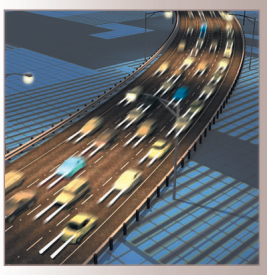
Editors: Michael Rabinovich • *mis@brics.dk*
Steve Vinoski • *vinoski@ieee.org*



Understanding UI Integration

A Survey of Problems, Technologies, and Opportunities

Creating composite applications from reusable components is an important technique in software engineering and data management. Although a large body of research and development covers integration at the data and application levels, little work has been done to facilitate it at the presentation level. This article discusses the existing user interface frameworks and component technologies used in presentation integration, illustrates their strengths and weaknesses, and presents some opportunities for future work.

The difficulty of creating applications from components has been a large area of investigation in software engineering and data management for the past 30 years. It has led to a body of research and development in areas such as component-based systems, enterprise information integration, enterprise applications integration, and service composition.

Although the results from these efforts have simplified integration at the data and application levels, the research community has done little work at the presentation level. User interface (UI) development is one of the most time-consuming parts of application development, testing, and maintenance,[1] so, clearly, reusing UI components is just as important as reusing application logic. But even though frameworks (such as Java Swing) help facilitate current UI development, the integration of coarse-grained and possibly stand-alone applications at the UI level hasn't received much attention.

In this article, we investigate the problem of GUI integration — that is, the integration of components by combining their presentation front ends instead of their application logic or data schemas. Here, component granularity is that of stand-alone modules or applications; the goal is to leverage components' individual UIs to produce richer, composite applications. The need for such integra-

Florian Daniel and Maristella Matera
Politecnico di Milano

Jin Yu, Boualem Benatallah, and Regis Saint-Paul
University of New South Wales

Fabio Casati
University of Trento

1089-7801/07/$25.00 © 2007 IEEE

tion is manifest, and examples are numerous: applications that overlay real estate information on Google Maps, aggregated dashboards that show consoles monitoring different aspects of a computer's performance (such as http://h20229.www2.hp.com/products/ovd/), or "Web" operating systems that allow coordinated interactions with multiple applications on the same Web page (see www.eyeos.org). All these examples require coordination among application UIs — zooming out on a map, for example, means that overlaid information on houses for sale must change as well.

This article aims to identify the basic characteristics of UI integration as a research discipline, discuss its main issues, and present the approaches that developers can take to address them. Specifically, we describe and exemplify the characteristics, challenges, and opportunities of UI integration in comparison with data and application integration. This is important not only to understand why UI integration differs from other integration problems — and, hence, requires unique solutions — but also to understand and learn from its similarities.

Integration Layers

To describe the different types of integration, we use a simple but concrete scenario based on actual applications developed at Hewlett-Packard. Consider a set of applications that monitor the performance and quality of systems, networks, services, and business processes. In this scenario, a system-monitoring tool logs metrics (such as CPU use) for a set of machines and sends alerts in case they go above certain thresholds, while a process-monitoring application looks at business-process executions and reports on key performance indicators such as process duration or process instantiation rate. Like most modern applications, each of these is structured into three layers: presentation, application (also called the business-logic layer), and data.

Historically, programmers developed monitoring applications independently, but we increasingly need to look at them in an integrated fashion. This is useful for root-cause analysis (to understand what system problem caused a delay at the process level) as well as business-impact analysis (to understand the "damage" that a backend system's failure or performance degradation caused at the process level), and, in general, to have an end-to-end view of managed IT systems. As a simple integration example, assume that two components form an integrated monitoring application: a business-process-monitoring tool and a system-monitoring tool. In an integrated application, when the user chooses to visualize a specific process in the process-monitoring tool, the health and availability status of its supporting systems should appear in the system-monitoring application. Let's examine how to build it.

In data integration approaches,[2] composite applications have their own presentation and application layers, whereas the data layer is an integration of the data sources that component applications maintain independently, as in Figure 1a. In our system-management scenario, the different monitoring applications collect data in their local repositories, unaware that they're the objects of integration. An integration layer brings data sources together and exposes a unified, homogeneous view to the composite application. The integration layer can materialize or remain virtual.

Data integration presents several issues, ranging from the resolution of mismatches between component data models (such as the same terms having different meanings) to the construction and maintenance of virtual schemas and query mappings between global and local schemas. It requires little "cooperation" from component applications: we can always tap into the applications' databases via SQL queries or Enterprise Information Integration (EII) technologies. The drawback is that doing so requires a significant effort to understand the data models, analyze semantic heterogeneities, and maintain the composite schema in the wake of changes to the component data schemas.[3]

Researchers have thoroughly studied application integration over the past 30 years, eventually giving us technologies such as remote procedure calls (RPCs), object brokers, and Web services.[4] In application integration, a composite application has its own UI, but its business-logic layer is developed by integrating the functions that component applications expose, as in Figure 1b. In our management scenario, monitoring applications could expose APIs that let clients retrieve performance data for certain systems or subscribe to alerts about performance degradation. The composite application uses these APIs to get information, correlate it across different monitoring applications, and display a consolidated overview on its GUI. When possible — that is, when such APIs are available — this integration model has several benefits, including

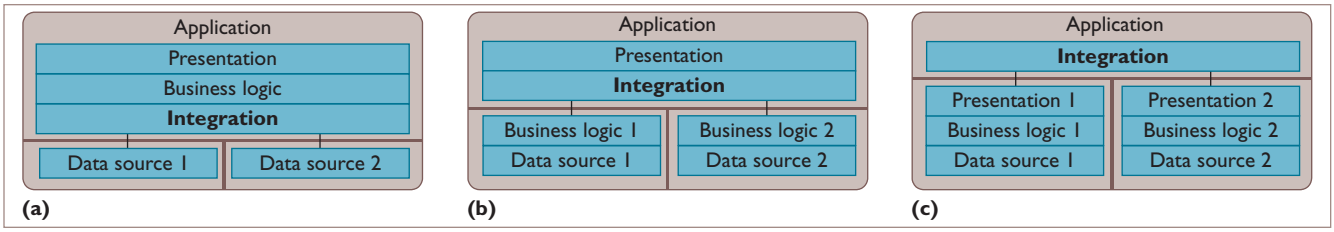


Figure 1. Component integration at different levels. The integration of (a) different data sources, (b) distributed business logic elements, and (c) two autonomous applications reveals how integration at the UI level can reduce development times and costs and thus foster the reuse of UIs.

- the granularity of the functions that the component applications provide is generally well suited for high-level integration (for example, we can tell an application to begin monitoring machine xyz without considering how this activity will affect data in the integrated application's database), and
- it's more stable because the component application is aware of the integration (it exposes the API) and will attempt to stabilize the interface across versions.

UI integration, as depicted in Figure 1c, should aim to compose applications in the presentation layer, leaving the responsibility of data and business-logic management to each component. UI integration is particularly applicable when application or data integration just isn't feasible (such as when applications don't expose business-level APIs), or when developing a new UI from scratch is too costly (such as when the component application changes frequently or its UI is overly complex).

The UI Integration Problem

Researchers have studied four classes of problems in data and application integration that are also key issues in UI integration. They're related to

- models and languages for specifying components;
- models and languages for specifying component composition;
- communication styles through which components can interact; and
- discovery and binding mechanisms (possibly enacted at runtime) for identifying components.

Ideally, models and specifications must be simple enough for users to understand and easily adopt, formal enough for applications and tools to parse, and expressive enough to model a wide range of concerns.

Let's examine these four dimensions and introduce a fifth that's more specific to the UI integration problem.

Component Model

In application integration, components are essentially characterized by an API and possibly a component model (such as in Corba). In data integration, data source schemas describe the components, but only recently have external specifications received attention in UI integration. Indeed, UI integration was mostly intended to reuse class libraries, but performing integration at the presentation layer also requires a component model that can support complex interactions and coordination. For each single component in our management scenario, for example, we must describe

- the software interface of the component used for integration, and
- the user interface of the component that enables the interaction with it.

We can distinguish among several "degrees of interoperability" that a UI component's interface allows.

GUI-only. Analogous to a traditional monolithic desktop application, all interactions with GUI-only components are performed through the component's UI logic. The only way to integrate a component application is to intimately know its UI, be able to track the user's mouse position or keystrokes, and thus understand what the component's UI shows and possibly even execute actions that cause UI modifications (such as by having the composite application simulate mouse clicks or keystrokes). Integration in this case is a daunting task.

Hidden interface. In many Web applications, the component has an interface that lets users control its UI, but it isn't publicly described. When interacting with a Web application, for example, we can

access and manipulate content by sending HTTP requests and displaying responses. Such applications obey a general protocol for interaction among clients (UIs) and applications, but it might be hard to identify how this protocol is formulated because it isn't assumed to be used programmatically by clients.

Published interface. In this ideal case, the component provides a public description of its UI and an API to manipulate it at runtime. A low-level API might allow control of individual UI elements such as button or text areas. A high-level API would instead expose a set of entities — that is, observable and controllable objects, such as "system" or "network" in our monitoring example, as well as operations to change entity status, such as "show status of system xyz."

Composition Language

In data integration, composition often occurs via SQL views that allow data designers to express a global schema as a set of views over local schemas.[2] In application integration, composition occurs either via general-purpose programming languages such as Java, or dedicated application integration languages, such as workflow or service composition languages (see www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsbpel). Although little work has been done in this direction for UI integration, we believe that similar approaches should work here as well. We distinguish between two kinds of composition languages.

General-purpose programming languages. Developers can adopt third-generation languages for application composition. Such languages are very flexible but lack abstractions to coarse-grained components (such as facilities for component discovery and binding or high-level primitives for synchronizing what UI components display).

Specialized composition languages. High-level languages, on the other hand, are typically used with an XML syntax tailored to the composition of UI components at the level of abstract/external descriptions. The main benefit of such languages is higher-level programming of the composition, which leverages the component model's characteristics. If the component model has the notion of entities, for example, the composition language would then provide primitives to directly deal with these concepts (such as change the displayed entity).

Communication Style

In our monitoring example, we want to know how monitoring components exchange UI events to receive instructions on what to display or notify other components about any user actions significant to the composite application (a user focuses on a different system, for example, so all UI components need to refocus as well).

In data integration, components are typically passive and don't initiate communications with the integrating application. This perspective is also common in application integration, in which a centralized entity (the composite application) invokes components as needed, although fully distributed interactions have become more common (such as a seller, buyer, and shipper interacting without a central coordinator).

In UI integration, we can also distinguish between centrally mediated communication, in which the composite application has a central coordinator that receives events from components and issues instructions to manipulate component UIs, and direct component-to-component communication, in which the composite application is a coalition of individual components — no first-class application orchestrates their activities. An additional distinction is the one between RPC-style interaction, in which components exchange information via method calls and returned data, and publish-subscribe interaction,[5] in which applications communicate in a loosely coupled way via messages exchanged through message brokers. In the latter case, a message broker distributes messages based on content or topic.

Discovery and Binding

Ultimately, UI integration involves identifying which components to integrate and how to get a reference to them (such as an object ID or a URI). We can do this statically (at design or deployment time) or dynamically (at runtime). In our monitoring example, the problem is how the composite application identifies and then binds to the relevant monitoring applications.

In data and application integration, binding between different data sources typically occurs at design time, when we define the global data schema. Although the integration middleware allows dynamic discovery and binding (see http://

microsites.cmp.com/documents/s=9063/cujcexp2007vinoski/),[4] this flexibility remains largely unexploited because of the difficulty of interacting with newly discovered components, especially when another company provides them (its reliability is questionable). In many cases, applications resort to combined static and dynamic binding, in which the application designer identifies and tests a set of potential components and the user then selects a subset of them at runtime based on the task; we call this solution *hybrid binding*, in which discovery is static, but we get the reference at runtime. The same distinction is possible in UI integration.

Component Visualization

So who is in charge of displaying a UI component, the component itself or the composite application? In our monitoring example, we want to know whether components display their own monitoring dashboard or whether the composite application receives UI markup code from the components and renders it.

Markup visualization requires interpretation from a rendering engine (such as a browser or a thin-client application) to translate the descriptions into graphical elements. Markup specifications typically describe static UI properties, whereas scripting languages provide dynamic behavior. We can also describe markup with document-oriented languages (such as XHTML and Wireless Markup Language [WML; www.openmobilealliance.org/tech/affiliates/wap/wap-238-wml-20010911-a.pdf]) or UI languages that model sophisticated application interfaces (such as eXtensible Application Markup Language [XAML; http://msdn2.microsoft.com/en-us/library/aa479869.aspx]; XML User Interface Language [XUL; www.mozilla.org/projects/xul]; User Interface Markup Language [UIML; www.uiml.org]; and eXtensible Interface Markup Language [XIML; www.ximl.org]).

For visualization, we distinguish between two different UIs.

Component-rendered UI. Here, the component handles the UI's rendering and display, thus the composite application is a collection of the components' UIs. This is the case with classical desktop applications that leverage executable components of linked graphics libraries.

Markup-based UI. Here, the component returns UI code and delegates the final UI's rendering to either the composite application or the environment in which the composite application executes. The composite application must thus be able to interpret the components' UI code and allocate suitable space on the display for component rendering. In this case, user interaction with the composite application generates UI events that the component can handle directly via a suitable scripting logic embedded in the component's markup, or the composite application can intercept generated UI events and forward them to the component for interpretation.

UI Composition Technologies

A few existing UI technologies provide some interesting insights and might even become candidate technologies for future UI composition. Table 1 compares the different UI technologies we consider in the context of UI composition.

UI integration should leave the responsibility of data and business-logic management to each component.

Desktop UI Components

UI composition was first considered for desktop applications. The introduction of component technologies eventually provided an environment in which applications developed with heterogeneous languages could interoperate. A typical example is ActiveX (http://msdn.microsoft.com/workshop/components/activex/activex_node_entry.asp), which leverages Microsoft's COM technology for embedding a complete application UI into host applications. Other examples include OpenDoc (http://developer.apple.com/documentation/mac/ODProgGuide/ODProgGuide-2.html) and Bonobo (http://developer.gnome.org/arch/component/bonobo.html), which rely heavily on the underlying operating system or on component middleware for interoperability.

By contrast, the Composite UI Application Block (CAB; http://msdn2.microsoft.com/en-us/library/aa480450.aspx) is a framework for UI com-

Table 1. Comparison of current user interface (UI) integration approaches.

	UI component model and external specification	Composition language	Communication style	Discovery and binding	Component visualization
Desktop UI components	Published, programmable API	General-purpose programming language	Centrally mediated and component-to-component communication could be supported	Static and dynamic binding	Component rendered
Browser plug-in components	Published, basic interface (startup configuration parameters)	Document markup code and JavaScript	Centrally mediated; very limited intercomponent communication via ad hoc JavaScript	Static binding	Component rendered
Web mashups	Hidden interface; published API	General-purpose programming language	Centrally mediated	Static binding	Typically markup based
Web portals and portlets	Standard interface based on public API; interface wrapped as a Web service	General-purpose programming language	Centrally mediated (interportlet communication under development)	Static and dynamic binding	Markup based

position in .NET with a container service that lets developers build applications on loadable modules or plug-ins. CAB components can be used with any .NET language to build composite containers and perform component-container communications. CAB further provides an event broker for many-to-many, loosely coupled intercomponent communication based on a publish–subscribe runtime event model.

Eclipse's Rich Client Platform (RCP; http://wiki.eclipse.org/index.php/Rich_Client_Platform) provides a similar framework but includes an application shell with UI facilities such as menus and toolbars; it also offers a module-based API that lets developers build applications on top of this shell. In addition, Eclipse lets developers customize and extend UI components (or plug-ins) via so-called "extension points," a combination of Java interfaces and XML markups that define component interfaces and facilitate their loose coupling.

Desktop UI components typically use general-purpose programming languages to integrate components (C# for CAB and Java for RCP, for example) because the component interfaces are language-specific programming APIs. Components perform their own UI rendering, and they could support flexible communication styles, including centrally mediated and component-to-component. Both design-time and runtime bindings are supported as well, with the latter relying on language-specific reflection mechanisms.

Many of the technologies for desktop UI components are OS-dependent. Although CAB and RCP don't depend on the OS directly, they rely on their respective runtime environments. The lack of technology-agnostic, declarative interfaces makes interoperation between components implemented with different technologies difficult to achieve.

Browser Plug-In Components

In markup-based interfaces, we can get rich UI features via embedded UI components such as Java applets, ActiveX controls, and Macromedia Flash.

The external interface of such components is very simple and usually requires only the proper configuration parameters when embedding components into the markup code (which represents the composition language). Plug-in components provide for their own rendering, with little further communication between components and the containing Web page, or among components themselves. A Web designer specifies component bindings at page-authoring time; during runtime, the browser downloads and instantiates the components.

Embedded UI components are easy to use, but their lack of a systematic communication framework is a limitation. They can communicate through ad hoc JavaScript, but this approach is far from uniform. However, this limitation comes more from the browser's sand-box mechanism for executing plug-ins than it does from the component model itself.

Web Mashups

Web mashups are Web sites that wrap and reuse third-party Web content (http://www-128.ibm.com/

developerworks/library/x-mashups.html?ca=dgr-lnxw16MashupChallenges). The first mashups couldn't rely on APIs because the actual content providers didn't even know their Web sites were wrapped into other applications. The first mashups for Google Maps, for example, predated the official release of the Google Maps API (www.google.com/apis/maps/). The API is Google's answer to the growing number of hacked map integrations, in which people read the whole AJAX code for the maps application and derive the needed functionalities.

Publicly available APIs for mashups are still rare, but their numbers are growing — most of them come from hidden interfaces. A developer thus performs the integration in an ad hoc fashion by leveraging whatever programming language the content source provides, either on the client or server side. Content providers typically provide content as markup code, and mashup developers integrate it in a centrally mediated way. Because content is markup based, the composite application (running in the browser) usually renders the components. The lack of infrastructure makes component–component communication difficult and only provides a way to statically bind components.

Because component interfaces might not be stable, most effort in mashup development is in manual testing. Due to the lack of framework support, code isolation isn't guaranteed, so conflicts among UI components can occur. Building a Web mashup is a time-consuming and challenging task.

Web Portals and Portlets

Web portal development explicitly distinguishes between UI components (portlets) and composite applications (portals) and is probably the most advanced approach to UI composition today. (The term *portlets* comes from Java Portlets [http://jcp.org/en/jsr/detail?id=168], but the following considerations on portlets also hold for ASP.NET Web Parts [http://msdn.microsoft.com/asp.net/default.aspx?pull=/library/en-us/dnvs05/html/webparts.asp].) Portlets are full-fledged, pluggable Web application components; they generate document markup fragments that adhere to certain rules, thus facilitating content aggregation in portal servers to ultimately form composite documents. Portal servers typically let users customize composite pages (by rearranging or showing/hiding portlets) and provide single sign-on and role-based personalization. Example portlets include weather reports, discussion forums, and stock quotes.

Analogous to Java servlets, portlets implement a specific Java interface to the standard portlet API, which was intended to help developers create portlets that can plug into any standard-conform portal server. JSR-168, for example, defines a runtime environment for portlets and the Java API (http://jcp.org/en/jsr/detail?id=286). For Java portlets, portal applications are based on the Java programming language, whereas with Web Parts, a Web developer programs applications in .NET. The portal application aggregates its portlets' markup outputs and manages communication in a centrally mediated fashion. Portlets also allow both static and dynamic binding; during runtime, the portal application can make portlets available in a registry for user selection and positioning.

JSR-168 doesn't provide interportlet communication mechanisms, but research is under way. Web Parts supports interpart communication with shared data structures, but this feature makes Web Parts tightly coupled — a publish–subscribe event mechanism might be more desirable.

Although portlets and Web Parts have similar goals and architectures, they aren't interoperable. Web Services for Remote Portlets (WSRP; www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsrp) addresses this issue at the protocol level by exposing remote portlets as Web services; communications between the portal server (WSRP consumer) and portlets (WSRP producer) occur via SOAP, which means developers can build the portal and portlets with different languages and runtime frameworks. WSRP 1.0 doesn't support interportlet communication, but ongoing work in WSRP 2.0 proposes an event distribution mechanism.

Although the efforts we've described here are certainly useful, we believe that effective standardization similar to that of standardizing service interfaces is needed for UI integration to really take off. In general, we see a lack of abstraction to conceptualize composition-oriented features in the context of UI integration.

UI development faces many challenges, including those related to software engineering issues and the human-computer interface. The integration of reusable components is a possible solution, but it directly affects the complexity and richness of the UIs we can produce for a given cost. Research in this very interesting and challenging domain is still in its early stages,[6] and we believe

that the models and infrastructures for UI integration will be an area of great interest in the coming years.

References

1. B.A. Myers and M.B. Rosson, "Survey on User Interface Programming," *Proc. SIGCHI Conf. Human Factors in Computing Systems*, ACM Press, 1992, pp. 195–202.
2. M. Lenzerini, "Data Integration: A Theoretical Perspective," *Proc. 21st ACM SIGMOD-SIGACT-SIGART Symp. Principles of Database Systems*, ACM Press, 2002, pp. 233–246.
3. A. Halevy et al., "Enterprise Information Integration: Successes, Challenges and Controversies," *Proc. 2005 ACM SIGMOD Int'l Conf. Management of Data*, ACM Press, 2005, pp. 778–787.
4. G. Alonso et al., *Web Services: Concepts, Architectures, and Applications*, Springer, 2004.
5. P.T. Eugster et al., "The Many Faces of Publish/Subscribe," *ACM Computing Surveys*, vol. 35, no. 2, 2003, pp. 114–131.
6. J. Yu et al., "A Framework for Rapid Integration of Presentation Components," to be published in *Proc. 16th Int'l World Wide Web Conf.*, ACM Press, 2007.

Florian Daniel is a postdoctoral researcher at the Politecnico di Milano, Italy. His research interests include modeling and design of Web applications, adaptivity, and business processes. Daniel has a PhD in information technology from Politecnico di Milano. Contact him at daniel@elet.polimi.it.

Jin Yu is a PhD candidate in computer science and engineering at the University of New South Wales, Australia. His research focuses on rich Internet applications and UI integration. Contact him at jyu@cse.unsw.edu.au.

Boualem Benatallah is an associate professor at the University of New South Wales, Australia. His interests include data and application integration, service-oriented computing, and data analysis for process and event-based systems. Benatallah has a PhD in computer science from the University of Grenoble, France. Contact him at boualem@cse.unsw.edu.au.

Fabio Casati is a full professor at the University of Trento, Italy. His research interests include business process intelligence, Web services, and data warehousing. Casati has a PhD in computer science from Politecnico di Milano, Italy. Contact him at casati@dit.unitn.it.

Maristella Matera is an assistant professor at Politecnico di Milano, Italy. Her research interests span modeling, design, and quality analysis of Web applications. Matera has a PhD in computer science from Politecnico di Milano. Contact her at matera@elet.polimi.it.

Regis Saint-Paul is an associate researcher at the University of New South Wales, Australia. His research interests include service-oriented architecture and data mining. Saint-Paul has a PhD in computer science from the University of Nantes, France. Contact him at regiss@cse.unsw.edu.au.