Lotus eSuite

by B. Briggs

The Java™ programming language presents many new opportunities for application developers and users. Lotus eSuite™ uses Java to provide an environment for the deployment and execution of business applications from the corporate intranet. Tools such as spreadsheet, calendar, chart, e-mail, and database access, among others, bring the benefits of network computing to the end user. This paper discusses the eSuite functions and the principles and goals that guided their development.

The Java** programming language presents many new opportunities for application developers and users. Its reach, nearly ubiquitous given now-widespread support for Java Virtual Machines and the truly global connectivity provided by the Internet and Java-enabled browsers, permits lightweight pieces of software to be efficiently downloaded and executed anywhere in the world.

In the summer of 1996, engineers at Lotus Development Corporation began to design and create a new class of application software designed to take advantage of Java. This product line, initially code-named Kona, and now officially named eSuite**, was announced formally in November 1997 and is now generally available.

This paper presents a technical overview of eSuite. It begins with a discussion of the principles and goals motivating the creation of eSuite. Next an overview of the architecture underlying each of the eSuite components is presented, including an exposition of the user interface frameworks and of the InfoBus, a key Lotus innovation enabling easy data transfer between software components. A brief overview of the eSuite component applets is then presented in order to illustrate these technologies in use. An example showing the components working together in a customized application concludes the paper.

eSuite goal

The overarching goal of the eSuite effort was to provide the tools by which the corporate intranet, and to some extent the broader Internet, could be transformed into an environment for the deployment and execution of business applications. With such tools as spreadsheet, calendar, chart, e-mail, database access, and others, we intend that the Internet—and more specifically, the World Wide Web—evolve from a medium hosting primarily statically published documents to one featuring much more interactivity. In a sense the goal was to bring the benefits of network computing to the end user.

However, to anyone attempting to create software for the Internet, it quickly becomes apparent that this "network-centric" environment differs in many important ways from the traditional "desktop-centric" model of personal computing. We have observed that simply porting existing desktop applications to Java without awareness of these differences results in failure. Therefore a discussion of the assumptions and conclusions that guided our development is in order.

The network-centric environment

In the initial planning phases of the project, the eSuite development team studied the nature of a network-centric computing environment and, specifically, its implications for end-user business productivity applications.

0018-8670/98/$5.00 © 1998 IBM

Our initial observation concerned the locality of application logic. In the present desktop computing paradigm, applications are typically loaded from a local hard disk (or a remote disk simulated to appear local), to which persistent application data are also saved. Each workstation must thus have its own physical copy of the applications, separately installed and maintained.

A number of studies have indicated that deployment and maintenance of such applications, particularly in large organizations, comes at a high cost, ranging from $8000 to $16 000 per year, per workstation. This concern has motivated the notion of relatively inexpensive, stateless devices (i.e., not possessing local persistent storage) called "network computers."

To achieve maximum economies, and to construct devices specific to particular applications, network computer hardware vendors are employing a wide variety of CPUs and system architectures. Java's platform independence, i.e., the ability of Java applications to run on any machine featuring a Java Virtual Machine (JVM), makes it the tool of choice for software on such devices.

Perhaps more importantly for our discussion, network computers, as well as Web browser programs, signal a shift away from workstation-resident applications and toward downloadable ones. Such applications reside on a server platform—not the workstation—and are downloaded on demand. Since there is then only one persistent copy of the application, information technology professionals charged with maintenance and upgrades find their tasks simplified.

We further observed a reluctance of some organizations to upgrade end users from so-called nonprogrammable terminals (NPTs, of which IBM 3720- and 5250-class terminals are the premier examples) to graphical personal computers and workstations. Part of the reason for this reluctance, as mentioned earlier, stems from the high cost of deployment and maintenance of applications on personal computers. However, it is also the case that while such users covet the ease of use of graphical user interfaces in principle, in practice modern personal computer applications have grown enormously complex, and therefore intimidating. The *Wall Street Journal*[1] counted approximately 4500 separate commands in Microsoft's Office 97** suite of applications, and Lotus SmartSuite** is in approximately the same category.

We also noted the extraordinary growth of the Internet, and of the World Wide Web in particular. Early in its history the Hypertext Transfer Protocol (HTTP) demonstrated its ability to serve as a backbone for application deployment. HTTP enabled transmission of richly formatted data to client workstations and enabled user input, albeit via very primitive mechanisms. Indeed, many enterprises now rely upon so-called internal "intranets" for mission-critical applications; yet the richness of these applications is today quite limited.

Our components are 100 percent Java in order to maximize the "cost of ownership" benefit to administrators and end users.

Finally, we wanted to provide software components that could be assembled into larger applications customized to a particular task. Seamless incorporation of components into more inclusive, tailored applications permits users to focus on the task, rather than the software tool (a portfolio manager rather than a spreadsheet, for example), thus achieving increases in productivity.

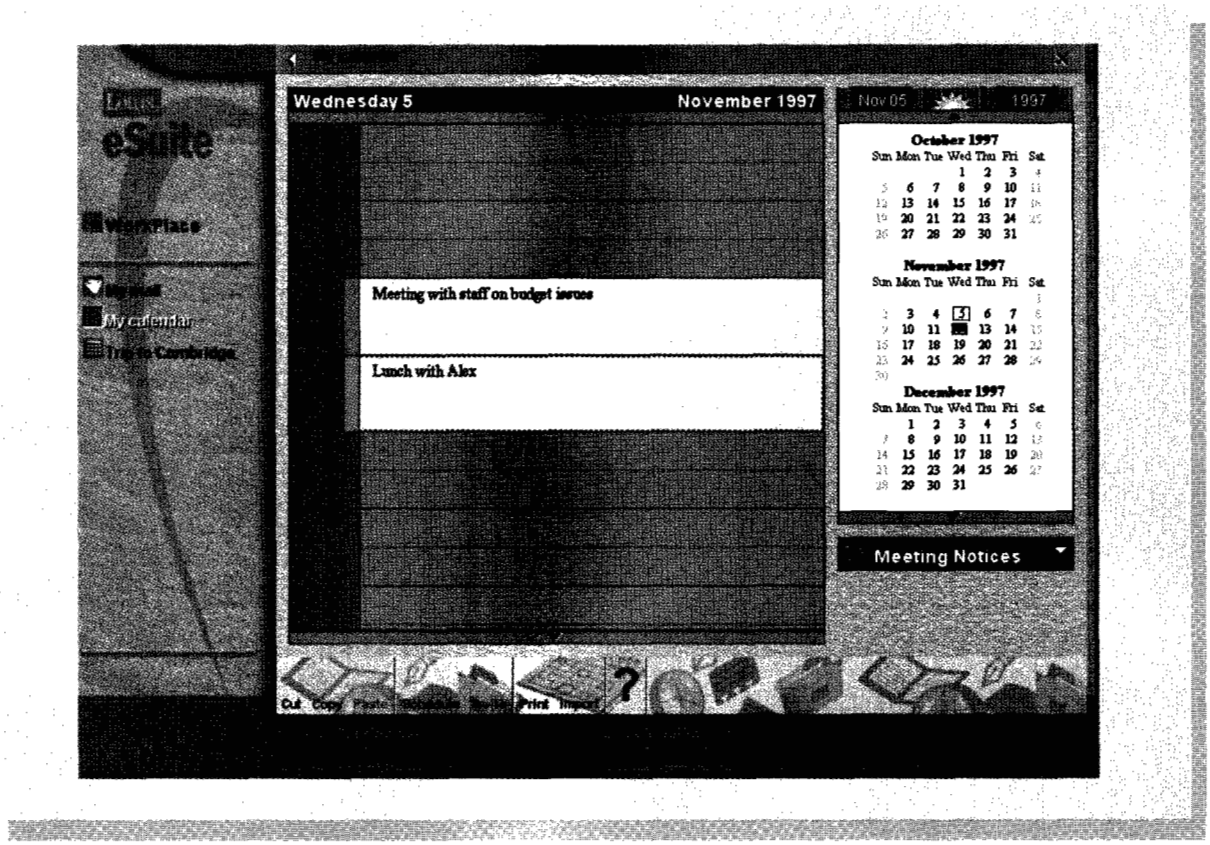
We recognized that these goals were in some cases quite complementary. For example, we wanted to build components so that designers could create highly customized applications; by restricting the functionality of the component to a highly focused set of features we also achieved the goal of compactness and thus efficient delivery over a network.

Design principles and goals

Based upon these observations, we adopted a series of design goals, which are summarized below.

Goal Number 1: 100 percent Java. Our components consist of 100 percent Java in order to maximize the "cost of ownership" benefit to the administrator and the end user. This goal allows the eSuite components to run on any hardware platform featuring a JVM and the requisite class libraries, without regard to processor type or operating system.

Figure 1 eSuite workplace



Goal Number 2: Broad context. Further, we wanted our components to "live" in as wide a variety of container applications as possible. A container application is one that provides application context to a component: layout location, focus, command services, and so forth. To accommodate so-called Java-builder programs, in which the container is a Java application, the eSuite components follow the guidelines in the JavaBeans** specification.[2] Another important container for the eSuite components was, and is, the all-Java network computer environment and, in particular, the workplace developed by Lotus (see Figure 1).

Somewhat orthogonally, since we wanted our components to run in the context of a World Wide Web page, that is, to be contained within HTML (HyperText Markup Language), they are applets as well. Hence our components are *both* JavaBeans and applets; for the remainder of this paper I will refer to them as applets.

Goal Number 3: Limited size. Each applet is intended to be less than 500 kilobytes of compiled Java bytecodes. The intention is to limit to about 1 megabyte, the size—and thus the load time—of the downloaded image, which will normally consist of the bytecodes of the applet, required resources (images, locale-specific strings), and data. With compression as implemented in both JAR (Java archive) and Microsoft CAB (cabinet archive) files, this brings the total download size to approximately 400 kilobytes, which empirically requires only a few seconds on a typical high-speed (10 megabytes per second) local area network. (Admittedly this is a "back-of-the-envelope" calculation—and, given the number of variables in the equation, it probably cannot be made more precise. However, it has empirically proven to

be accurate enough to use as a base for design decisions.)

Goal Number 4: Specific functionality. Each component's functionality is intentionally designed to be focused and specific, rather than all-encompassing. The spreadsheet, for example, consists only of a recalculating grid; the chart, normally included in a full spreadsheet application, is in fact a separate component. Given that an application developer might want to link spreadsheet, or relational database, or other arbitrary data to the chart or to another component, we recognized that a simple data transfer mechanism was required, and this led to the invention of the InfoBus, described later.

Further, the level of functionality was deliberately limited, the intention being to provide only the most commonly used functions within a component. For example, the word processor supports multiple fonts, colors, tables, and images, but not cross references or footnotes.

Goal Number 5: Usability. We wanted the components to be easy to use from a command and control standpoint. Usability tests have shown certain limitations with the overlapping-window user interface model (naive users forget about obscured windows, for example). Further, in some environments, such as browsers, there is very little opportunity for user interface "merging" as is possible in other component architectures (for instance, Java applets hosted by browsers generally cannot integrate with the browser's interface, e.g., its main menu). Finally, we wanted a very high degree of customization capability for the user interface: for example, to allow the user interface to be completely removed in a prebuilt application, as described later. These factors motivated creation of the InfoCenter, also described later.

Goal Number 6: Standardization. Wherever relevant, our components utilize protocols and formats standardized by the relevant groups. This decision had a number of useful implications: our client-side components can run against any number of servers, since they use standard protocols; and since most of the applets use HTML as a standard file format, their persistent data can be viewed, if in read-only mode, by *any* browser regardless of whether it has access to the component that created it.

Goal Number 7: Multiple usage models. Finally, we realized that our components would be utilized in a rather wide spectrum of usage models. In some environments, primarily those in which *ad hoc* content creation is performed, users expect applications to be launched and utilized in what might be termed a "traditional" model—that is, the user runs a particular application (say a spreadsheet, or a word processor), creates data, saves the data, and then exits the application. However, we also recognized the growing importance of more task-oriented environments. In these, software *components* are linked together to construct a much more customized application, such as a patient billing form, or portfolio management application, or database reporting tool. Clearly we had to accommodate both models.

The level of functionality was deliberately limited, providing only the most commonly used functions within a component.

User interface

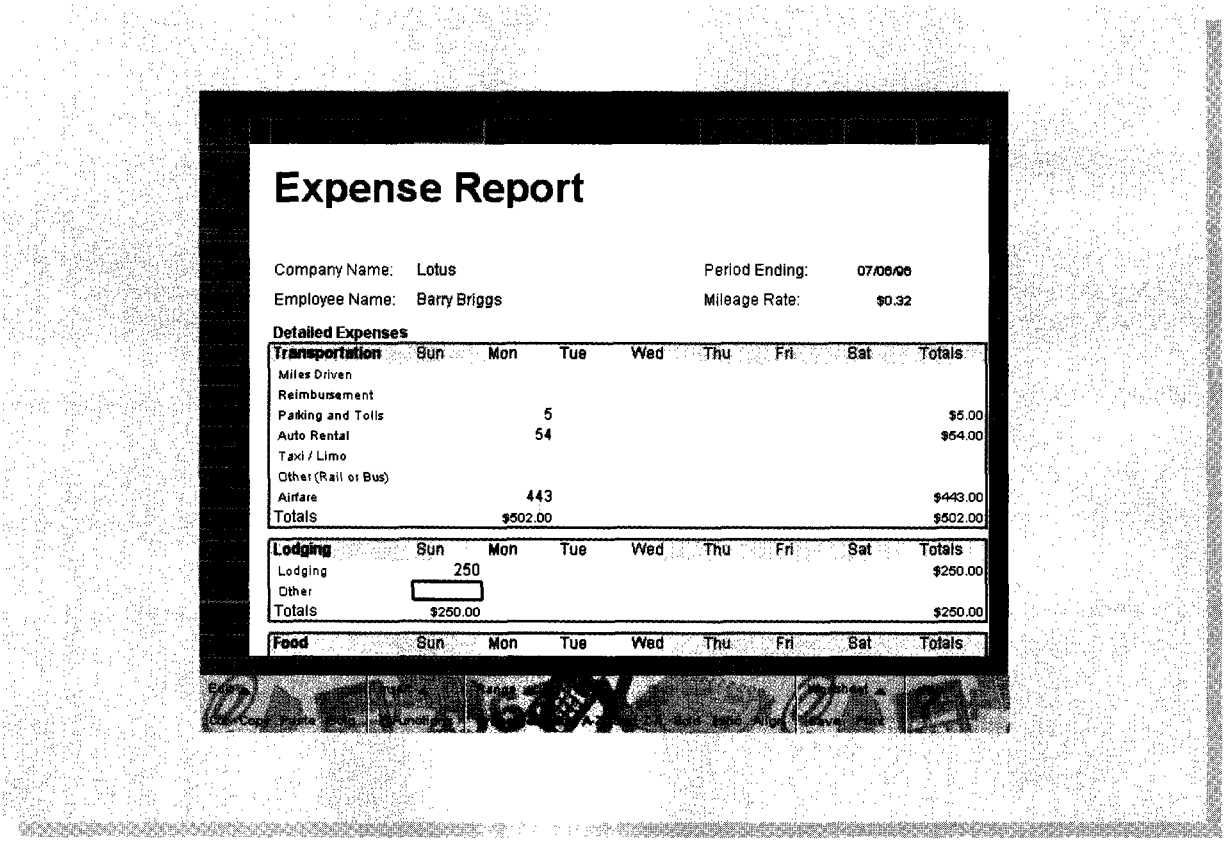
The design of the user interface focused on simplicity and ease of use. One of the key tenets of the design, therefore, was to eliminate as much as possible the use of so-called "three-dimensionality," that is, data obscuring other data because of such constructs as overlapping windows. In fact the eSuite user interface is specifically not a windowed one, the goal being to focus the user on the task rather than on the interface.

This decision has implications for the applets' functionality as well. For example, the spreadsheet applet supports only one sheet, not a notebook of sheets, as does its vastly more powerful SmartSuite counterpart Lotus 1-2-3**.

Figure 2 shows a basic eSuite spreadsheet. The spreadsheet, like all eSuite components, supports the graphical manipulation expected of such an applet: in this case, the ability to drag and resize a column or row, to select a (single) range of cells, and so on.

At the bottom of the spreadsheet is the menu bar, a command and control area we call the InfoCenter.

Figure 2 eSuite spreadsheet and InfoCenter



The InfoCenter was perhaps the most heavily tested subsystem in our usability laboratories, and user input resulted in numerous improvements. For example, our observation that users' eyes tend to gravitate toward the top left motivated placement of the InfoCenter at the bottom—thus focusing the user on the task rather than on the interface.

InfoCenter structure

Three primary constructs make up the InfoCenter: the action bar, panels, and QuickPicks.

The primary user interface component implemented by the InfoCenter is called the "action bar"; on it is presented a set of menu choices that are context sensitive, i.e., supplied by the applet. To reduce the number of gestures required by the user, the most commonly used submenu items—for example, cut, copy, and paste under the edit selection—are promoted and reside directly below the top-level menu selections. (This effectively replaces the function of a tool bar—single-click access to a function—in most present-day applications for Microsoft Windows**.) Clicking on "Edit" will in fact bring up a menu that includes cut, copy, and paste, as well as other less-frequently used functions.

Selecting "Properties" (see Figure 3) shows a panel enabling the user to modelessly change the characteristics of the given selection, such as its font, color, point size, and so on. By "modelessly" we mean that the changes take place immediately; the user need not bring up the panel, use it, and then dismiss it for the changes to occur.

Further, the properties shown are relevant to the current selection. As the selection changes, the applet

Figure 3 InfoCenter property panel

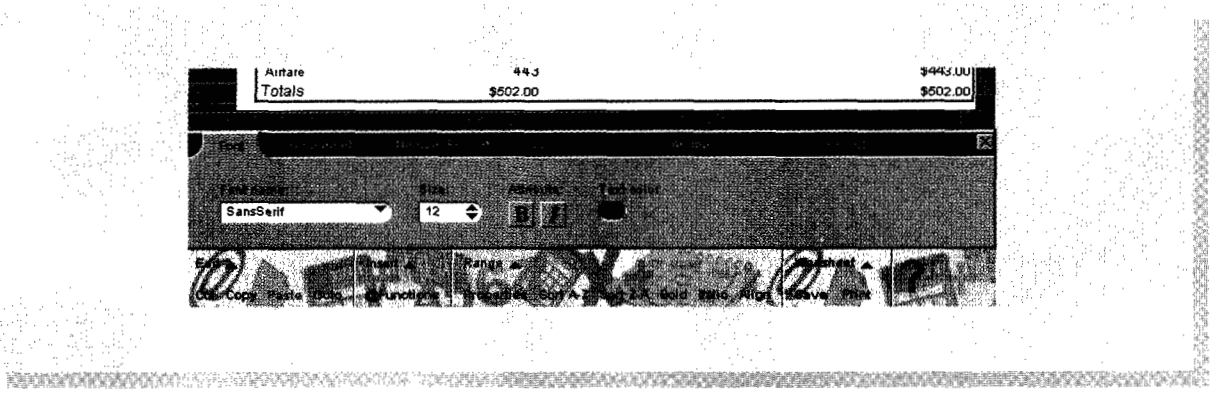
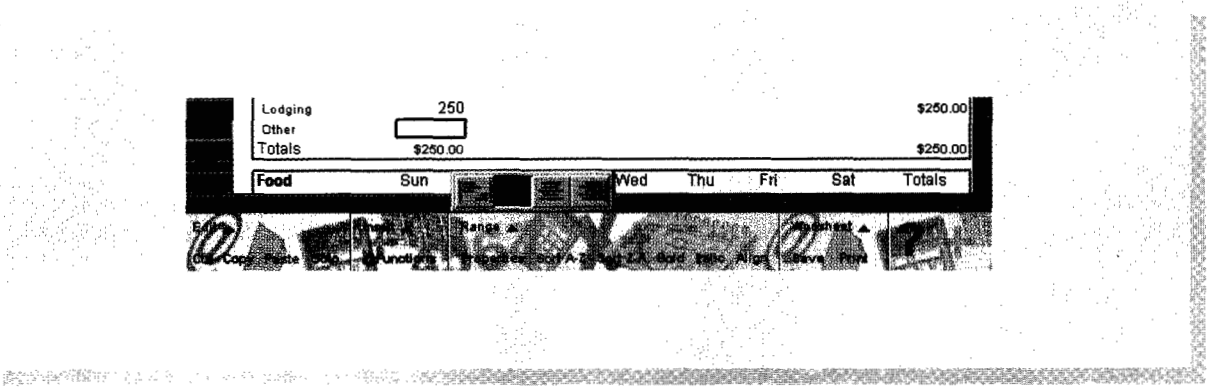


Figure 4 InfoCenter QuickPick



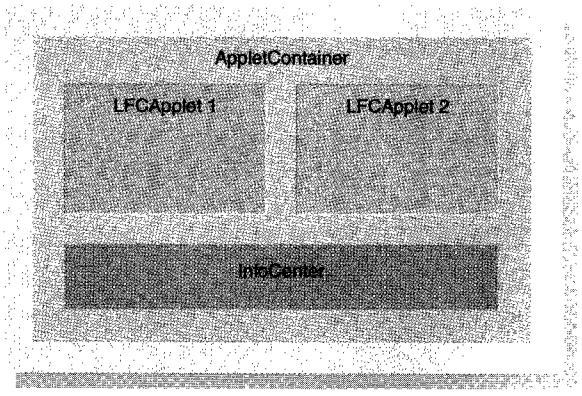
notifies the InfoCenter, and individual property panels are added or removed as appropriate. For example, if only text is selected, then the "Number Format" panel is not relevant and therefore not displayed. The internal architecture of this mechanism is described later.

As a further shortcut, a user interface tool called a "QuickPick," which is a simple pop-up, can be utilized as a shortcut. A QuickPick requires a minimum of mouse gestures from the user in order to accomplish some action. In Figure 4 the alignment tool is depicted. The user clicks on the "Align" menu item, which causes the tool to be displayed. The user then selects the desired alignment, and the effect is immediate.

Frameworks

To realize the user interface, and to accomplish the desired goals, we created a set of frameworks in which the applets execute. These frameworks provide the following common services: user interface command and control (the InfoCenter), rendering enhancements, printing, persistence, internationalization, resource management, and others. Figure 5 depicts a basic schematic of the three key components of the frameworks: the AppletContainer, the LFCApplet, and the InfoCenter. (In developing eSuite components and applications, it should be noted that the AppletContainer and InfoCenter are both optional, thus allowing maximum control by the developer of the interface presented to the user.)

Figure 5 Frameworks schematic



An instance of the AppletContainer can physically contain one or more LFCApplet instances; the intent is that several components can be linked to create the impression of a single application. Passed to the applet container are HTML parameters (described more fully later) that dictate the layout of the individual applets—left-to-right, top-to-bottom, and so on. (Internally, the AppletContainer employs the Java AWT (Abstract Window Toolkit) GridBagLayout class to implement this feature.) Optionally the AppletContainer can paint a background image across all the contained applets, thus again providing a more unified, and esthetically pleasing, appearance. The AppletContainer instance also creates the InfoBus instance used to connect each contained applet to the InfoCenter instance; the architecture of the InfoBus and its use in the InfoCenter will be described later.

All eSuite applets are instances of the Lotus base class LFCApplet, which is derived from the JDK** (Java Development Kit) Applet class. LFCApplet and its supporting classes provide a wide variety of services with scope beyond that of this paper; briefly, they consist of the following.

- *Internationalization functions and resource management.* The LFCApplet base class provides function that allows an instance to determine the locale in which it is running. Based on this information it loads the appropriate "resource bundle," a file containing localized strings, images, and other information.

- *Persistence framework.* The JDK[2] provides two mechanisms for object persistence. The first, a Java interface called *serializable*, automatically bundles up all of an object's public and private data in an opaque format; when the object is reinstanced at a later time the data are reinitialized. The serializable mechanism has some advantages, chief of which is its ease of implementation—virtually no programming is required. The other mechanism is a Java interface called *externalizable*. The externalizable interface defines a number of methods that must be implemented by the class. Its chief advantage is that the format, type, and amount of the data to be saved or retrieved is entirely under the control of the class.

  However, the externalizable mechanisms, while closer to what we desired, were not deemed to be flexible enough; in particular, they do not allow the implementing class to distinguish between different file types, and this capability is important for the eSuite applets. For example, we wanted the spreadsheet to be able to read and write data in HTML as its default format, but also in Lotus 1-2-3 worksheet format (WK1), given its widespread use in the industry. Hence, we invented a new persistence interface. The *persistable* interface permits clients to advertise supported file types, and to read and write documents of a selected type.

  Finally, to support the particular requirements of network computers (NCs) we created a *swappable* interface. At this writing NCs typically have no disk cache or local swap file; current NC operating systems do not provide a means to gracefully handle out-of-memory situations. In addition, we did not want the user to have to save all data when logging off; we wanted the software to save its state and restore it when the user logged on subsequently. The swappable interface consists of two methods: saveState and restoreState. The caller provides a data stream from which to read or write.

- *Convenience functions.* Such functions simplify existing functionality or ensure that applet usage of a given function is consistent. Access to various cursor shapes, which in the case of running as an applet on a Web page requires recursively traversing the containment hierarchy until a Frame object can be retrieved, is an example of function simplification; management of thread groups is an example of consistency.

- *Context-sensitive help.* eSuite uses HTML as its help data format.

Control flow between applets and the InfoCenter

An LFCApplet base class method, assertSelection (Selection s), provides a means by which applets coordinate their functions with the InfoCenter. Each time a user selection changes—for example, the user paints a new range in the spreadsheet—assertSelection( ) is called. Passed to it is an object s that implements the interface Selection. (A Java interface is like a class, but with only declarations of its methods.)

From methods in the selection interface, the InfoCenter retrieves objects describing what commands, panels, and QuickPicks can be enabled and how to process them. These commands themselves are kept in "action descriptors," which are text strings kept in the resource file to facilitate translation. The action descriptor also names a class to be instantiated when the command is actually invoked. For example, here is the action descriptor for the "Cut" command:

```
IDA_CUT=direct, lotus.fc.ic.ClipboardCommander,
  ID_CUT, IC_AD_STR_CUT, IC_AD_IMG_CUT,
  null, ICL_AD_STR_CUT
```

Here the keyword *direct* specifies that executing this command operates directly upon the applet (as opposed to invoking additional user-interface functions), and that the class that will actually be instantiated to handle the command is lotus.fc.ic. ClipboardCommander. (The remainder of the arguments symbolically identify the command string—"Cut"—so that it can be easily translated, the command identifier, and other parameters not relevant for this discussion.)

The InfoBus

The InfoBus component was designed to provide a lightweight, yet powerful, means of data transfer between other components. As the name suggests, its implementation is reminiscent of a hardware bus, in that data are placed on the bus by one component and are immediately available to all attached components. Applets implementing InfoBus functionality can easily and without programming be linked together; Figure 6, a screen shot of a functioning application, depicts the eSuite spreadsheet and chart applets. The chart of course graphically reflects the numerical data in the spreadsheet; it is important to note that the chart contains no spreadsheet-specific code.

The basic philosophy behind the InfoBus distinguishes applets as being data *producers*, data *consumers*, or both. Applets that produce data publish the data to the bus in a canonical format; consumers can then retrieve the data using any of a number of access methods. This common encoding is defined by the DataItem class. When one component has data to share with one or more other components, it publishes a DataItem instance.

Two fundamental mechanisms must exist for successful data exchange between two (or more) components. First, the components must be able to *rendezvous* on the data. The InfoBus uses events and named data items to accomplish this. When a data consumer wishes to receive InfoBus data, it adds an InfoBusDataConsumer listener, which is a Java event handler. Then the consumer waits for a named data item. Conversely, a data producer creates a data item and fires an event with the name as an argument (the complete specification of the InfoBus is available from the Web[3]). The origin and use of these names is entirely the responsibility of the applet.

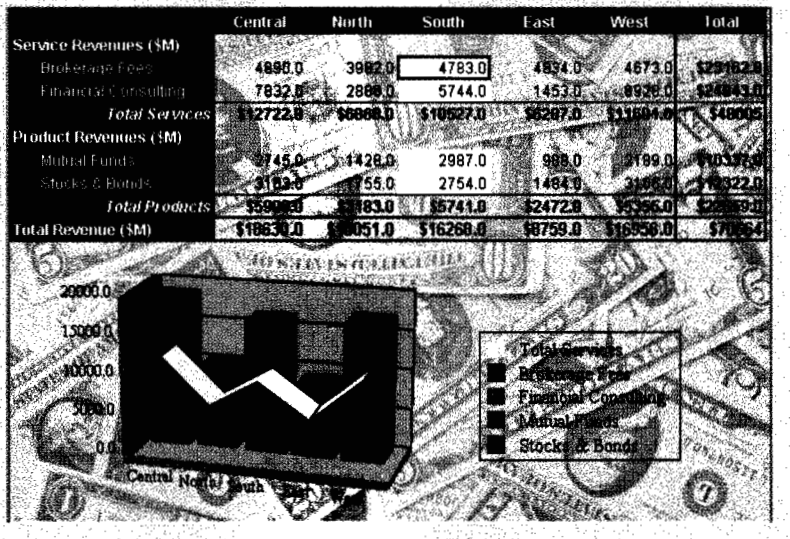
Consider the example of the spreadsheet and chart shown in Figure 6. In spreadsheets, a traditional and commonly used naming scheme is that of the named range. This implementation of the spreadsheet publishes the names of its named ranges on the bus; one of them is called "Sales." The chart component has been programmed, in this case via a parameter passed through HTML, to render the information in the "Sales" named range. In short, the spreadsheet publishes the existence of a data item named "Sales"; the chart subscribes to it. Changes made to the underlying data in the spreadsheet result in an instance of the DataChangedEvent being fired, and the chart is updated.

The second requirement for successful data exchange is a set of one or more agreed-upon access mechanisms. The InfoBus specifies a broad range of these, expressed as Java interfaces, not all of which must be implemented by every InfoBus data producer. (A data consumer can use Java's instanceof operator to query the available access methods on the data item.)

The InfoBus specification describes the following access mechanisms:

- Immediate Access. This is the simplest of the access mechanisms. A consumer can retrieve the con-

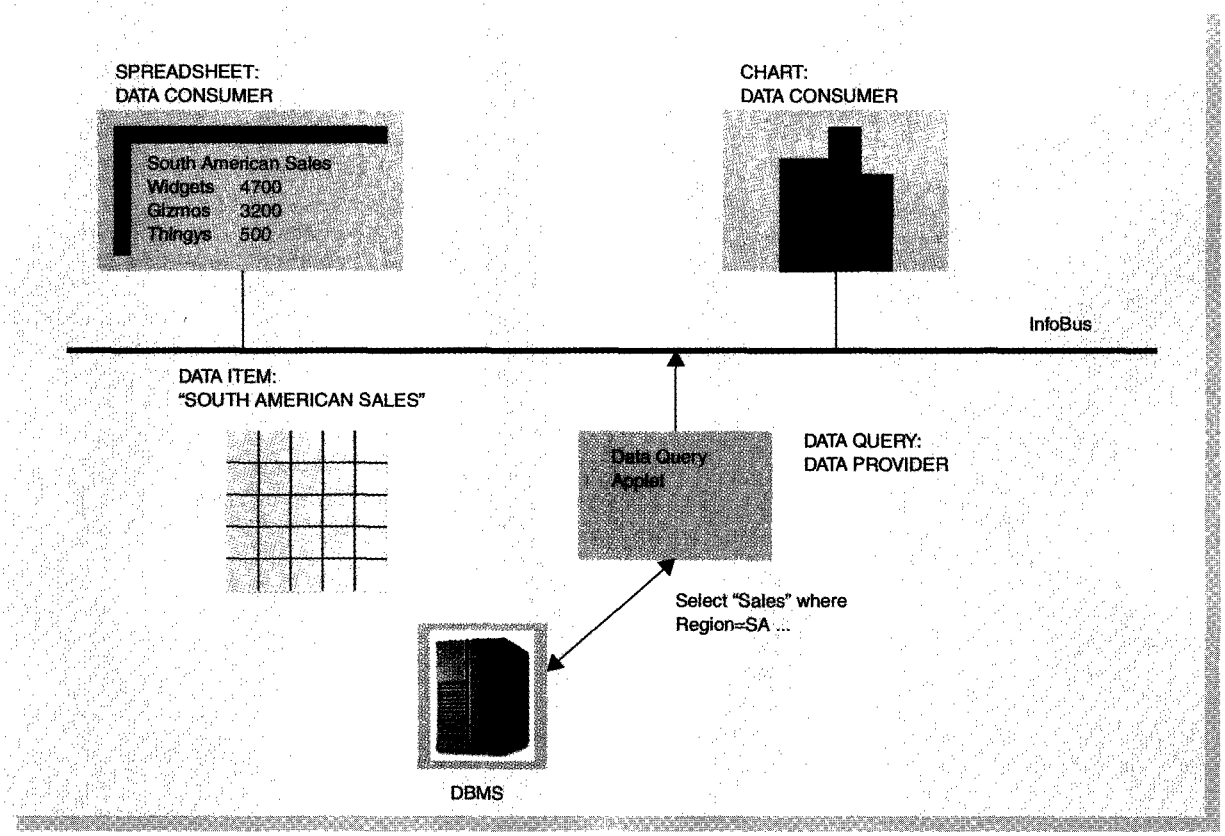
Figure 6 Spreadsheet and chart



tents of the data item either as a Java string or as a Java object. For reinterpretation of the Java object, the data item may return a Transferable instance via an optional method. This exposes the object in its various "data flavors" (the JavaBeans equivalent to clipboard formats[4]).

- Collection Access. The Collection Access interface is the base interface of the other access mechanisms. A getCursor( ) method returns a DataCursor instance, allowing the consumer to traverse or enumerate the data within a Collection instance.
- Array Access. Using the Array Access interface, the consumer can query the data for their dimensionality. (The array can be n-dimensional.) Having determined the bounds, the consumer can subdivide the data further into additional arrays. An example usage might be for a chart component to subdivide a single data item into several, each representing a different renderable data series. Each subdivided data item can have its own data cursor.
- Keyed Access. This interface allows a consumer to retrieve a data value by a key.
- Rowset Access. The Rowset Access mechanism is generally for use by InfoBus data producers to publish data retrieved from relational databases. The Rowset Access interface allows consumers to determine meta-data, that is, the number of columns and their names and data types, as well as to handle insertion and deletion.
- Database Access. The DbAccess interface provides a mechanism by which a data consumer can closely control retrievals from and updates to a relational database through the InfoBus. Using DbAccess, a consumer can pass a Structured Query Language (SQL) command to a DBMS (database management system) InfoBus data producer. The result set is normally returned as a Rowset Access data item,

Figure 7 InfoBus sample operation



and it can optionally be made generally available to all consumers on the InfoBus.

For the present, the InfoBus is strictly a single-machine (or more specifically, single-JVM) architecture; in its current form it is not distributable. To make distributed data (such as those from a relational database on a server) available to InfoBus members requires a local applet to communicate with the DBMS—typically through a Java Database Connectivity (JDBC**) driver—and to then present those data to the InfoBus. Figure 7 shows this configuration, which can be used as a model for other forms of distributed data manipulation. For example, we have written an applet to take a named data item from the InfoBus and post it to an HTTP server using the Common Gateway Interface (CGI) mechanism.

We have found the InfoBus to be extraordinarily useful, occasionally in ways not originally anticipated. For example, the InfoBus is the conduit for passing selection information from the applet to the InfoCenter; when the user's selection changes, a DataItemChanged event is signaled to the InfoCenter. Similarly, once InfoBus functionality has been added to an applet—a task that is relatively straightforward—the InfoBus becomes a convenient communications portal. For example, we have written another very small applet that takes information from a JavaScript** user interface "widget" (e.g., a button-click event) and posts it to the InfoBus.

Programmability

Each applet can be driven by external programming mechanisms, and as such presents a public application programming interface (API). The architecture of the API was inspired by the "facade" model described by Gamma et al. in the seminal work *Design Patterns*,[5] the idea being to present a unified set of

method calls on the applet. The alternative was to expose the internal API of the applet itself and thus its object hierarchy and internal structure generally. This was felt to be not only confusing for the developer but also unnecessarily restrictive for us as we evolve the eSuite products. Having an API layer insulates the applet and thus allows the underlying structure to change or to be completely replaced.

The API itself is simply a set of documented Java public methods. In accordance with the JavaBeans component specification,[6] properties are retrieved and updated via "get/set" methods. Hence they can be introspectively viewed in Java-builder programs.

The API can be invoked from Java itself, or from any language capable of calling Java public methods, such as JavaScript, Object REXX (Restructured Extended Executor), VBScript (Microsoft Visual Basic** Scripting Edition), and others. Hence a JavaScript program on an HTML page can retrieve a handle to a given eSuite component and programmatically drive it, for example.

Applet overview

In this section the first set of eSuite applets is described at a functional level. The intent is less to provide a product description than to show how the eSuite applets incorporate the designs and technologies outlined in previous sections.

Spreadsheet. The eSuite spreadsheet component is a lightweight, two-dimensional spreadsheet supporting up to 256 columns by 8192 rows. It offers a reduced set of Lotus 1-2-3 internal functions (called "@functions"). It publishes named ranges of data to the InfoBus for consumption by other components, and it can read data produced by any InfoBus data provider. Persistent (file) data are saved in HTML format, and thus spreadsheet data can be read as HTML tables by browsers.

Word processor. With the word processor applet, users can construct simple documents that may include graphics. Basic, commonly used functionality is implemented: word wrap, multiple fonts and colors, tables, images, etc.

Chart. The chart component retrieves its data from the InfoBus as described earlier and renders a fully three-dimensional chart in a variety of formats.

E-mail. The electronic mail client is highly modular; it can support either the Internet Mail Access Protocol (IMAP) or the Post Office Protocol (POP), and its messaging transport layer is described formally by a set of Java interfaces; thus new or additional messaging transports can be created.

Calendar. The calendar component supports user appointments and utilizes the e-mail client to perform group scheduling. Communication between the calendar and the e-mail client is via the InfoBus.

Presentation graphics. This applet allows users to create and view simple presentations, and includes templates and drawing tools. It can also be used to build graphical navigators and be used on HTML pages to "jump" to a different slide in the presentation or to an address specified by a URL (uniform resource locator).

Database access. The database access component, possessing no user interface, is an interface between the InfoBus and a JDBC-compliant database. Its operation, somewhat simplified, is shown in Figure 7.

Sample

To illustrate the usage of the InfoBus and the applets, a brief example application will be described in this section. Consider the HTML page shown in Figure 8. Four eSuite applets are used to create this application: the spreadsheet, the chart, the database query applet, and a small applet used to transfer a JavaScript event (the button click) to the InfoBus. When the user presses the "Submit Query" button, the data query applet is activated to perform a query of the relevant data. The query applet has been programmed via an HTML parameter to post the results in an InfoBus data item named "ShoeQueryResult." Completion of this query results in an event, DataItemChanged, signaled on the InfoBus.

The spreadsheet applet possesses a named range called "ShoeQueryResult," which by the coincidence in names receives the data from the InfoBus. Another named range, "ShoeChart," holds the information to be charted. This is the name parameter passed to the Chart applet.

The appendix contains an excerpt from the HTML code used to set this up. Note not only the tags used to control the InfoBus but also those that control the appearance of the applets; in this case the InfoCenter has not been enabled, nor has the traditional inverted "L" of most spreadsheet programs.

Figure 8 Kathy's Shoes: InfoBus example

This then provides a much more seamless integration of the applets with the host Web page.

This application also illustrates the modularity of the eSuite components. Notice that because this application is not intended for content creation, but rather for data entry and analysis, the InfoCenter has not been enabled. The result therefore is that the end user's focus is on the task at hand, and not on the format of the data or the usage of the tool.

Conclusions

Lotus eSuite represents not just a use of the Java programming language in applications software; in fact we developed eSuite to respond to a shift in how computing is approached by individuals. We have provided basic functionality in packages reduced in size and simpler to use; we have made our components easy to "stitch" together into highly customized and tailored applications; we have engineered them so that they run well in a variety of environments and on a wide spectrum of hardware platforms. Thus eSuite is in fact the first incarnation of true "network-centric" computing.

Acknowledgments

The author wishes to acknowledge the contributions of the members of the eSuite team; in particular, Douglas Wilson, the inventor of the InfoBus; Noah Mendelsohn, a primary contributor to the JavaBeans specification; Jeff Buxton, architect and leader of the frameworks team; Jonathan Booth, senior overall architect; Howard Kirsten, lead user-interface designer; and Peter Masters, designer of the persistence architecture. Additionally the author wishes to recognize the extraordinary contributions of the entire eSuite team in delivering this seminal set of technologies and products to market.

Appendix

<!-- Include an instance of the Sheet applet, which gets the results of the query. The applet is programmed to load a template file that has all range names, formats, and formulas defined for presenting the results of the query. It is also programmed to disable column and row headers, the edit line, and the status line and to enable the horizontal and vertical scroll bars.

Note: The template file has two range names defined: "shoeQueryResult" where the results of the query are placed and "shoeChart" where the inputs to the Chart applet reside. -->

```
    <APPLET NAME="shoeSheet" CODEBASE="..\..\.." CODE="lotus.sheet.Sheet" WIDTH=281
      HEIGHT=352>
       <PARAM NAME="filename" VALUE="PQueryTemplate.html">
       <PARAM NAME="invertedL" VALUE="false">
       <PARAM NAME="hScroll" VALUE="true">
       <PARAM NAME="vScroll" VALUE="true">
       <PARAM NAME="editLine" VALUE="false">
       <PARAM NAME="statusLine" VALUE="false">
    </APPLET>
  </TD>
 <TD>
```

<!-- Include an instance of the Chart applet that graphs a range of data on the Sheet applet. The charted data are calculated from the results of the database query. The name of the applet (shoeChart) determines the name of the data item that is charted so it is important (in the context of this example) that the Sheet applet defines a range with this name. -->

```
    <APPLET  NAME="shoeChart" CODEBASE="..\..\.." CODE="lotus.chart.ChartApplet" WIDTH=472
      HEIGHT=352>
       <PARAM NAME="bgimage" VALUE="KS_chartbg.gif">
       <PARAM NAME="parseDirection" VALUE="column">
       <PARAM NAME="plotLayout" VALUE="Pie">
       <PARAM NAME="rendering" VALUE="3D">
       <PARAM NAME="viewPoint" VALUE="1000,1000,1000,-90.25">
    </APPLET>
  </TD>
 </TR>
</TABLE>
```

<!-- Include an instance of the JdbcSource applet to perform a specific database query. We make the applet invisible on the Web page by using a minimal WIDTH and HEIGHT setting in the APPLET tag and setting the "allowUI" parameter to false.

The "database," "ID," and "password" parameters specify the database to use for the query and the identifier and password to use when accessing it.

The "autoConnect" parameter is set so that the applet connects to the database once it is started.

The "retrievals" parameter specifies a list of query names to process; in this case the list contains only one query name: shoeQueryResult.

The "shoeQueryResult" parameter specifies the actual query to execute. Note that the query contains a parameter.

You can specify the format of each resulting data set by specifying a parameter whose name is the name of the query with "_resultStyle" appended. In this case, "shoeQueryResult_resultStyle" specifies that only the data are to be returned in the data set.

Note that a query in the list of query names is not processed until its query trigger is set on the InfoBus. The query trigger is a Boolean value. Its name is the name of the query with "_trigger" appended. -->

```
<APPLET NAME="shoeQuery" CODEBASE="..\..\.." CODE="lotus.jdbc.JdbcSource" WIDTH=1
  HEIGHT=1>
  <PARAM NAME ="database"
    VALUE="jdbc:dbaw://kona-dpp:8889/Sybase_SQLANY/PQueryData/PQueryData">
  <PARAM NAME="ID" VALUE="dba">
  <PARAM NAME="password" VALUE="sql">
  <PARAM NAME="allowUI" VALUE="false">
  <PARAM NAME="autoConnect" VALUE="true">
  <PARAM NAME="retrievals" VALUE="shoeQueryResult">
  <PARAM NAME="shoeQueryResult" VALUE="select Price,Quantity,Description from SalesData where
    State= '%state% ' ">
  <PARAM NAME="shoeQueryResult_resultStyle" VALUE="Data">
</APPLET>
```

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

Cited references

1. D. Clark and D. Bank, "Microsoft May Face a Backlash Against 'Bloatware,' " *Wall Street Journal* (November 18, 1996).
2. JDK 1.1.4 Object Serialization Specification, available from Sun Microsystems, or see http://java.sun.com/products/jdk/1.1/docs/guide/serialization/spec/serialTOC.doc.html.
3. M. Colan, *InfoBus Specification*, available from http://java.sun.com/beans/infobus/ibspec.html.
4. *JavaBeans*, G. Hamilton, Editor, available from Sun Microsystems (1997).
5. E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns*, Addison-Wesley Publishing Company, Reading, MA (1995).
6. See http://java.sun.com/products/jdk/1.1/docs/api/Package-java.ava.awt.html.

Accepted for publication March 16, 1998.

Barry Briggs *Lotus Development Corporation, 55 Cambridge Parkway, Cambridge, Massachusetts 02142 (electronic mail: Barry_Briggs@crd.lotus.com).* Mr. Briggs is a Lotus Fellow in the Internet Applications Division of Lotus, an IBM subsidiary. He has been with Lotus for ten years and served as chief software architect of Lotus 1-2-3 and other products, including Lotus Notes™. He holds B.A. and M.A. degrees from the University of Massachusetts, Amherst. Currently he works on a variety of software initiatives; his primary interest is in the area of client-side computing.

Reprint Order No. G321-5683.