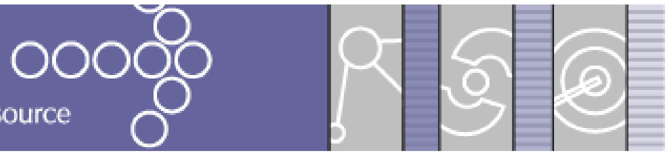
JAVA DEVELOPER'S JOURNAL
The World's Leading Java Resource


Rich User Interfaces

We're putting our money on GUIs not HTML

Most Java developers intuitively understand the advantages of using a rich Java Swing user interface instead of an HTML interface. The fact that rich user interfaces provide a better experience for the user has often been cited as the primary reason they should be employed in a particular application.

In this article, however, we provide four factors that support the claim that rich user interfaces may be the better choice for many applications - and not just because they provide a better user experience.

We begin by illustrating that rich GUIs can be developed and maintained at a lower cost than HTML-based user interfaces. We then argue that since Web services are typically highly interactive in nature as opposed to being a data delivery system, rich GUIs may be more appropriate than HTML interfaces for Web services. And finally, we describe a new architecture, which we refer to as the *Application Canvas*, for development of Web services-based applications that allow for integration of autonomous Web services on the client. As the application canvas is user-programmable, the user is able to link outputs of one Web service to the inputs of another, thereby, allowing the user to derive functionality not foreseen by systems developers.

Pros and Cons of Rich User Interfaces
We define rich user interfaces to mean Java Swing-based user interfaces. These interfaces improve user experience, but the advantages most overlooked are lower development and maintenance costs - and a high potential for reuse of interface components.

Some believe that rich GUIs are expensive to build. We argue that rich user interfaces can be built at the same cost, if not less expensively, than HTML-based user interfaces. Additionally, until recently, a case could be made that another disadvantage to rich GUIs was deployment. We'll address this issue by detailing the significant progress made in Java deployment technologies to drive deployment costs to a level comparable to the cost of deployment of HTML user interfaces.

The Case for Rich User Interfaces
Rich user interfaces are composed of high-level user interface components, such as tabbed panes, table widgets, and tree widgets. Compare an HTML tabbed pane with a tabbed pane in a rich user interface.

A tabbed pane is a single component in the Swing component set, and the properties of this component are standardized and well documented. In contrast, an HTML interface can display a tabbed pane; however, the designer must put a tabbed pane together from lower level graphics, HTML, and possibly JavaScript components. The construction of the HTML tabbed pane is likely to be nonstandard and may entail the development of a significant amount of scripting code. Therefore, developing a user interface that requires a tabbed pane in Java Swing will require less code to be written than a user interface with the same functionality developed in HTML. That's why sophisticated HTML user interfaces, especially when combined with JavaScript, may prove

to be more expensive to build and maintain than rich user interfaces composed of high-level user interface components.

Maintenance and cost of development of software systems, as well as user interfaces, are often measured by the degree of reuse achieved during development. Reuse is extremely difficult to achieve when the interface is built from low-level HTML and JavaScript components. But when the interface is built from high-level Java Swing components, reuse is much easier to achieve. Composite widgets can be constructed from Swing JavaBeans and later reused throughout the user interface of the application.

Another advantage to using high-level user interfaces is that the skills required to develop an HTML user interface and the skills required to develop a Java application are often possessed by different sets of professionals. Hypertext interfaces are best designed by graphic artists, who have training using HTML design tools. Many Java programmers are excellent in designing Java user interfaces but are not motivated to use graphics design programs that are often required to develop usable HTML interfaces. Therefore, developing an HTML interface will often require the hiring of a graphic artist in addition to the Java programmer developing the application.

High-level interfaces allow local updates whereas HTML interfaces require a whole page refresh. The fact that the whole page needs to be refreshed introduces additional delays the user must deal with. This fact may be a significant usability concern for those who must use the application every day. Users of mission- and time-critical applications, such as financial trading, supervisory control, and data acquisition systems, may find such delays a serious usability issue.

Highly Interactive Applications

Hypertext-based interfaces have been developed as a mechanism for browsing through large amounts of text. They are, therefore, well adapted for browsing text-based documents, such as electronic newspapers and other data delivery systems. But the hypertext model doesn't work well for many highly interactive software applications, such as financial trading systems, software development systems, control systems, and accounting systems. Many Web services are likely to exhibit more characteristics of interactive applications than data delivery systems. Therefore, rich user interfaces may be a better choice for many Web services-based applications.

The advantages of using high-level user interfaces as discussed so far have focused on usability issues, but the architectural advantages of using them may be of greater importance to systems designers. This may be especially true in the case of systems that are built from autonomous Web services components. Rich user interfaces allow separate Web services to be tied together using object-dependency mechanisms.

For example, a text field that is the output of one Web service can be set up to have dependent text field widgets that are inputs to other Web services. Dependency-directed backtracking algorithms can be used to trigger computations that are performed by a number of cooperating Web services.

The Case Against Rich User Interfaces

It's often argued that rich user interfaces are more difficult to develop than hypertext-based user interfaces. The difficulty most often cited is that Java GUIs require the development of a client application, or applet, that supports the interface. Once the application is developed, additional code needs to be written in order to connect the application with a server. The methods of connecting the client and the server have not been standardized. There are a number of possibilities ranging from Remote Method Invocation (RMI) to HTTP/XML-based schemes. In addition, a server-side portion of the application often needs to be developed in order to provide data marshalling and unmarshalling, as well as code that coordinates the logic contained in the client and the logic contained in the server.

Recently, however, a new architecture has emerged that allows a Java client to be deployed directly from servlets and JavaServer Pages (JSP). This architecture is identical to that used for the development of applications that present an HTML interface. The fact that the same architecture can be used for the development of Java and HTML clients suggests the development costs for each type of user interface should be similar. As we have argued earlier, however, development and maintenance costs may be lower for the rich user interface as it's constructed from higher-level reusable components.

Tipping the Scale

Deployment is the single most important consideration that can tip the scales in favor of HTML. HTML interfaces will run without installation on any machine that supports an Internet browser. The penetration of Java Virtual Machine installations, however, is far behind the number of machines that have a browser installed. Java deployment technologies have made significant progress in recent months.

Depoyment technologies such as Sun Microsystems' Java Web Start and Sitraka's Deployment Director can ensure that the client machine is precisely up-to-date. But these technologies require a one-time download of about 5MB and possibly some intervention from the user. That's why HTML may be the better solution for consumer applications that are intended for novice computer users and are deployed over the Internet in an uncontrolled environment. To deploy applications that are used for mission-critical functions over the Internet and intranets, current deployment technologies will prove to be adequate.

Rich Client Systems Architectures

With the advent of Web services, an application architecture based on cooperating Web services has emerged. Modern applications are constructed from Web services that are optimized for a particular purpose. For example, a car rental service can be combined with a flight-booking and hotel-reservation service, a personal calendar service, and a personal wallet service into a single application. We focus on two approaches for connecting cooperating Web services based on the Model View Controller (MVC) pattern in which the cooperating services form the Model layer. In the first approach, the Web services are connected on the server, whereas in the second approach, the services are connected on the client.

Connecting Multiple Application Services

Multiple services can be combined into a single application on the server and presented to the user as a single monolithic application that performs a particular unified service. The user is not aware that the application is actually composed of multiple services and doesn't need to understand the intricacies of each of the services. This architecture is depicted in Figure 1.

The Controller

The integration of multiple cooperating services is the function of the application controller. It's the application controller that must coordinate and tie the services together. The user of the application is then simply presented with a single application interface and isn't aware that the application is composed of several cooperating Web services. The application controller doesn't concern itself with persistence. Persistence is the responsibility of services that reside in the Model layer. The controller can often be programmed as a collection of server-side JavaBeans that connect to the required services and are accessed by the View layer to display a user interface.

The JavaServer Pages View Layer

The View layer can be implemented on top of a servlet engine as a collection of JavaServer Pages (JSPs) or servlets. If the view is composed of JSPs, then the view must be made specific to the output format expected by the client machine. In the case of HTML clients, the JSPs output

HTML. In the case of rich Java clients, the JSPs output a program or markup language that can be interpreted by the Java client. In the case of wireless clients, the JSPs output WML. It's important to emphasize here that rich Java clients shouldn't receive special treatment and shouldn't require development of an additional component to handle interaction with the rich client. If development of an additional component were required, additional costs would be incurred. The rich client must therefore be able to behave as if it were a browser and request new JSPs as its user "browses" through the Java Swing user interface.

The XML View Layer

If the View layer is implemented as a collection of servlets, the output from the servlets can be device-specific to the display format used by the client, or it can be made display-format agnostic. If the view servlets are to be made independent of the display device, they must output XML independent of all devices. The XML output must then be transformed into a display-specific markup or script language using an XSLT processor. Again, no special consideration should be given to the rich Java client. The Java client should be able to interpret output from the XSLT processor.

Push-Based Architectures

Certain rich client applications require the ability to push real-time data to the client as opposed to relying on refresh operations. In this case, the client must incorporate a listener component that listens to messages broadcast from the controller. The messages might be encoded in XML or simply contain a script that will then be interpreted by the client to display a user interface.

Remote AWT-Based Architectures

The Remote Abstract Windowing Toolkit (RAWT) architecture is similar to the X Window System pioneered in the Unix operating system. Essentially, RAWT allows an application executing on one machine to display a Java user interface on another. Events trapped on the client machine are transferred to the machine where the program actually executes. This architecture is similar to the thick client architecture, except that the application executes on the server as opposed to a remote client. The difference is, of course, that communication between the presentation layer of the application that displays the user interface and the rest of the application is done using local procedure calls as opposed to remote procedure calls.

The disadvantage of combining multiple application services on the server is that the user is deprived of the ability to combine multiple services in a way not envisioned by the programmer. In certain environments, users require a high level of customization and, often, frequent changes to the user interface and functionality of the application. This is the case when a new application is being developed and its users modify requirements as they learn more about how they could use the application to perform a business function.

The Application Canvas

The application canvas is the software that makes the integration of separate Web services possible on the client. The application canvas is like a modern-day spreadsheet that enables the output from one Web service to be tied to the input of another. The application canvas directs computation using a dependency-directed backtracking algorithm that computes the values of each individual cell - just as a spreadsheet would.

A key advantage of using an application canvas is that each Web service can now maintain its own separate user interface that allows the Web service to be used regardless of the presence of other services. That is, each Web application service can be displayed on the canvas and used independently. This frees system developers from building a server-side controller as the controller actually resides on the client in this case.

The user interface displayed by each Web service should be a rich user interface that allows the programmers and the users to create dependencies between output cells of one service and input cells of another.

Entirely Dynamic, Entirely Independent
To provide the elements of functionality needed to integrate separate application services into a useful application, the application canvas must incorporate most, if not all, of the following components.

The application canvas must provide support for JSP- and servlet-based rich clients. A desktop-like interface akin to that of the Swing JDesktopPane is a good choice for displaying windows that belong to autonomous Web services. It's important the user interface presented by each Web service is entirely dynamic so that the application canvas can remain entirely independent of the Web services it will manage.

Another important component of the application canvas is the dependency database that tracks dependencies for each input cell. The database must provide storage that persists across user sessions. The dependency database must be an embeddable database with a small memory footprint.

The dependency backtracking algorithm allows the application canvas to detect changes in output of one Web service and feed these changes to the input fields of another Web service.

The Thick Client
In cases where the client application must operate when the client computer isn't connected to the network, development of a thick client may be the only option. In this case, the client usually is required to store some data provided by the user. Often, the client will be synchronized with the server or with its peers when the computer goes online again. There are at least three classes of applications in this category: RPC-based, JSP/servlet-based, and P2P.

RPC-Based Clients
Remote procedure calls have been a popular means of developing client/server-based applications. Many RPC-based frameworks, such as RMI/CORBA, provide high-level functions that make it easier for developers to build RPC-based systems. Most RPC systems provide data marshalling and unmarshalling as well as toolkits for automatic proxy class generation. RPC is sometimes combined with XML where input and output parameters are encoded in XML as opposed to a binary format.

An important design consideration that arises when building RPC-based clients is that the client and server will need to contain additional components to interpret RPC messages on the server and on the client. Development and maintenance of these components may prove expensive. Whenever possible, using servlets and JSP communication should be considered.

JSP/Servlet-Based Thick Clients
JSP/servlet-based thick clients distinguish themselves from RPC-based clients in that the client is actually composed of two parts: the static part that's not dependent on the network connection and the dynamic part that is. When the client computer is not connected to the network, the client application executes the static part. When the client is connected, the dynamic part of the application based on the JSP/servlet architecture becomes active. In essence, this architecture is a thick Java client with an embedded thin Java client that's using servlets and JSPs to display a user interface.

The advantage of combining the thick client with a dynamic thin client is that the communication component on the client and server can now be eliminated from the design. But the application may still need to synchronize its data with a server or Web service.

P2P Framework
The most promising data synchronization approach is perhaps Sun Microsystems' JXTA P2P framework. The JXTA framework is an excellent synchronization data mechanism for the parts of an application that act as peers. RPC is an alternative to the P2P approach. In most cases, however, the RPC-based approach is likely to result in the development of more custom network interface code - resulting in higher development and maintenance costs.

Deployment Strategies
Once you have the rich client solution, you need to get it to the end user. Various techniques are available, each with its own pros and cons.

- ***Applet:*** Deploying the Java Swing user interface as an applet requires the end user's browser to have the Java Plug-in installed. The cost of requiring the Plug-in is a one-time download for each user of a 5MB-15MB setup file, where the size varies based primarily on platform. (Microsoft Windows is at the low end.) In addition, using the applet may or may not require changes to the HTML file to load. This depends upon the Plug-in version supported and the user's browser. Using an applet as the delivery mechanism does ensure that the user has the latest version of the client. Each time a user connects up to the back end, the applet can be downloaded. The Java Plug-in supports caching, so it won't have to download every time.
- ***Install Programs:*** InstallAnywhere and InstallShield are two popular solutions for deployment. These tools and others like them offer the ability to provide a complete Java Runtime Environment (JRE), the same 5MB-15MB download as the Java Plug-in, in addition to your rich client application. Installation via these solutions is something users are familiar with, but it can lead to users having multiple JREs on their desktops, one for each installable object. The major drawback of these programs is the coordination of new releases. You as the vendor must actively notify all users of the update and then each user must download the new release. New versions of these products are thankfully adding active update capabilities, so this last issue can be resolved.
- ***JNLP:*** Java Web Start is Sun's latest attempt to solve the incompatible browser runtime environment and deployment issues. It uses a protocol called *Java Network Launching Protocol* (JNLP) to provide for the installation of applications outside the browser but within a secure execution environment. It requires an HTML page for the initial load of the application but, once loaded, permits the browser to be shut down. There is also built-in support for incremental updates. Sitraka's DeployDirector is another JNLP-based deployment alternative that adds additional features.

Commercial and Free Software

A number of commercial and free software products have emerged that make it easier for Web services developers to take advantage of rich GUIs. Several of these are mentioned, however, this list includes only some of the more popular ones and is not intended to be exhaustive.

- ***Altio's flagship product AltioLive*** is a framework for the development of rich clients that execute as applets within a Web browser. Their front end uses an extensive proprietary widget set compatible with Java installations shipped with Internet browsers. The framework also includes a presentation server that drives the user interface. The presentation server uses proprietary protocols to communicate with the client. AltioLive also includes a design tool for developing the graphical user interface.
- ***Bean Markup Language (BML)*** is an instance of an XML-based language customized for the JavaBean component model. The language allows beans to be constructed and wired together. The language is designed to be directly executable. That is, processing a BML script results in the construction of a running Java application as described by the script. The BML language has elements for creating new beans, accessing existing beans, and executing bean methods.
- ***Droplets User Interface Server*** is a presentation server that extends some of the ideas developed by the Remote AWT project at IBM. In the case of the Droplets Server, a C++ API is provided in addition to the Java API. Droplets also offer some prebuild servers that handle a number of applications with rich Java GUIs that include e-mail, customer care management, and content management. Droplets also include an innovative deployment mechanism that allows application icons to be dragged from a Web page onto a user's desktop.
- ***The Remote Abstract Window Toolkit*** (AWT) for Java is a server-side implementation of AWT that allows any application running on one host to display a Java user interface on another host. The AWT calls made on the server are transmitted to the client for processing. That's why an AWT application running on the server can actually display its user interface on the client. The RAWT toolkit is available from the IBM alphaWorks Web site.
- ***Spidertop*** provides tools for the development of JSP and servlet-based Java user interfaces. The flagship product, called Bali, includes an interpreter JavaBean component that interprets a scripting language that can be served from servlets and JSPs to display Java Swing user interfaces. The framework also includes a graphical JSP builder incorporated into Sun Microsystems' Forte for Java that allows developers to build JSPs graphically. The framework also includes a presentation server that simplifies development of server-side controller components. The use of the presentation server is optional as the interpreter JavaBean can communicate with servlets and JSPs without the intervention of the presentation server.

Four Good Reasons

We have identified four factors that will drive the adoption of rich user interfaces for Web services-based applications.

The first factor is the emergence of a new class of architectures that permits the development of rich front ends based on servlet and JSP J2EE back-end systems. When using servlet and JSP-based architectures, development costs of rich GUIs are comparable to development costs of HTML-based front ends, while all the benefits of the Java GUIs are maintained.

The second factor is the interactive nature of Web services-based applications. Many Web services-based applications will exhibit more characteristics of typical computer applications than characteristics of information delivery systems. Highly interactive systems are better presented using rich Java front ends than HTML.

The third factor is the advances in deployment technologies for rich Java GUIs.

The fourth factor is the emergence of the application canvas. The application canvas will give developers a compelling reason for developing rich Java GUIs for Web services, as the application canvas permits the end user to tie separate Web services into composite applications not foreseen by the Web service developers.

We believe these four factors will result in a widespread adoption of rich Java GUIs for Web services-based applications.

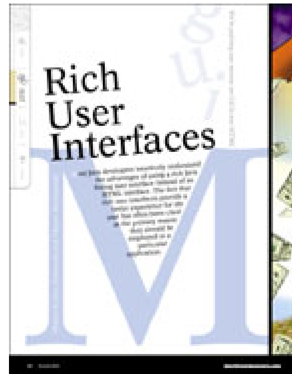


Figure 1

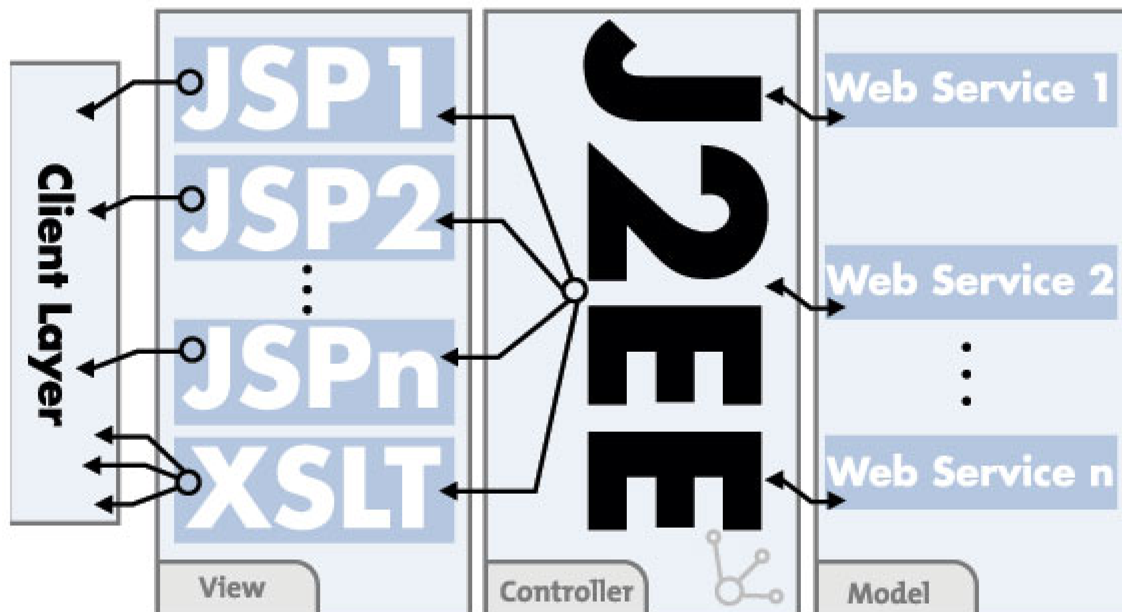


Figure 2

All Rights Reserved
Copyright © 2001 SYS-CON Media, Inc.
E-mail: info@sys-con.com

Java and Java-based marks are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries. SYS-CON is independent of Sun Microsystems, Inc. SYS-CON, JDJEdge 2001 International Java Developer Conference or Java Developer's Journal is not affiliated with Sun Microsystems, Inc.