This article charts some basic approaches for designing components for composite application development that have maximum reuse. You can use many different strategies to provide the optimum components.

Composite applications are a key element in a Service Oriented Architecture (SOA) and contextual collaboration strategy. They support business flexibility for companies and organizations that want to respond rapidly to changing demands in today's competitive markets. Composite applications are aggregations of user interface components that are loosely coupled to support intercomponent communication. Components can be reused in multiple composite applications.

The ability to combine multiple technologies into a single application provides significant business value. It enables companies to protect and extend their existing assets with increasing degrees of flexibility, and it helps them respond quickly and cost-effectively to their emerging business requirements with applications that are significantly easier to create than in multiple application development environments.

This loose coupling of coarse-grained components is a cornerstone of the composite application architecture. It also enables diverse groups in different locations to leverage each other’s efforts and to interoperate with each other. Each group needs to understand only the inputs and outputs for a component and not the internal logic. For example, one department may be working on the accounting application while another group is working on a sales lead tracking application. In a composite application you can add some components from the sales lead tracking application to the accounting application to give pertinent views of assets in an accounting context. Similarly, the sales lead tracking application can incorporate components from the accounting application to give accounting information in an asset context. As you develop more and more composite applications, the potential for integration across your company increases exponentially. The goal is to make the whole greater than the sum of the parts.

The composition application model is already familiar to IBM WebSphere Portal developers. This approach has been extended to IBM Lotus Notes V8, enabling Lotus Notes developers to surface their Lotus Notes applications as one or more components in a composite application. IBM Lotus Domino Designer V8 has been extended to leverage the property broker as well as to
provide a more intuitive user environment. Lotus Notes V8 also supports the inclusion of Eclipse components, so a composite application may have any combination of Lotus Notes components and Eclipse components. This may just be done for on-the-glass integration (presented together in the same user interface) or, if extended, to use the property broker so that components can interoperate with each other. You can define composite applications using the Composite Application Editor or the WebSphere Portal Application Template Editor.

The development of components for composite applications is different from traditional Eclipse or Notes application development. You aim to create components that are sufficiently flexible so that they can be deployed in applications at later dates without significant rework. This article discusses approaches to designing such components and best practices to consider adopting.

Prerequisites

This document assumes familiarity with composite applications in Lotus Notes. Review the introduction to the composite application section in the IBM Lotus Domino Designer Help.

The tutorial, "Customer interests composite application sample for IBM Lotus Notes V8," can be used to gain hands-on experience by assembling an NSF-based composite application that includes NSF components and an Eclipse component that use inter-component communication.

It takes a village

Composite application components have a very open definition. The ability to do almost anything you like is powerful, but it can also lead to uncertainty about what you should do and how to establish a standard approach to development. Keep in mind that the categories of elements given in this article are suggestions for one possible approach. No approach is wrong. Any one approach either works for your company or does not. If you think our suggestions work for you, then adopt them. Feel free to change them if you have other requirements.

This sort of integration, though, does not happen purely by creating composite applications. It is possible to create components that operate no better together than two separate Web pages side by side. The key to a successful deployment of the component application architecture is establishing a common forum where dialogue takes place about the different development efforts. Be sure to create development processes in your organization for coordinating these efforts and for ensuring that you can capitalize on the potential benefits of composite application design.

If your company has something akin to an architecture review board, that may be an appropriate place for such dialogue to happen. If your company is smaller, even nominating a single person to be the data type owner serves this need. Such an organization must not slow down development. Too much focus on process cancels out the benefits you may otherwise gain. Instead, the review board is more of a brainstorming center, where different groups can inform each other of their ideas, and can realize the potential for benefit across all groups.
Types of components

There are very few technical restrictions on what can constitute a component. A component is, literally, a rectangle of UI that has some well-defined entry and exit points. To help us organize our approach to determining the components to build, let's define a few general categories.

The ability of composite applications to share components between them blurs the distinction between actual applications. And this is the perfect user experience. Users should be able to move between the activities they need to do without any actual awareness that they are moving from one domain to another. At the component development level, components tend to come from and be related to a single application domain. They have interaction points that allow connections to other domains, but in the most flexible design these are established at application assembly time and not at the component design stage. On the purely practical side of development, effort is usually divided according to application domain. Resources, timetables, and deliverables are usually accounted for in divisions, and so it is practical to look at the design along those lines.

The sales lead example is made up of three basic application domains:

- The core sales lead application, which tracks companies and sales leads
- A discussion application, which contains discussions about sales leads
- A legal application, which tracks contracts issued to customers and their revisions

Domain-centric components

Domain-centric components are the essentials of the created applications. When you create a composite application to cover the functionality of your domain, major UI elements may focus solely on your domain and make sense only in the context of your domain. Ensuring you have an array of components that can support all the existing functionality of an application is often the first tier of component development. If you are converting an existing application, one approach is to first ensure parallel functionality in your composite application. These large blocks bring the most functionality in the quickest way possible. They may be used a few times in your application, but they are less likely to be used by another composite application. Consequently, these components may be more tightly coupled for additional richness.

In the sample application, components such as CompanyList, SalesLeadList, SalesLeadDetail, and SalesLeadEdit are examples of domain-centric components. See figure 1.
Figure 1. Examples of domain-centric components
Domain-contextual components

If domain-centric components are the essentials of an application domain, then domain-contextual components are the trimmings. These are the components that decorate the peripheral view in your application and provide domain-related information, but in a context-specific way or by belonging to a particular information subset. In a classic conversion approach, domain-contextual components are the second tier of development. They are not essential to the fundamental workings of your application, but they increase usability by showing contextual information and saving on page transitions or cut-and-pastes. As such, they are likely candidates for inclusion in other composite applications. In the same way, they can provide domain-related information when there is a common element in both application domains.

Alternatively, another approach to converting an application to a composite application is to first produce some contextual information components in a way suitable for dropping into other composite applications. This approach demonstrates the value of reuse early in your adoption of the composite application architecture. If the application is large and complex, a longer schedule may be appropriate for total conversion. If the application is done with a different architecture (for example, a green-screen application running on text terminals served by a backend system), then total conversion does not make sense. But just as Web services provide access to these backend systems, domain-contextual components can be developed to bring information out of these systems (quite possibly using the same Web services) for reuse in new composite application projects.

In the sample application, components such as CompanySummary, LeadSummary, and LeadList ConstrainedByCompany are examples of domain-contextual components. You can see in figure 2 how they are deployed in the discussion and legal application domains to bring in related core information within their contexts.
Figure 2. Examples of domain-contextual components

Generic and utility components

As you develop a composite application, you come across elements that are not specifically tied to your application domain. These are useful components that have applications for anyone developing composite applications.

In the sample application, components such as Pager, Browser, RSSReader, and MailSearch are examples of utility components. Lotus Notes comes with several components of general use. For
example, the Lead Manager sample application, which makes use of the names and address book components, and the Managed Browser are both provided with the product.

**Central activities**

Now that you have a framework for classifying types of components, let's discuss the processes that a central group may organize around composite application development.

First, having a central forum for development groups to present their plans for composite application development is of immeasurable benefit. One group's application domain may be far removed from that of another, but lessons can be learned from one group's development efforts and applied to another group's. Keeping groups informed of plans and getting feedback can only improve the process.

Another reason for common presentation is to foster a language for composite application development. The process for the design and development of composite applications is the same as any other application development process. It requires establishing requirements, specification, test planning, and rollout. Composite applications are a little different because component development may not have a definitive application target, and application assembly can occur on an ad hoc basis. By sharing each group's process for how specifications are written, how test plans are developed, and how rollouts are organized, you can leverage efforts and optimize resources. These will be covered in detail in a future article in this series, "Designing Composite Applications: Managing the Process."

In a company where rigorous development is required, such as one subject to heavy government regulation, or in a company where diverse efforts are part of a coordinated public application, such a forum may be more formal, and approval may be a requirement before development can proceed. In a less rigorous environment, approval may be a courtesy. But even as a courtesy, the exchange of more information that is part of the approval process can mean greater potential reuse later.

These benefits help the process of developing domain-centric components. Greater benefits are achieved for domain-contextual components. Because these are components designed for good use outside an application domain, presenting plans for them (and demonstrating when they have been produced) is a great place to judge feedback. Your design may be perfect for another group's use, or that group may want to make a small change. Or they may want the same component, but with a slightly different information set. Also, because other people are focused on their application domains, they may have ideas for how you can use their domain-contextual components in your application to benefit your users.

Another great fallout from central group discussion occurs in the determination of generic and utility components. You can promote components for general use, so that they can be used in all domains instead of being restricted to a single domain. Some components may start by belonging to one group. As other groups make use of them, they may transfer to other groups that add features they need. As the effort progresses, the component becomes more feature-rich and, as
long as strict backward compatibility is maintained, the other applications that deploy it benefit as well.

An alternative to this serial process occurs when a group that needs a component with cross-group potential brings it to the central discussion body. If other groups are interested in the component, they may be able to share resources to the common benefit. One group may contribute to the development of the component, and another may contribute to its testing. Richer components and applications can result.

In the included sample, there is a Tag Cloud component. This component was originally developed for the sample with minimal requirements. Another group saw that we were using one, though, and they decided that the Tag Cloud component would benefit their own application. They took over its development to enhance it with their own features. Because they didn't break the basic features required for our sample application, we continue to use it and derive benefit from the features they added.

The most important technical aspect of a review group is maintaining a common set of data types. Components communicate with one another in terms of data sent from properties to actions along a wire. These wires cannot be made unless both endpoints use the same data type. Two components may have the concept of a date, but unless the dates are identically defined (namespace, name, type), they cannot be connected. A review group (or even an individual data type owner) can maintain a list of common data type definitions across the company. As each group presents its anticipated efforts, it should also identify the different data types required for properties and actions. Ones that match with existing company definitions can be determined, and the common definitions can be used. The new and unique ones can be added to the list of company definitions.

Domain-centric components often talk only to each other. Many of the data types that they define may not be of general interest. Domain-contextual components, however, ideally have interest from a wide variety of consumers. For these, it is essential that coordination for data types takes place.

Your goals are to maximize the reuse of components, to minimize the time to assemble composite applications, and to allow line-of-business users to easily extend and modify existing composite applications by adding domain-contextual, generic, and utility components.

**Component pattern**

Several types of repeated patterns occur when you develop components for use in composite applications. These are handy shorthand options to consider when you design the components that you need for a composite application. Similarly, layout patterns occur that are useful ways to arrange collections of components, and application patterns can give structure to your entire application.

These will be covered in detail in a future article in this series, "Designing Component Applications: Design Patterns."
**Deciding component granularity**

You can take different approaches as you decide exactly what to deploy as a component. If you design things from scratch, as with a new application, picking one of the component patterns described previously can help. With this as a basis, you can see which portions of your application domain fit those patterns.

Another useful method for new applications is to work backward. Generate what you want the application to look like, and then determine the components you need to build that. If you start from the screens depicting the application, in a storyboard for example, you can first identify areas in common. Any piece of the user interface that is the same (or similar) on several application pages is a candidate for becoming a component.

The next step is to go through your storyboard and look for sections of user interface that belong to other application domains or that may make sense being used in other application domains. Using this method of successive division, you can turn your design into components.

Designing components to port an existing application is both harder and easier. It is easier in that you already have an application design. But it is harder because you have to compromise between using what you already have and designing for component reuse.

In the case of a Lotus Notes application, you can work with some fundamental types of user interface. The future article, "Designing Component Applications: Lotus Notes Components," details an example of converting a discussion database to a composite application. The principles outlined there are applicable in general to many types of Lotus Notes applications.

With an Eclipse-based application, you have an advantage because the application is already based on views, which translate directly to components. Lotus Notes V8 offers richer interaction and reuse capabilities; the level of granularity chosen for projecting your application onto Eclipse may not drive the best benefit when projecting it as a composite application. Using your Eclipse views (or perspectives, if you want to go further back), you can apply the method of successive division to derive your components.

**Communicating among components**

Components communicate with one another through the use of properties and actions. You can use components in several different ways to conduct operations.

**Properties**

Properties represent outgoing values. The simplest method is for propagating a simple data value. For example, if a component presents a list of data items, it may broadcast a property representing the name of the currently selected item. You have a number of choices when you broadcast such values. Data items usually have a number of entries, for example, name, part number, supplier code, quantity in stock, and so on. You could choose to broadcast a separate property for each entry. On large or complex data items, this broadcast could get quite complex.
The present implementation of the property broker allows you to broadcast only items that can be expressed as strings. You do not have the option of broadcasting a complex item as-is.

One approach to get around this, though, is to define a method for serialization. The details of the complex item could, for example, be represented with an XML data structure and subsequently serialized into a string. This technique is often used for Web services. Some drawbacks should be noted. First, any time an element changes, the entire structure has to be broadcast. This approach is appropriate only for groupings of settings that have a relationship on the backend. Second, there is a reduced chance that this component can interoperate with components from other domains. Although individual entries may have types in common, if the overall data structure is specific to the application domain, in order for another component to take advantage of it, it has to know about it. This limitation moves the design away from being loosely coupled.

You can also use properties for event notification. Event notices are broadcast only when something changes. For example, a component performing a complex operation may broadcast its state. While it is submitting, it may broadcast a value in a state property to indicate this. Other components may be structured so that they can listen to this state and disable themselves when the component they are linked to is in an inappropriate state.

For greater reuse, individual components should avoid dictating the application flow. Instead, they can defer that decision to assembly time by signaling their view state change. A common component for handling transitions between components can then coordinate the change (again, this decision is deferred until assembly time, not design time, thereby increasing the reusability of the component).

For example, in the included Sales Lead application, many components broadcast a property called ViewState. This property contains a predefined value of the sort of operation or state transition the component wants to propagate up to the application level. For example, the Company Edit component sets this to #company.save after you save it or #company.close after you click the Close button. At application assembly time, the components are wired to a page switching component that maps these notifications to the specific page transitions desired.

**Actions**

Actions represent incoming values. Like properties, actions can be used in several different ways.

As with properties, the simplest way is for a simple data receiver. The component may consume this property and just store it, pending a later operation, or it may do a single data-oriented operation, such as updating the value of an edit field. For these actions, the data types directly correspond to the field they represent.

Another typical use is to consume an action that has a direct result on the display. Most commonly, this takes the form of passing an index to an information-retrieving component, which looks up the corresponding data record and displays the contents or a subset of the contents. Resultant values may be further propagated through properties. The data type of this is typically an unique ID or other index. For a listing component, this may be a sort or search criteria. In other uses, it may control the display mode of a component, or enable or disable the component based on the role of
the user. If this is an index, it usually takes on the data type of the corresponding field. For a more
general search string, you can define a special type, but for maximum interoperability, you can also
consider making it a generic string type.

The case in this article deals with service-type operations for actions, but only in the case where
they accept a single argument. What if you have a service you want to expose through the
component that requires several inputs? Now you can pass only a single argument to an action.
You have two choices to support more complex services. First, you can take the same approach
as you did with properties; you can serialize the required arguments together into a single type of
string. For example, the mail component accepts an action with a data type containing a mailto:
string. This, effectively, serializes a list of to, cc, and bcc addresses, plus a subject and body
field. As with serialized values for properties, the disadvantage is that you have a limited use
type, possibly unique to this action, which makes interoperation difficult. Additionally, if the input
values come from different components, assembling them together into a single serialized string is
difficult.

The other approach is to have several data receiver actions correspond to each possible
argument. Last, you have a doit type gesture in the user interface, for example, a button, a hot
spot, or a right menu click. When this gesture is triggered, it takes the values that have been
previously set, and if all non-optional ones are set, then it enacts the operation. (If insufficient
values have been set, the UI can show state to indicate this. For example, a button could be
disabled.) This approach has the advantage of being based on simple types, and it is useful in a
variety of circumstances where you may not have all data available in the context or the data may
be coming from several components.

**Conclusion**

The composite application architecture enables your company to combine multiple technologies
into a single application, bringing key application capability within easy reach of your users. In
addition to the greater productivity your users enjoy, creating richer composite applications by
leveraging existing interoperable components increases your development organization’s ability to
respond quickly and cost-effectively to emerging business requirements.