frameworks (SFF),[3–5] a distributed, Java-based set of object-oriented business application frameworks.

This technical note provides some insights to assist the developer in initially using VAJ and the SFF. It does not discuss specifics related to the use of VAJ nor the operation, configuration, and use of the SFF; both topics are better left to the product manuals. The emphasis here is on discussing the advantages and the approach to using VAJ to work with the SFF.

VisualAge for Java features

The advanced features of the VAJ IDE make it ideal for use with the SFF.

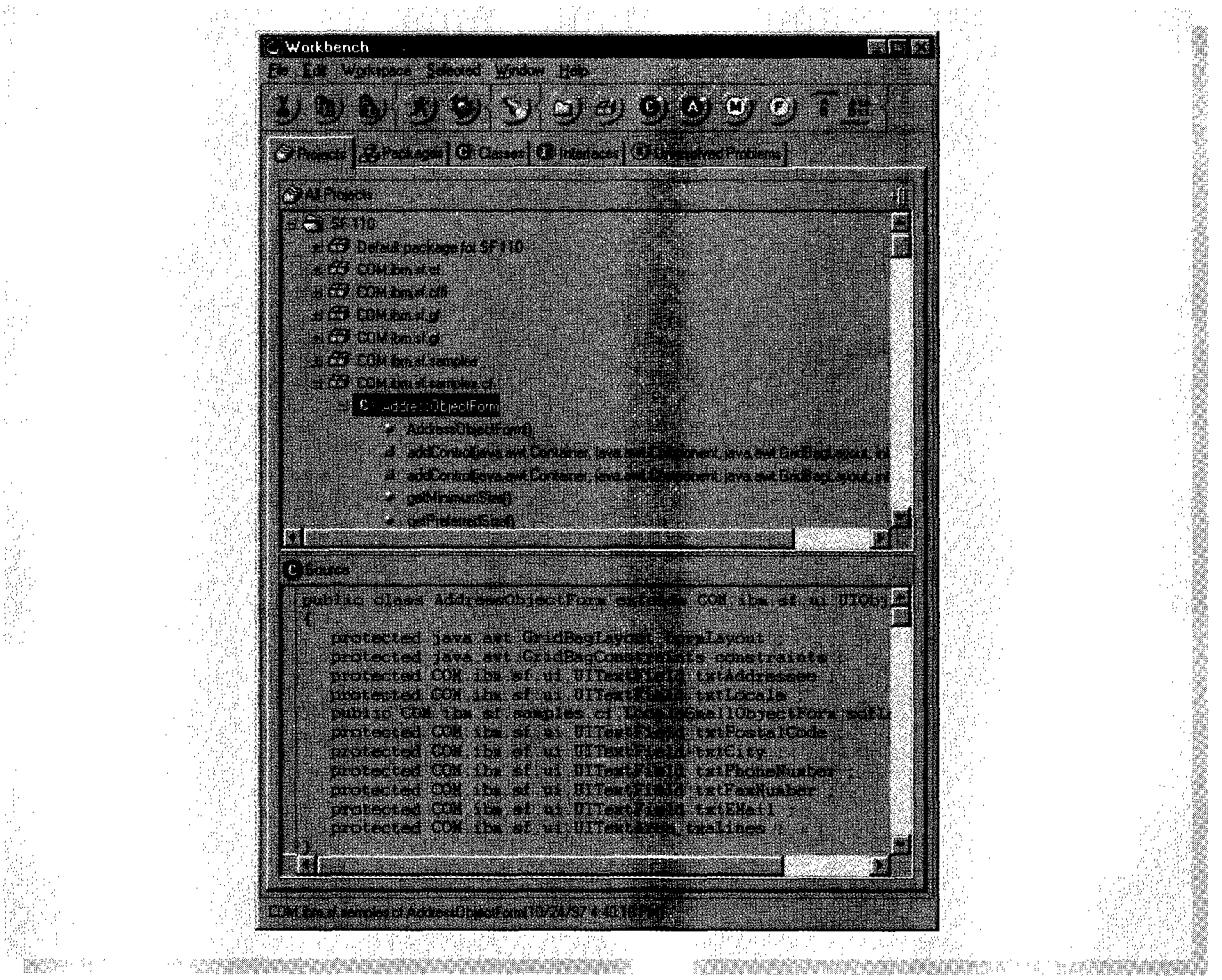
With large frameworks such as the SFF, it is important that the development environment provide facilities to navigate through the large number of classes that are provided, as well as help manage programs exploiting the frameworks as they evolve and grow.

The VAJ Workbench. The heart of the VAJ IDE is the Workbench (see Figure 1), the starting point for most development activities. The VAJ environment provides three groupings for managing program components. The first two are familiar to Java programmers—class and interface definitions that scope methods and fields, and packages that are used to collect and scope Java classes. The new and highest grouping is that of a project, which may contain any number of packages. For example, one project could be used to collect all of the SFF classes, with perhaps separate projects used to group programs exploiting the framework.

The Workbench is used to manage information in the current work space; program information in the IDE will be in the work space when in use or archived to the repository when not needed.

Workbench navigation. From the Workbench, it is possible to visually navigate through the available projects (for any user programs loaded or created in the IDE, as well as those supplied with the IDE such as the Java Development Kit [JDK] packages). Each IDE object (project, package, class, interface, or method) can be expanded (as indicated by a "+" marker) to examine the information it contains

Figure 1 The VAJ IDE Workbench showing the project and source views

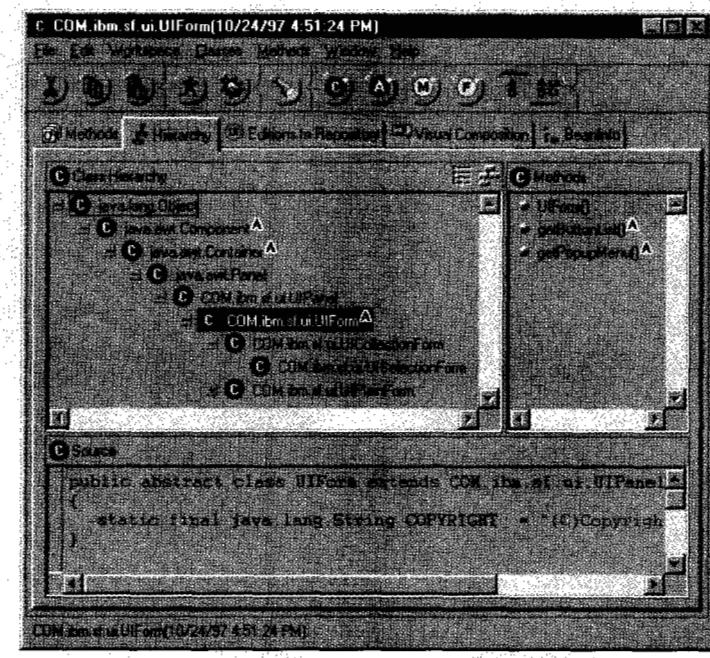


(projects include packages that contain classes or interfaces, which can contain methods and fields). Any of these IDE objects can be opened for editing. For example, a class definition can be opened by the class browser (see Figure 2): to see its place in the entire IDE class hierarchy; to examine and edit any of its methods; to examine, update, and generate its bean information; to add a new property; to add function through visual construction; etc.

A powerful set of search and browsing tools is provided to assist in finding the IDE objects to examine or edit. Since the IDE tracks all the IDE object relationships within the work space, it can immediately provide a list of objects matching a given search criterion from which the developer can select and open in any context. For example, the search facility can be used to obtain a list of all references to a Java object or a particular method. The developer could then examine or update each of those references.

Finally, in addition to its name or signature, or both, each object within the IDE is tagged with an icon to identify its type. Tagging allows one to identify at a glance packages, projects, classes, interfaces, and methods. Also visible is the scope of Java objects (public, protected, or private), as well as any modifiers—whether a Java object or method is static, synchronized, hand-coded, or generated by an application builder, etc.

Figure 2 The VAJ IDE class browser



Work space, repository, and version control. To help the developer manage changes as program development progresses, the VAJ IDE uses concepts of a separate work space and a repository to provide version control and automatic edition creation.

The current edition (the current, active copy) of an IDE object (that is, a project, package, class, or method, as opposed to the Java definition of an object) resides in the IDE work space and is directly accessible by the developer. The IDE archives information not currently needed into the repository. IDE objects can be moved to or from the repository into the work space as needed.

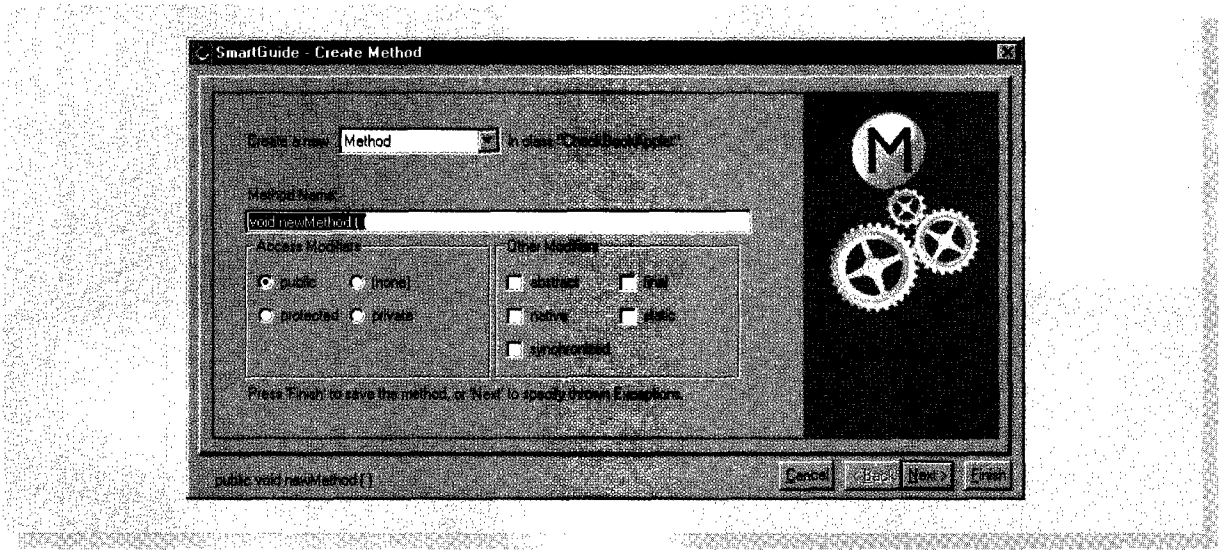
Each saved change to an object on which the IDE operates results in the creation of a new edition of that object, and a dated copy of the previous edition is automatically archived into the repository. Whenever necessary, it is possible to revert back to any of the archived (and presumably working!) editions of that object.

New IDE objects representing Java source code may be created in the work space using the IDE editor or SmartGuides (see next subsection). Source or byte code can be imported directly from files in the file system. Projects can be imported from metafiles previously exported from a VAJ IDE into the repository, and any portion of those projects can then be moved back into the work space.

At appropriate points in the development cycle, new versions of projects, packages, or classes can be created from the most current editions. The meta-data associated with those versions can be exported from the IDE. A version could then be used by other developers (perhaps working on other components of the same program) or archived for recovery purposes, or a version may simply mark a checkpoint or milestone in the development cycle.

At any time in the cycle, source code or compiled Java byte code can be exported for use within any other JDK, IDE, or Java run-time environment (JRE).

Figure 3 Creating a new method using SmartGuide



Developing, editing, and compiling programs. The IDE provides a Java-aware, context-sensitive text editor as well as a visual composition editor (VCE). The text editor provides visual feedback on language constructs and is integrated with the Java incremental compiler to highlight syntax errors. The VCE combines the drag-and-drop and WYSIWYG (what-you-see-is-what-you-get) paradigms that dramatically simplify the layout of graphical user interface (GUI) components and the interconnection of program components (especially when interconnecting JavaBeans**[6] using the java.awt delegation event model or JavaBeans events and listeners). The VCE uses the JavaBeans component model that allows any class library compliant with JavaBeans to be used. The developer may also construct JavaBeans for use with this or any other builder that is compliant with JavaBeans. Because the JavaBeans model allows for the use of any Java class, whether or not it actually implements the model, any available Java class or interface may be used in the composition of a program.

Program components built within the VCE can be directly modified with the source editor, yet can still be reloaded into the VCE for further modification and regeneration. Source changes outside the editor are preserved.

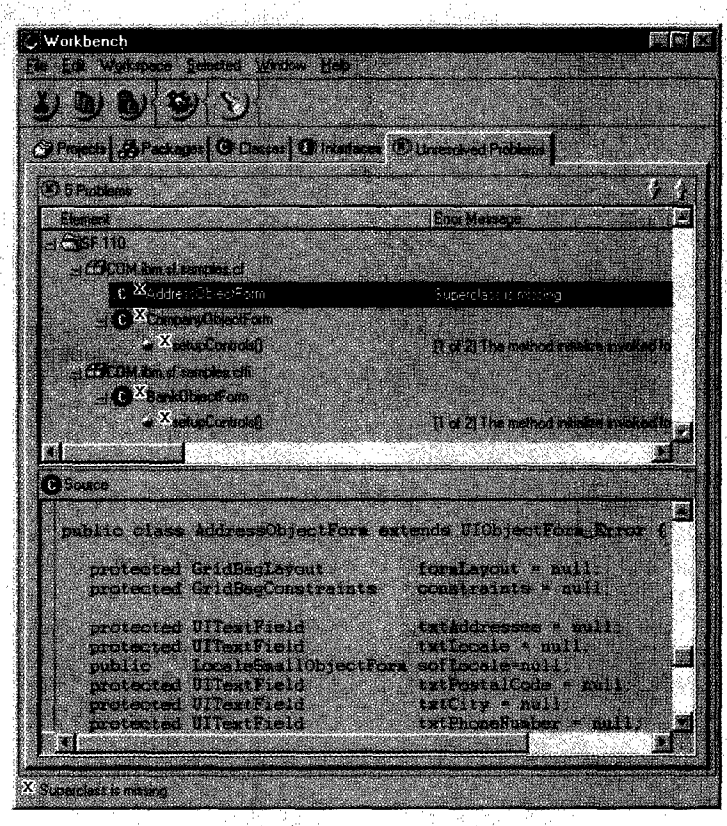
In addition, the following SmartGuides are available to help create a program:

- Project and package SmartGuides for creating new project and package structures
- Class, interface, and applet SmartGuides for creating new class or interface definitions within a package. When the new class is derived from an existing class or set of interfaces, these guides can optionally generate stubs (placeholder empty methods with the correct signature) for all required methods.
- A method SmartGuide (see Figure 3) for adding methods and constructors and for specifying method modifiers (static, final, abstract, native, synchronized) and visibility (public, protected, private)
- BeanInfo SmartGuides for easy creation of JavaBeans. The guides provide automatic addition of properties, with the appropriate get and set methods, data member, and event support if the property is bound, etc. Also generated is the bean information support class, including the customization of the information presented to the bean users.

When developing use of the VCE, a single "test" button is available that automatically generates, compiles, and then launches the program to allow an immediate test of any modifications. This feature is extremely useful when fine-tuning the appearance and behavior of the program.

The incremental compiler is invoked automatically whenever a new Java class, interface, or method is

Figure 4 Finding and fixing problems



created (including those automatically generated or created as the result of an import action) or a change is made and saved to an existing Java object. The compiler is invoked for the changed object as well as any other object within the work space affected by that change.

There is immediate visual feedback of any compile errors found in the source, including interface differences between classes, interfaces, or methods, or when required methods are missing from derived interfaces or class definitions. The compiler verifies consistency between all Java object definitions and usage within the work space. These problems can be corrected immediately or be left for later resolution. The "Unresolved Problems" tab in the Workbench (see Figure 4) will navigate directly to any problems, at which time the developer can directly edit the source. Saved corrections will be immediately reflected throughout the IDE.

The incremental compiler will provide information related to the impact of changes made to the program, or the impact of moving to a new version of an externally provided software component. When the developer deletes or redefines a method, or imports a new class, interface, or package, the compiler immediately flags any inconsistencies created by that change. Again, the Unresolved Problems view allows the developer to find and update the affected source code immediately. Should the problems related to the change prove too difficult to resolve, the developer can use the repository return to the previous edition, thus undoing the changes.

Execution and debug. Any valid Java object with a main method can be started at any time without restriction. Also, any class derived from java.applet.Applet can be launched in the provided applet viewer. The VCE provides a "test" button that will generate, compile, and launch the program under construc-

tion automatically. Finally, a scrapbook is provided that will allow execution of any code fragment within a Java object context. The scrapbook instantiates the specified Java class, then executes the indicated code; this is useful for testing new methods or source changes, etc. The IDE allows multiple execution threads to be active at any time.

A powerful source-level debugger is included within the IDE, allowing the developer to set breakpoints anywhere within the program. The debugger can also be invoked after the program is started. The "Debug" button is selected, then the thread(s) to be debugged is chosen. The debugger will halt that thread and show the current execution point. Alternatively, the source to be debugged can be found by using the class browse or method search; then the right mouse button menu can be used to set a breakpoint on the line where the program is to be halted. The program is then forced to execute a path that contains the breakpoint (perhaps by using the user interface of the program or the scrapbook). The debug window will appear automatically when the breakpoint is encountered. Breakpoints are removed in the same fashion, or the breakpoint window can be used to examine, disable, or remove breakpoints. Once a thread has been suspended in the debugger, its execution can be canceled, resumed, or single-stepped into or over methods.

Once execution is stopped at a breakpoint, the developer may use the debugger to inspect the execution stack, any local data, active Java object data members (fields), etc., using the symbolic inspection facilities.

The features of the debugger should be familiar to users of any advanced development environment. However, the VAJ debugger goes further: Once the cause of the problem is determined, the offending source code can be modified from within the debugger window and thread execution resumed. Any thread running within the IDE will immediately pick up the changes (made here or in any of the other edit windows), allowing the developer to move immediately to the next problem.

Java object field values can be examined by selecting the target from the debugger window or by highlighting the field and bringing up an inspector window (available from the right mouse button pop-up menu).

Getting up to speed. Learning a large framework such as SFF, the structure of applet and application, VAJ, or even the Java language and programming interface itself can be difficult for a new developer. Of course, VAJ provides a help facility available from anywhere within the IDE, "Getting Started" and "How Do I" support, and complete manuals for the IDE as well as for the JDK packages. The help engine is based on Hypertext Markup Language (HTML), allowing easy integration of any program documentation into the IDE (by adding bookmarks) that follow the javadoc[7] conventions that are the standard for most Java applications and libraries.

The various views provided by the IDE can quickly show the structure of packages and Java object hierarchies. The search facilities can uncover relationships among classes, interfaces, and packages.

The WYSIWYG and incremental nature of the IDE and visual composition editor ensures that there is immediate feedback when changes are made to an applet or application, shortening the learning curve considerably. Should the purpose or definition of a class or its properties be unclear, the VAJ SmartGuides can be used to create a program using that class, and the developer can then observe directly how different property settings affect the object behavior. Similarly, the scrapbook can be used to call an unclear method with various parameter settings to understand how parameter values will affect behavior.

Installing VisualAge for Java. Installation is quite straightforward; simply follow the instructions provided with the VAJ setup program. Minimum system requirements are any Pentium** processor, OS/2 Warp* Version 4, Windows 95**, or Windows NT** 4.0 or later, 32 MB (megabytes) of memory, 30 MB of paging space, and 70 MB free space for VAJ. For use with the SFF, the recommended machine configuration of a Pentium 166 MHz (megahertz) processor and 128 MB of physical memory should be considered a minimum. An additional 400 MB of free disk space is necessary above that required for the installation of the SFF (see the following section).

An HTML browser is required in order to read the on-line documentation and use the on-line help feature. A Transmission Control Protocol/Internet Protocol (TCP/IP) configuration is recommended or a stand-alone TCP/IP loopback must be configured for systems that are not connected to a network. This

configuration will be required for using both the VAJ help facilities and the San Francisco servers.

More information can be found on the Internet at http://www.software.ibm.com/ad/vajava.

Loading the San Francisco frameworks

A minimum service level for VAJ is required to successfully load and run the SFF in VAJ. Details are available in the VisualAge for Java service at http://www.software.ibm.com/ad/vajava.

Tip: At the time of this writing, it is necessary to manually update the VAJ ide.ini file to import the SFF. In the [VM Options] section of the file ide\program\ide.ini (found in the directory into which VAJ was installed), the memory limit should be changed from maximumMemoryLimit=64000000 to maximumMemoryLimit=128000000.

The SFF consist of a number of packages containing the various classes and interfaces in the framework. Importing the frameworks into VAJ can take a considerable amount of time, depending on the machine type and the amount of available memory. It is possible that a VAJ interchange file that already contains the SFF may be available by the time this technical note is published. Check the VAJ or the San Francisco Web sites (http://www.ibm.com/Java/ Sanfrancisco) for the latest information. The interchange files have a .DAT file extension.

To import an interchange file:

1. Select the menu file→import.
2. Choose "Import an interchange file."
3. Select the provided .DAT file(s).
4. Press "Finish."

Once the interchange file has been imported, move the SFF from the repository:

5. Select add project from the "All Projects" window pop-up menu.
6. Choose "from repository."
7. Select the San Francisco project(s) and the appropriate editions (consult any provided documentation).
8. Press "OK."

If the interchange files are not used, the San Francisco byte code and source code can be imported instead directly from the SFF installation. It will typically take longer than importing from an interchange file, the difference being the time it takes for VAJ to process each individual class (byte code or source) file.

To import the San Francisco packages:

9. Use the right mouse button on the project area of the Workbench "All Projects" page to select the "Add Project."
10. Add a project called "San Francisco" (or another name as preferred).
11. Select this project.
12. Select the file→import menu item.
13. Select "Class Files" and move to the next page.
14. Click the Browse button.
15. Navigate the file system to select the root directory of the SFF installation.
16. Select the COM directory.
17. Select OK.
18. Select Finish.

Tip: If using a system with limited memory, importing the packages should be done in steps for efficiency. Importing sets of packages allows VAJ to resolve dependencies on related components without the need to load and resolve the entire framework. Import the COM.ibm.sf.gf and COM.ibm.sf.util packages together, followed by the COM.ibm.sf.cf, COM.ibm.sf.cffi, COM.ibm.sf.ui, and COM.ibm.sf.gl packages.

Once the import of the SFF is complete, the VAJ workspace should be saved. To simplify re-importing the frameworks for use by any other developers on the project, or if VAJ is reinstalled, a new version of the SFF should be created. It is done by bringing up the project pop-up on the SFF project and choosing "create version." When complete, the SFF project can be exported as a VAJ interchange file. Again, because of the size of the project, this will take some time (though less time than importing the SFF byte code files into the IDE).

Once the SFF byte code has been imported, the available San Francisco source files may be optionally imported. The procedure is similar to that outlined above; however, choose "Java Files" instead of "Class Files," and select the directory of the source files you wish to import (e.g., COM\ibm\sf\samples). The source of some of the San Francisco packages is also provided in the COM\ibm\sf\source directory.

Tip: It is not advisable to import only the source files. The byte code files must be imported because var-

ious compiler-generated stub and skeleton classes are required to execute San Francisco within VAJ; source for these classes has not been provided in the SFF installation. Importing the necessary byte code files later results in a large number of compile errors discovered during the initial source import.

Building a program

Once San Francisco has been brought into the IDE, program development may begin. Use of the Visual Composition Editor is explained by the VAJ manuals; specifically, in the "Concepts" and "Tasks" sections. However, San Francisco provides a number of samples. The fastest progress will likely be made by first ensuring that the samples run, then by modifying or reusing parts of those samples to incrementally develop a program. A few tips might facilitate this effort:

- Create the applications in a separate project and package. Group related classes in packages and related applications in projects.
- Use the SmartGuides to create classes that extend the SFF. This will allow stubs for required methods to be generated automatically.
- Use the VCE rather than handcrafted code to build GUI components to leverage the WYSIWYG layout capabilities of the IDE.
- Use the VCE to interconnect objects that utilize the JavaBeans event models to ease the task and to help ensure compliance with the JavaBeans design pattern.
- Use the IDE SmartGuides to create beans that will signal events, and the BeanInfo page of the SmartGuides to customize JavaBeans (i.e., add properties, methods, and events) to ease the task and ensure compliance with the JavaBeans design pattern.
- Use the "Event to script" feature to add and invoke non-GUI function to the program (use this feature to incorporate the code copied from the SFF examples) to separate the visual from the non-visual portions of the program. This is often much easier to manage than making source modifications to the generated code.

It is not necessary to complete an entire program before beginning unit testing; indeed a feature of VAJ is to allow for incremental development, unit testing each part of the program as the implementation progresses. Java is ideal for allowing each class to be tested individually; a main or Applet.init method can be added to each class being developed, or the scrapbook can be used to test each class.

Another good practice is to lay out the GUI of the program using the Visual Builder early in the implementation phase. A number of advantages result, including:

1. Early validation of the user interface with the intended users/customers
2. Advanced prototypes with method stubs to enable early testing of program flow and performance
3. Creation of a useful unit test structure
4. Easy incremental and iterative development
5. Early separation of the view (GUI) from the internal model (behavior)

Since the San Francisco GUI applications were not constructed with the VCE, it is best to use the non-GUI sample as a base, even when constructing GUI clients using VAJ. This procedure will allow as much of the code as possible to be generated, as the San Francisco GUI applications cannot be modified within the VAJ Visual Builder.

Starting San Francisco servers

The SFF requires a run-time environment to support its function. Servers provide infrastructure such as naming, security, and factory services, and distributed business logic execution. Business logic exploiting SFF may execute within the calling client program or within the SFF servers. Working with server resident (or remote) objects requires that the SFF servers be started within the VAJ IDE run-time environment.

To start the SFF servers, select the SFF project in the Workbench window, then the "Run" button. Choose "run main( )," then selecting the appropriate class from the presented list will cause the "run" dialog for setting the startup parameters to be presented. Parameters can be saved and will be recalled when that class is run again.

The program to be tested is started in the same fashion. Each separately started class runs in its own virtual machine (VM).

Tip: For frequently started programs, create a separate class that specifies the necessary parameters as arguments, which are then passed by calling the main(String[ ] args) method of the target class. Any

Figure 5 Sample source for starting the naming server

```
/**
 * This class was generated by a SmartGuide.
 * Executable class used to start the SFF Naming Server
 */
public class StartSFGSMServer {
// location of the SFF installation in the file system
public static final String loc = "I:\SFF110";
// location of the file system based SFF datastore
public static final String datastore = "/this" + loc + "/COM/ibm/sfr/etc/Naming";
/**
 * This method was created by a SmartGuide.
 * Start the Naming Server, passing in supplied parameters
 * @param args java.lang.String[]
 */
public static void main(String args[]) {
        // define the system properties needed by the server at startup
        java.util.Properties sp = System.getProperties( );
        sp.put("datastore", datastore );
        sp.put("serverName","SFGSMServer");
        try {
                COM.ibm.sfr.gf.SrvServerImpl.main(args);
        }
        catch (Throwable t) {
                System.out.println("StartSFGSMServer failed" + t.getMessage( ) );
                t.printStackTrace( );
        }
    }
}
```

additional startup handling can also be included using this technique. The new class can then be run using the IDE run button rather than respecifying the parameters each time—particularly useful when continually starting processes using the same class but with different parameters, such as the SFF servers.

For example, to start the naming server see the sample in Figure 5.

Execution environment

The VAJ IDE, the Sun JVM, and the San Francisco servers are relatively large programs for today's systems. They tend to use a lot of memory and CPU resources, so a little planning can help the response time of the applications within VAJ.

First, the servers should be run either within the IDE or on a different system. They must be run from within the IDE if it is necessary to debug code running within the server.

Attempting to run the servers using a Java virtual machine (JVM) other than the VAJ IDE is possible by using the startup tools provided by the SFF installation. It is likely, however, that a stand-alone system will spend much time swapping between the IDE and the server JVM processes, unless a large amount of real memory (256M or more) is available. Memory requirements are reduced when the servers run within VAJ. Running the servers on a different system reduces system requirements even further.

When running applets within the VAJ IDE, they must be permitted to access the project text resources of the San Francisco classes resident in the file system. Use the applet menu to set the applet security level to "unrestricted."

Deploying a program

It must be remembered that the VAJ IDE is a development environment; it is not intended (in this release) as a production environment. However, since applications are developed using Java, they will deploy with little or no effort onto any compliant Java VM (currently at JDK 1.1), available today for most operating systems.

The program source can be exported from the VAJ IDE for deployment in a number of different ways:

- As .java files containing Java source for a separate build outside of the IDE
- As .class files containing compile Java byte code into a user's class path for immediate execution
- As published projects so that the project resources will be included in the exported materials
- As a JAR (Java ARchive) file[8] containing all of a user's classes for easy download into a browser client for use as applets
- As interchange files for backup and recovery or exchange with other team members or between machines

There will be a considerable increase in program performance; the advantages of the VAJ IDE come with a performance penalty (though that penalty will be considerably reduced in the future). If, during development, performance becomes a concern, the developer should remember to measure performance after exporting and testing outside of the VAJ IDE before restructuring the program.

Concluding remarks

VAJ is a powerful new IDE that considerably reduces the learning curve and increases developer productivity. Its advanced features, listed below, make it ideally suited for managing large application frameworks such as the SFF.

- Powerful project, package, class, and method hierarchical navigation. All IDE components are objects with specific associated actions. This approach is very helpful in managing large frameworks and projects.
- Source-level debugging with update in place (no need to navigate through large amounts of source) and instant effect (programs need not be restarted).
- Workspace-based incremental compilation—impact of source changes on dependent classes and interfaces (usage or derivation) is seen immediately.
- Syntax-sensitive text editor and incremental compilation. They allow source code to be checked for clean compilation before being saved in the IDE work space.
- Repository-based, automated version control with automatic backup of modified code for configuration management.
- Powerful search facilities to find the definition or locations of reference of all IDE objects, including classes, interfaces, methods, and fields. Searches are completed quickly by the IDE by using the work space class meta information.
- Class creation SmartGuides. They allow all required method stubs to be generated automatically. This greatly simplifies the task of extending application frameworks, by generating the stubs necessary for implementing the concrete instance of an abstract class or interface (in any combination).

Acknowledgments

My thanks to Dan Geort and Randy Baxter who were fundamental in helping me understand SFF and have it run on my system, and to Marcio Marchini, who with a few simple tips increased my VAJ productivity one hundredfold.

Cited references and notes

1. For information related to VAJ, see the URL on the Internet at http://www.software.ibm.com/ad/vajava.
2. For information on the Java environment, languages, and products, see http://www.javasoft.com.
3. For information on the SFF, see http://www.ibm.com/Java/Sanfrancisco.
4. A. Thomas, "San Francisco: IBM's Business Object Framework," Patricia Seybold Group's Distributed Computing Monitor (September 1997).
5. D. Andrews and M. A. DeGiglio, *IBM's San Francisco Project: Java Building Blocks for Business Application Developers*, White Paper: Progress Report, D. H. Andrews Group (July 1997).
6. JavaBeans is the standard component model defined by the JavaSoft Corporation to allow the interconnection of Java components from different software vendors. Java classes from any vendor that comply with the JavaBeans specification can be used directly in any development environment from other vendors that is compliant with JavaBeans.
7. A tool provided with the JDK used to generate HTML-style documentation directly from a Java source file.

8. For more about JAR files, see http://www.javasoft.com: 81/docs...orial/post1.0/whatsnew/jar.html.

Accepted for publication January 12, 1998.

Michael G. Polan *IBM Canada Ltd. Laboratory, 1 Park Center, 895 Don Mills Road, Don Mills, Ontario, Canada M3C 1W3 (electronic mail: polan@ca.ibm.com).* Mr. Polan is a senior document analyst at the IBM Canada Laboratory and was the team lead for the VisualAge for Java and the VisualAge for C++ Data Access Builder component.

Reprint Order No. G321-5674.

[[Page 226 is blank]]