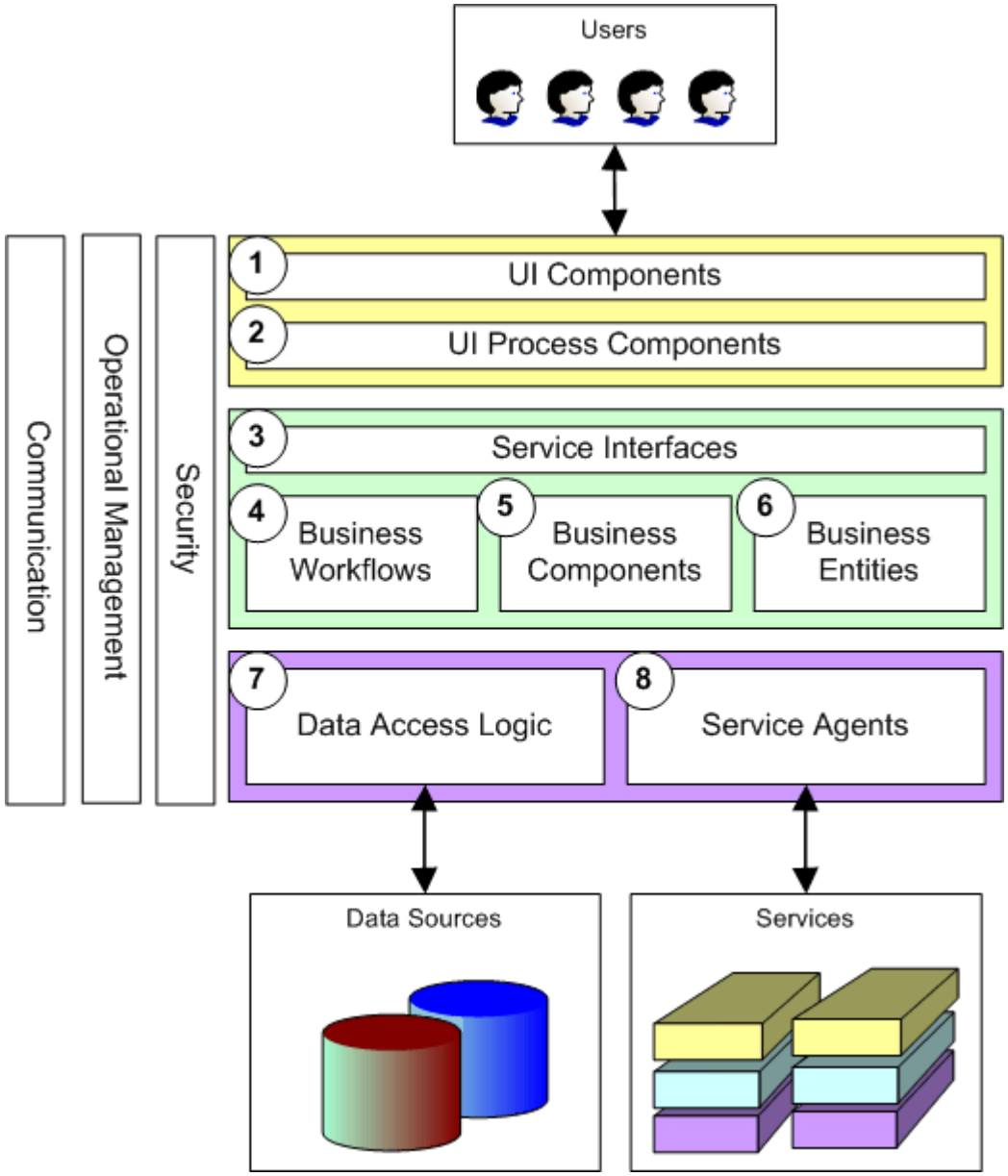
Designing Components for an Application or Service

Summary

Detailed Summary

Component Types

Every software solution, regardless of its specific business needs, uses similar kinds of components. For example, most if not all distributed applications need to use a data access component, Identifying these kinds of common components found in distributed applications will help you build a blueprint for an application or service design. The following figure shows common component types found in most distributed applications:



These components types are described below:

1. **UI Components**
   User Interfaces are implemented using Windows Forms, ASP.NET, controls, or any other technology uses to format/render data for users and to acquire/validate data coming from users.
2. **UI Process Components**
   In many cases, a user's interaction with the system follows a predictable process. For example, to search for an item you may need to specify some sort of product/item ID, apply an appropriate filter, and then indicate how output should be formatted. Rather than hard-coding this logic within the UI elements, you

can use a UI process component to help synchronize and orchestrate these user actions. UI Process Components (or UI Managers as I prefer to call them) ensure that the process flow and its state management are encapsulated within a component that can be potentially reused by multiple UIs. In the example above, a `SearchManager` is the appropriate UI process component.

3. **Service Interfaces**
   Service interfaces are often referred to as *business facades*. Service interfaces are used to expose business logic as a service.
4. **Business Workflows**
   Business workflows define and coordinate long-running, multi-step business processes. Many business processes involve multiple steps that must be orchestrated and performed in the correct order. For example, committing an order may involve multiple steps (verifying inventory, calculating total cost, authorizing payment, arranging delivery) and can take an indeterminate amount of time to complete. Therefore, tasks involved in committing an order and the data required to complete them would have to be managed.
5. **Business Components**
   Business components implement the business logic of the application. Business Workflows use business components. From the previous example given in Business Workflows, the tasks involved in committing an order would each be implemented as a business component - you would have a component to verify inventory, another component to calculate total cost, and so on.
6. **Business Entities**
   Business entities are internal data structures that represent real-world entities that the application has to work with such as a product or an order. Business entities are often implemented as `DataSet`s, `DataReader`s, or XML streams so that they can be passed between data access components and UI components.
7. **Data Access Logic**
   Data Access components are used to abstract data access in a separate layer. Doing so centralizes data access functionality and makes it easy to maintain.
8. **Service Agents**
   Service agents can be considered as *adapters* that convert the varying interfaces of different services into compatible interface with the client application. Service agents can also provide additional services such as mapping between data formats required by the application and the service. For example, if the credit card authorization service is on a Unix machine, you could use a service agent to manage communication via TCP/IP and translate data formats back and forth between Unix and the .NET application.

The remaining components, security, operations, and communications involve components used to perform exception management, caching, authorizations, and other tasks required to communicate with other applications and service. and will be discussed in detail in Security, Operational Management, and Communications Policies.

The following table roughly illustrates how the given component types (process components, data access logic, etc) map to certain design patterns:

Component Type	Related Design Pattern
UI Component	Presentation Layer View Layer Client Layer
UI Process Component	Application Controller Pattern Mediator Pattern Application Model Layer
Service Interfaces	Remote Facade Pattern
Business Workflows	Domain Layer
Business Components	Domain Layer Transaction Script Pattern
Business Entities	Data Transfer Object Domain Model
Data Access Logic Components	Data Source Layer

	Infrastructure Layer Integration Layer
Service Agents	Data Source Layer Infrastructure Layer Integration Layer

Design Recommendation for Applications and Services

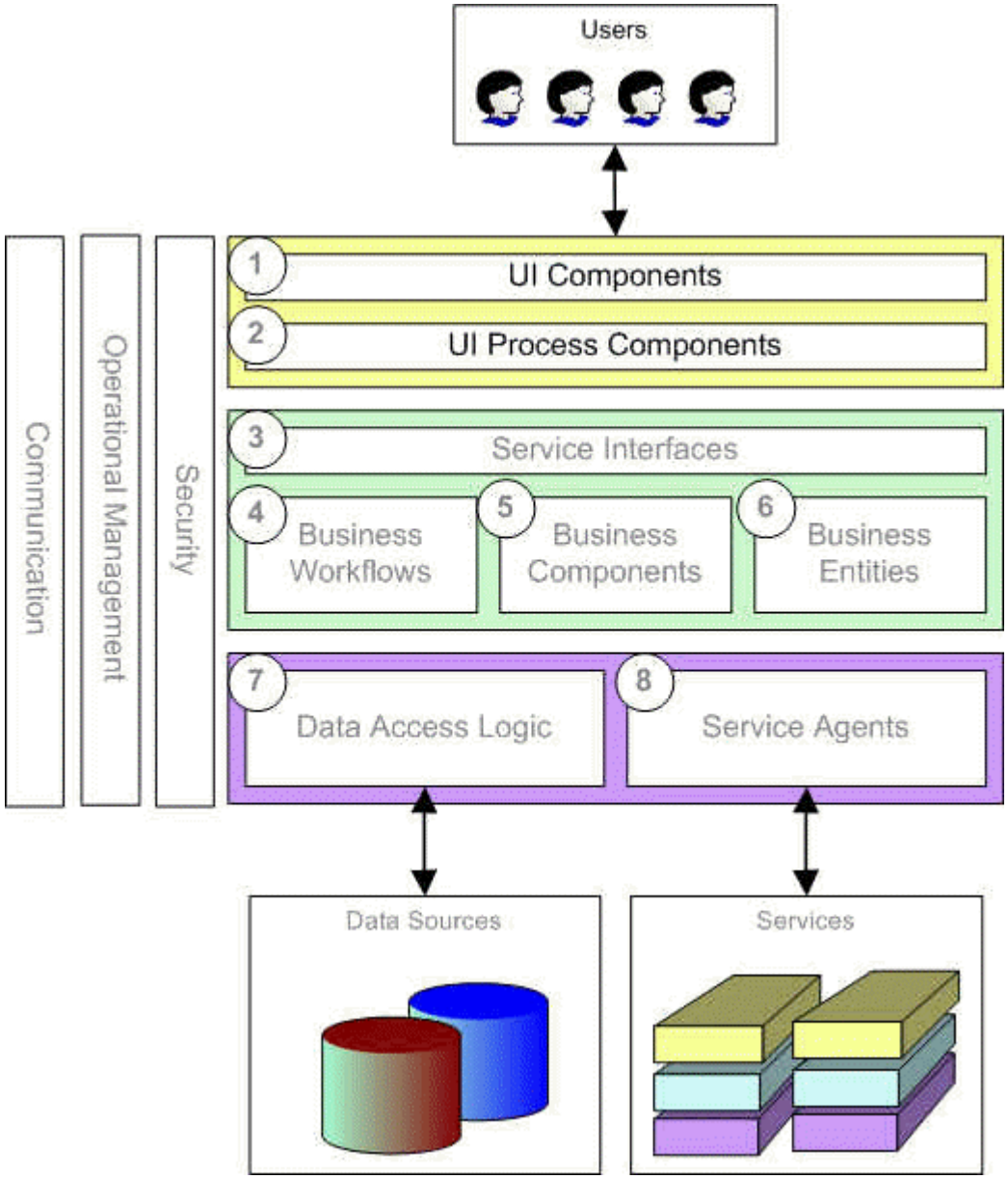
Consider the following recommendations when designing applications and services:

- **Identify required kinds of components**
  Some applications have only one user interface with simple UI and may not require UI process components. Other applications may be simple is terms of their business processes and do not need to integrate with other services via service agents.
- **Understand component communication**
  Component communication can affect physical distribution. Keep coupling low and cohesion high by choosing coarse-grained rather than chatty interfaces for remote communication.
- **Use consistent data format**
  Mixing XML, strings, `DataSet`s, `DataReader`s, serialized objects and other formats in the same application will make the application more difficult to develop, extend, and maintain. If you must mix data representation formats, keep the number of formats low. For example, the data access layer can return data in `DataReader`s for ASP.NET front-end, but may return `DataSet`s for user by business processes.
- **Abstract policies from application logic**
  Keep code that enforces policies abstracted as much as possible from the application business logic. Rely on attributes, APIs or utility components to provide a *single-line of code* access to functionality related to policies such as authorizing users, publishing exceptions, tracing, and so on.
- **Determine kind of layering**
  In a strict layering system, components in layer A can only call components in layer B and cannot call components in layer C. In a more relaxed layering system, components in a layer can call components in layers that are not immediately below it. In all cases, try to avoid upstream calls and dependencies in which layer C invokes layer B. I prefer to use a relaxed layering system to prevent cascading changes throughout all layers whenever a layer close to the bottom changes, and to prevent having components that do nothing but forward calls to layers underneath.

Designing Presentation Layers

Definition: The presentation layer consists of all components required to enable user interaction with the system.

The most simple presentation layer contains UI components such as Windows Forms or ASP.NET. For more complex user interactions, you can design UI process components to orchestrate UI elements and control user interactions. UI process components are useful when user interactions follow a specific pattern (just like a wizard used to accomplish a task).



Designing presentation layers contains two broad topics:

- Designing UI Components
- Designing User Process Components

Designing UI Components

UI Components, which manage interactions with the user, may be implemented in different ways. For example, you can have Web-based UI, Windows-Forms UI, voice rendering, document-based programs, mobile client applications, and so on.

When accepting user input, UI components::

- Must not initiate, participate, or vote in transaction.
- Must acquire data and assist in data entry using visual clues, validation, and the appropriate controls for the task.
- Must capture events from the user and call the appropriate event-handlers.
- Must restrict the types of input a user can enter. Numeric fields should allow non-numeric characters.
- Must perform data entry validation - restricting range of value or ensuring that mandatory data is entered.
- Must perform simple mappings and transformations of data entered by the user to values needed by the underlying components..
- Must interpret user hints such as drag-and-drop.
- May use a utility component for caching reference data that changes infrequently
- May use a utility component for paging where long lists of data are displayed as paged sets.

When rendering output, UI components:

- Must perform formatting of values.
- Must perform any localization work on the rendered data.
- Must provide the user with appropriate status information - for example, "processing...", "displaying...", "connected", "disconnected", etc.
- May customize the appearance of the application based on user preferences.
- May use a utility component to provide *undo* operations.
- May use a utility component to provide clipboard functionality.

While the list is not exhaustive, it lists some of the more common requirements that all UI components should provide. UI design requirements for more specific types of UI are shown below:

Windows Desktop UI

Windows UI is typically used when you need to provide rich user interaction and/or offline (disconnected) capabilities, or even integration with the UI of other applications. Windows UI can take advantage of a wide range of state management and persistence options and can access local processing power.

There are three main families of Windows standalone user interfaces:

- **Full Blown**
  Gives the greatest amount of control over the user experience and total control over look and feel of the application. However, it ties the application to a client platform and the application needs to be deployed to the users.
- **Embedded HTML**
  You can use additional embedded HTML with the Windows Forms application to allow for greater run-time flexibility as the HTML can be loaded from external sources or from a database in disconnected scenarios. However, additional coding is required to load the HTML, display it, and hookup events from the HTML control to the application logic.
- **Plug-In**
  UML Use Cases may suggest that your UI could be better implemented as a plug-in for other applications. In this case, you only need to worry about code to

collect data and work with your business logic.

Consider the following design recommendations when creating Windows Form-based applications:

- Use data-binding to keep data synchronized across multiple forms that are open simultaneously. No need to write complex data synchronization code.
- Avoid hard-coding relationships between forms and rely on the user process components to open them and synchronize data and events. Be especially careful about hard-coding relationships from child forms to parent forms. For example, a product-finder form can be used from any other form and not just from an order-entry form.
- Implement error handlers all forms. For example, all event handlers must include catch handlers. You may even implement an exception hierarchy where each form has its own exception class that can be used to set form-specific exception information.
- Validate user-input in the user interface. In some cases, you may have to enable/disable controls and visually cue the user when input is invalid. Validating user input in the user interface prevents unnecessary round-trips to server-side components.
- If you are creating custom user controls, expose only public properties and methods.
- Do not implement event handler functionality directly in the event handler. Implement all event handler functionality in a separate class to increase maintainability and reuse. For example, when using grids and editing cells you often have to implement a hander for a `BeforeCellUpdate` event where you verify user input, and if input is invalid you display a message box. The verification functionality should be implemented in a separate helper function `VerifyUserInput` that will be called by the `BeforeCellUpdate` event handler:

```
private void grid_BeforeCellUpdate( object sender, System.EventArgs e)
{
    VerifyUserInput( grid.GetCurrentCell );
}
```

Internet Browser UI

Consider the following design recommendations when creating ASP.NET user interfaces:

- Implement a custom error page, and a global exception handler in global.asax.
- Because client-validation at the browser relies on having JavaScript enabled, you should validate data in the controller functions just as well. If the UI user process has some sort of a `Validate` function, call it before moving to other pages.
- If you are creating Web user controls, expose only public properties and methods.
- Use ASP.NET view state to store page-specific state. Keep session and application state for data with a wider scope.
- Event handles and controller functions should invoke functions on the user process component to guide the user through the current task rather than redirecting the user to another page.
- Do not implement event handler functionality directly in ASP.NET pages. Implement all event handler functionality in a separate class to increase maintainability and reuse

```
<asp:Button id="btnAdd" OnClick="btnAdd_Click" />
```

```
/* Event handlers */
private void btnAdd_Click(object sender, System.EventArgs e)
{
    AddItems();
}
```

```
/* Helpers */
private void AddItems()
{
    // Code to implement adding items to the shopping basket
}
```

Document-based UI

In some cases rather than building a custom Windows Forms application to facilitate user interaction, it might me more appropriate to allow user to interact with the system through documents created by common tools such Microsoft Office. Documents can be considered as a metaphor for working with data, and users may benefit from viewing/entering data in using documents from tools they often use.

Consider document-based UI in the following cases:

- **Reporting Data**
  Your application - which may be Windows Forms or ASP.NET - may provide a feature that allows users to view/manipulate data in a document of the appropriate type. For example, grids may have to option to export data to Excel where users can view/ manipulate data.
- **Gathering Data**
  Users may use Excel to enter order data and then submit the document to some business process.

In general, there are two common ways to integrate documents into applications, working with the documents from the inside and working with the documents from the outside:

Working with Documents from the Outside

In this scenario you treat the document as an entity. The document itself has no specific awareness of the application. In this design model, the UI will perform the following functions:

- **Reporting Data**
  The reporting code obtains reporting data (from some ongoing business process or from the database), and then generates a document, injects data into it, formats it, and then finally present it to the user.
- **Gathering Data**
  A user starts by entering information in a document and then submits it to a Windows-based or Internet-based application. The application scans through the document's data through the document's object model, and then performs the necessary actions. At this point, you may decide to either preserve the document (for auditing purposes) or to dispose of it.

Working with Documents from the Inside

In this scenario you embed application logic into the document itself to provide the user with an integrated experience within the document. In this design model, the UI will perform the following functions:

- **Reporting Data**
  You can implement some custom buttons or menu entries to retrieve data from some business process or database and then display it.
- **Gathering Data**
  You can implement some custom buttons or menu entries to collect data from the document and then invoke user or business processes.

Accessing Data Access Components from the UI

UI may need to render data that is readily available as queries exposed by data access components. Calling data access components from the UI may seem to contradict the layering concept, however, it can be used to adopt the perspective of your application as one homogenous service. In general, you should allow UI to directly invoke data access components when:

- You are willing to tightly couple UI semantics with data access methods and schemas. This tight coupling will require joint-maintenance of data access and UI components.
- Your physical deployment places data access and UI components together, allowing you to get data from your data access components in streaming formats (like `DataReader`) that can be bound directly to the output of ASP.NET pages for best performance. From an operational perspective, allowing direct access to data access logic components to take advantage of streaming capabilities means that you also need to install the appropriate drivers to enable data access components to talk to the database. For example, a data access component that talks to an Oracle database requires that **Oracle Client** be installed on the same machine where the data access component is installed.

Designing User Process Components

Note: Implementing UI with user process components is not trivial. Before committing to this approach, evaluate whether or not your application requires this level of abstraction and orchestration.

As noted in the definition of User Process Components,a user's interaction with the system may follow a predictable process. Rather than hard-coding this logic within the UI elements, you can use a UI process component to help synchronize and orchestrate these user actions. UI Process Components - or UI Managers as I prefer to call them- ensure that the process flow and its state management are encapsulated within a component that can be potentially reused by multiple UIs. For example, a `SearchManager` is an appropriate UI process component to synchronize and orchestrate (i.e., manage) product look-ups. Therefore, partitioning user interaction flow from the activities of rendering data and gathering data from the user can increase the application's maintainability and provide a clean design to which you can easily add seemingly complex features. Otherwise, an unstructured approach to implementing user interface logic may result in undesirable situations where if you need to add a specific UI to a given device (i.e., Windows Forms, or ASP.NET), you may need to redesign control logic and data flow.

User process components are typically implemented as .NET classes that expose methods that can be called from the UI, with each method encapsulating logic to perform a specific action in the user process. As expected then, the UI creates an instance of a user process component and uses it to transition through the various steps of the process.

Designing a user process component to be used from multiple UIs (Windows Forms, ASP.NET, etc.), results in a more complex implementation to avoid device-specific issues. Nonetheless, it can help distribute the UI development work between multiple teams, each using the same UI process component. You should also design process components with globalization in mind to allow for localization to be implemented in the user interface. For example, try to use culture-neutral formats and Unicode strings to make easier for the localized UI to call the user process component.

The following code snippet show a typical user process component for product look-up:

```
public class SearchManager
{
    /* Data members */


    /* Public interface */
    public void SetProduct( Product obProd )
    { ... }

    public void SetFilter( Filter obFilter )
    { ... }


    public ProductsCollection Search()
    { ... }

    public void ClearProducts()
    { ... }

}
```

The following topics are covered:

Advantages of user process components

Separating user interaction functionality into user interface and user process components provides the following advantages:

- **Persists long-running user-interaction state**
  This allows a user interaction to be paused and resumed, possibly even using a different UI. For example, the UI for shopping carts often allows users to store and retrieve their items at a later time.
- **Isolates long-running user-interaction state from business-related state**
  Because some user processes can be paused and resumed later, the intermediate state of the user process should be stored separately from the application's business data. Storing this intermediate state is often done by marking the user process as *serializable*. For example, a user could specify only part of the information required to place an order, and then resumes the checkout at a later time. The pending order data should be persisted (serialized) separately from data relating to completed orders, allowing you to perform business operations on completed data only. This approach avoids having to implement complex filters to avoid operation on incomplete data.
- **Allows the use of the same user-process by more than one UI**
  For example, a user process for processing shopping carts can be used by both Windows Forms and ASP.NET versions of the same application.
- **Handles concurrent user activities**
  Some applications may allow a user to perform multiple tasks at the same time by making more than one UI element available. With user processes, you can simplify state management of multiple ongoing processes by mapping each UI element (form, grid, etc.) to a particular instance of the user process component.

- **Synchronizes activities across multiple UI elements**
  If multiple UI elements (forms, grids, etc.) are used in a particular UI activity, it is important to keep their state and data synchronized. For example, it might be required that data be simultaneously displayed in tabular and chart form. Changes to data should be reflected in both forms of display. User process components help implement this kind of functionality by centralizing the state of related UI elements in a central location. You can further simplify synchronization across multiple UIs by using bindable data.

Design of user process components

The following sub-sections explain the steps involved in designing user process components:

User process identification

Before you can design a user process, you must first identify it. When reading the following points, it can be helpful to have the 'saving a shopping cart' user process in-mind:

- Identify the *business* (not user) process that the user process will help to accomplish.
- Identify the data needed by the business process. The user process must be able to submit this data to the business process.
- Identify state that you will have to maintain throughout the user activity to assist rendering and data capture in the UI.
- Design the visual flow of the user process and the way that each UI element receives or gives control flow.

User process functionality

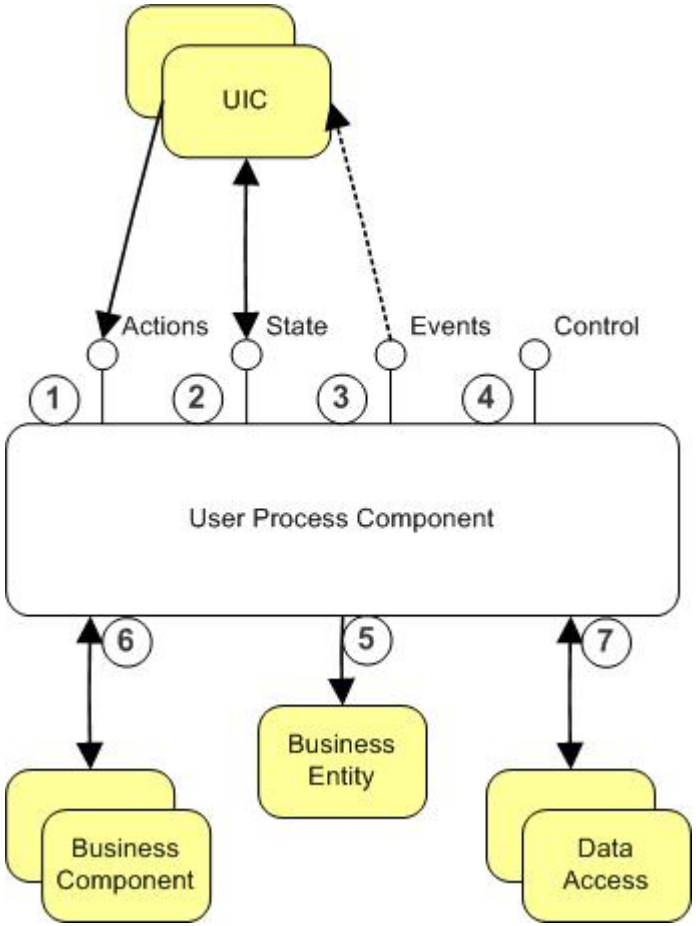
User process components do the following:

- Separate the user interaction flow from its implementation.
- UI elements need to keep a references to the current user process component. This reference should be kept in some member variable.
- Encapsulate how exceptions affect the user process flow.
- Keep track of the current state of the user interaction.
- Should not start or participate in transactions.
- Maintain internal business-related state, usually holding on to one or more business entities that are affected by the user interaction.
- May provide a *save and continue later* where a particular user interaction may be restarted in another session. User process state can be persisted in many places:
  - If the user process is to be continued from another machine or device, state can be persisted in the database.
  - If the user process is to be continued from a disconnected environment, state can be saved in the local hard drive.
- If the user process is running in a web farm, you will need to store state in a central location so that this state can be accessible from any server.
- Separate user processes are most needed in applications with a large number of dialog boxes, or in applications where the user process may benefit from customizations.

- User processes often store state in one or more of the following locations:
  - If the process is running in a connected fashion, store state in the database. In disconnected scenarios, store state in local XML files, isolated storage or in the local SQL Server 2000 Desktop engine (MSDE).
  - If the process is not long-running and does not need to be recovered in case of a problem, you should persist the state in memory.
  - For ASP.NET applications, store state in the Session object. If you are running in a Web farm, store state in a central state server or in the database.
- User processes should be serializable to help implement any persistence scheme.
- User processes should also include exception and error handling by propagating exceptions to the UI.

User process design

A user process should be designed to expose the following interfaces:



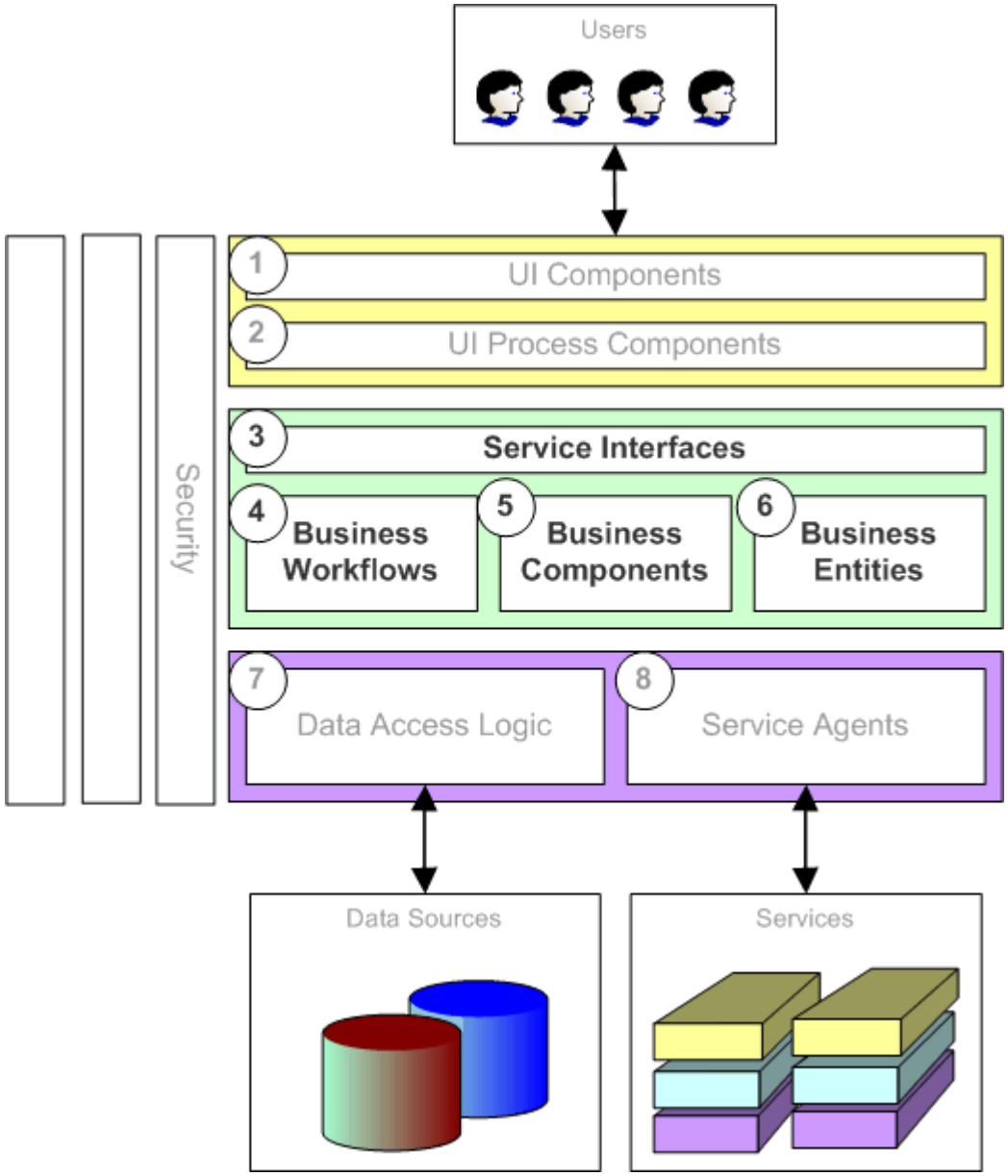
These interfaces are explained below:

1. **Actions**: These are methods that typically trigger some change in the state of the user process. Actions are implemented as methods such as `PlaceOrder`, `ProcessPaymentDetails`, and so on.
2. **State**: These are set/get properties that allow access to business-specific and business-agnostic state of the user process.
3. **Events**: Events that are fired on changes in the business-specific and business-agnostic state of the user process.
4. **Control**: Control functions to start/pause/restart/stop a particular user process. Control methods are often used to load required data for UI.

Designing Business Layers

Business functionality is the core of the application. An application performs a business process that consists of one or more tasks. In the simplest case, each task can be represented as a single method in a component, and this method can be called either synchronously or asynchronously. For example, you might have an application whose business process is selling books. Selling books obviously consists of many tasks such as capturing customer's order, verifying payment details, arranging delivery, and so on. A more complex business process has multiple tasks, but each task may require multiple steps with long running transactions. For these complex business processes, the application needs some way of orchestrating business tasks and storing state until the process is completed.

You can design the logic in the business layers to be used directly by the presentation components, or to be encapsulated as a service and be called through a service interface. Your business components may also make requests of external services, in which case you may need to implement service agents to manage conversation with those external services. The following figure and subsequent sections refer to individual components of the business layer:



- Business Components and Business Workflows
- Service Interfaces
- Business Entities

Business Components and Business Workflows

One of the first decisions in designing a business layer is to decide if a set of business components will be sufficient or if you need to orchestrate the business process performed by these business components.

When to implement the business process using a set of business component:

- You do not need to maintain state beyond the current business activity and the business functionality can be implemented as a single atomic transaction.
- You need to encapsulate functionality that can be used from other business processes.
- The business logic is computationally intensive or needs fine-grained control of data structures and APIs.
- You need to have a fine-grained control over data and flow logic.

When to implement the business process using business components and workflows:

- You need to manage a business process that involves multiple steps and long-running transactions.
- You need to expose a service interface that implements a business process enabling your applications to talk with other (external or internal) services.

For example, the business process of *placing an order* involves many steps and these steps must be performed in a specific sequence (collecting order item, calculating total cost, authorizing payment, arranging delivery, and so on). The appropriate design for this business process is to create business components encapsulating each step and then to orchestrate these steps using a business workflow.

Designing Business Components

Business components implement business rules and accept/return simple and complex data. Business components should expose functionality that is agnostic to data stores (databases, files, storage, etc.) and services needed to perform the work.

If a business process invokes other business processes in the context of an atomic transaction, all the invoked business processes must ensure their operations participate in the existing transaction, so that their operations can roll-back if the calling business logic aborts. This implies that is should be safe to retry an atomic transactions if it fails without worrying about making data inconsistent. *Think of a transaction boundary as a retry boundary*. Microsoft offers several technologies for managing transactions - Distributed Transaction Coordinator (DTC), COM Transaction Integrator (COMTI), and Host Integration Server.

If you cannot implement atomic transactions, you must provide compensating methods and processes. Note that a compensating action does not necessarily roll-back application data to the previous state, but rather restored the business data to a consistent state. For example, when canceling an order, you have to restore the product quantity for each ordered product, but you may also impose a fee for order cancellation.

The following list summarizes recommendation for designing business components:

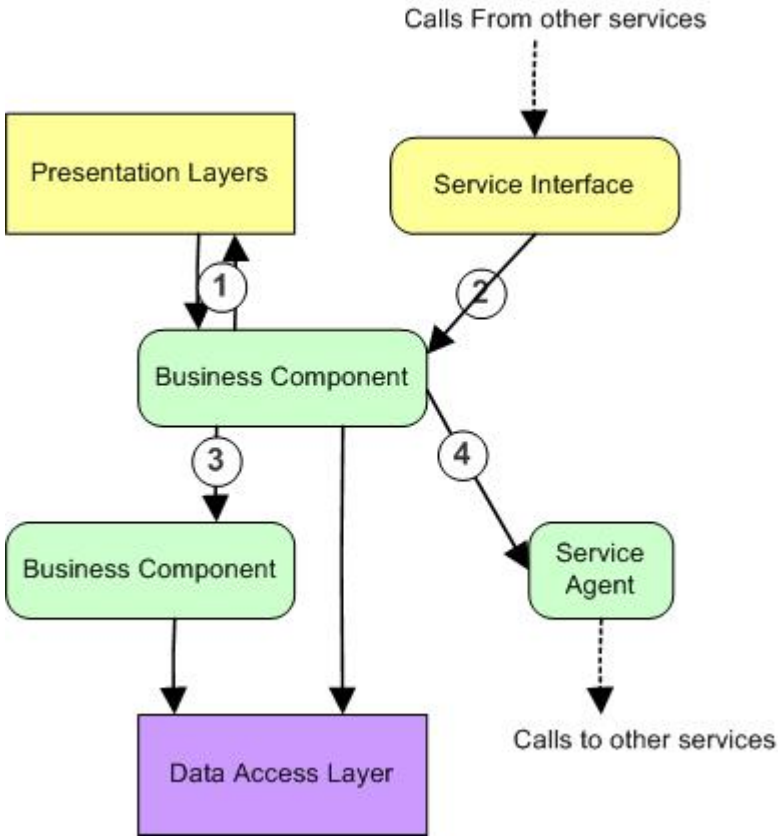
- Use message-based communication as much as possible.
- Ensure that processes exposed through service interfaces will not reach an inconsistent state if the same message is received twice.
- Choose transaction boundaries carefully so that retries and composition are possible.
- Business components must be able to run in the context of an service user - not necessarily impersonating a specific application user.
- Choose and keep a consistent data format for input parameters and return values (i.e, `DataSet`s, XML, and so on)
- Set transaction isolation levels appropriately.

Implementing Business Components

The following list summarizes recommendation for implementing business components. Business components:

- Will be called by the user process layer, service interfaces, and other business processes typically with some business data to operate on.
- Will invoke data access logic to retrieve and/or update application data. Business components can also invoke service agents and other business components.
- Are the root of transactions and must therefore, vote in transactions they participate in.
- Must validate input and output.
- May need to expose compensating operations for the business processes they provide.
- May call external services through service agents.
- May call other business components and initiate business workflows.
- May raise an exception to the caller.
- May use the features of Enterprise Services for initiating and voting on transactions. See Enterprise Services in MSDN for more details.

The following figure shows a typical business component interacting with components from surrounding layers, and with components from the same layer:



Patterns for Business Components

There are many commonly used patterns that can be applies when it comes to implementing business components:

Pipeline Pattern

A pipeline defines a collection of steps that execute in sequence to fulfill a business process. Each step may involve reading or writing data to confirm the pipeline state, and may or may not access an external service. When executing an asynchronous service as part of a step, the pipeline can wait until a response is returned or process to the next step in the pipeline if a response is not required in order to continue processing. The pipeline pattern can be used when:

- You can specify the sequence of a known set of steps.
- You do not need to wait for an asynchronous response from each step.
- You want all downstream components to be able to inspect and act on data coming from upstream components - but not vice versa.

The advantages of this pattern is that it is simplifies the actions of a business component into small and well-known steps, enforces sequential processing, and makes it easy to wrap in an atomic transaction. Disadvantages include inability to handle well conditional constructs, loops and other flow logic.

Event Pattern

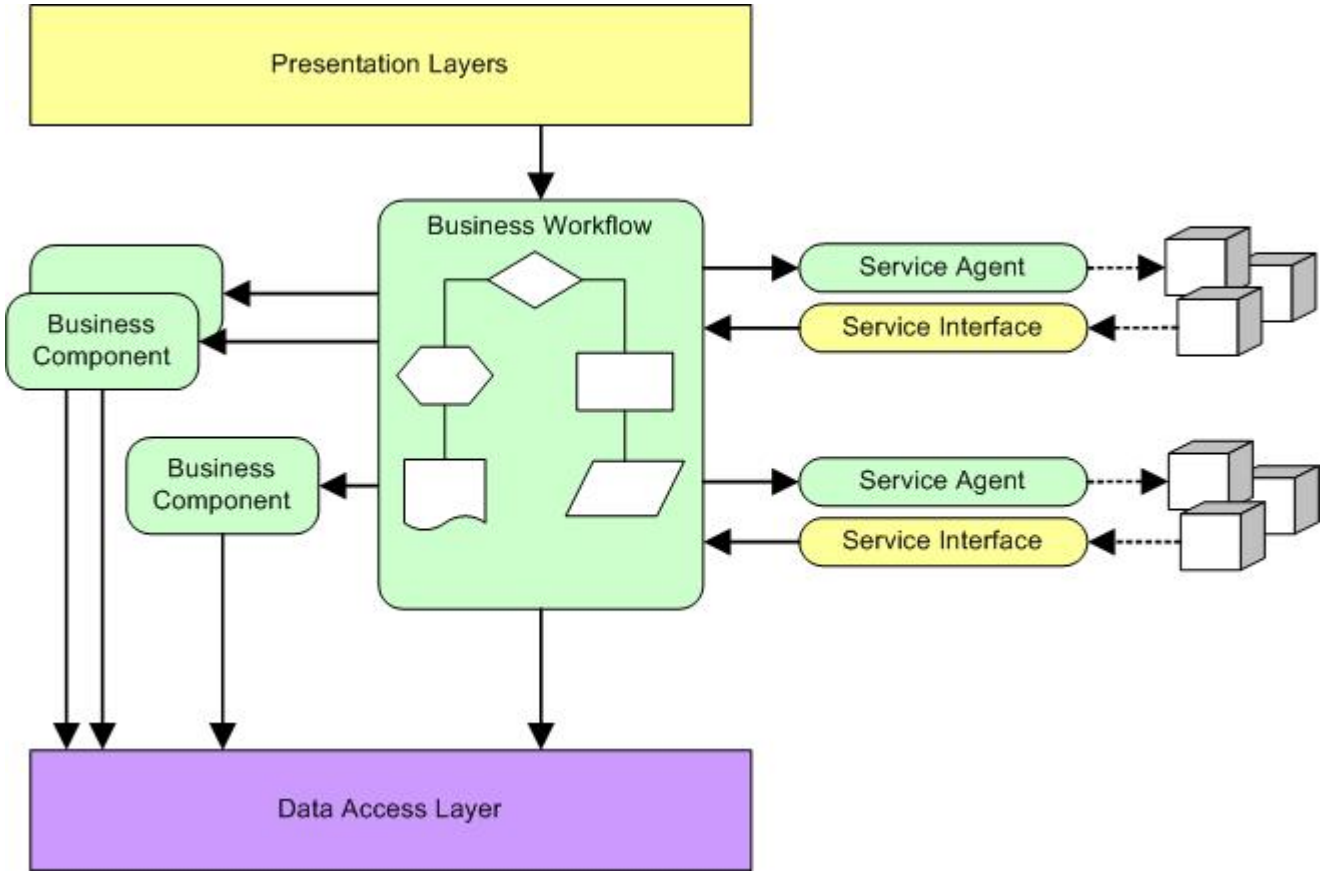
From a business component point of view, the event pattern is semantically the same as that in Windows Forms. Whereas UI elements in a Windows Forms application fire UI events (`OnMouseDown`) in response to user actions and event handlers handle those events, so are business events fired under particular business conditions and business components are written to handle those events. The event pattern is used when:

- You want to manage independent and isolated implementations of a specific function.
- Responses from one implementation do not affect the way other implementations work.
- All implementations are write-only or file-and-forget

The advantages of this pattern is that maintainability is improved by keeping unrelated business processes independent, encourages parallel processing, and is agnostic to whether implementations run synchronously or asynchronously because no response is required. Disadvantages include factors such as inability to build complex responses for the business function, and inability to use status/data of another component.

Business Workflows

The following diagram shows how an orchestrated business process interacts with other components from with the same layer and from different layers:



Note the following points:

- Business workflows can be invoked from other services or from the presentation layer.
- Business workflows can invoke other services through service agents
- Business workflows invoke business components.
- Business workflows invoke data access logic to perform data-related activities.

When designing business workflows you must consider long response times or no responses at all.

Service Interfaces

- Using Business Facades with Service Interfaces
- Transaction Management in Service Interfaces

If you are exposing business functionality as a service, you need to provide clients of this business process with an entry point by creating a *service interface*. A service interface is typically implemented as a facade that handles mapping and transformation services, and enforces a process and a policy for communication. A service interface implements a set of methods that can be called individually or in a specific sequence to form a conversation that implements a business process.

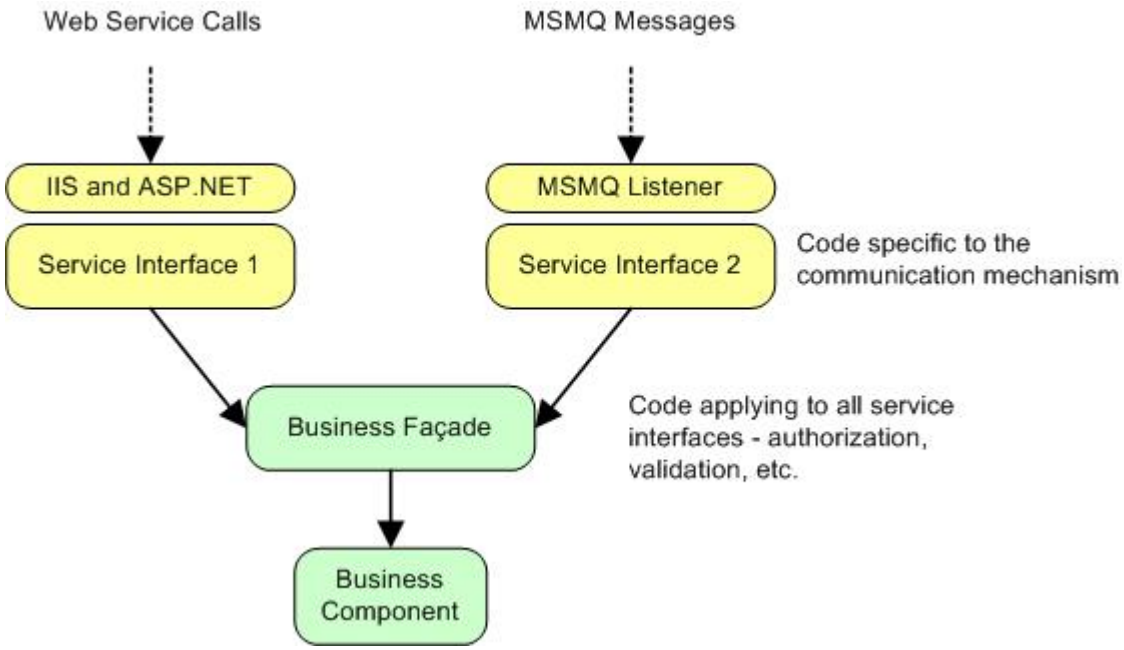
For example, a service interface for credit card payment may expose two methods - `AuthorizeCard` and `ProcessPayment` - and these two methods need to be called in sequence to actually accept payment from the user. From an implementation perspective, it is important to note that a service interface is actually another component (a C# `class` rather than an `interface`.)

Consider the following when designing service interfaces:

- Think of a service interface as a trust boundary for your application.
- Design the service interface is such a way that internal changes to implementation will not require changes to the service interface. This is especially useful if the service interface was exposed to the external consumers.
- The same business logic may need to be consumed in different ways by different clients, so you may need to publish multiple service interfaces for the same functionality.
- Different service interfaces may define different communication channels, message formats, authentication mechanisms, transactional capabilities, caching, mapping, and simple format/schema transformations. However, service interfaces should not implement business logic. For example, the same service interface may be implemented as two C# classes, one supporting communication through DCOM and another through .NET Remoting.
- Service interfaces can be implemented in many ways - for ASP.NET applications, a service interface is just another ASP.NET page, and for .NET application, a service interface is just another C# class. If the .NET client uses Message Queuing to send message, the service interface could be a Message Queue Trigger.
- Service interfaces should be designed with maximum interoperability with other platforms and services, relying whenever possible on industry standards for communication, security, and formats.

Using Business Facades with Service Interfaces

If you are planning on building a system that may be invoked through different mechanisms, then you should add a facade between the business logic and the service interface. For example, if you choose to build a Web Service, then most of the service interface logic will be in the Web Service itself (`.asmx` files). This facade provides extra maintainability because it can isolate changes - say in the communication mechanism - from the implementation of the business logic. The service interface then only deals with the specifics of the communication mechanism or channel (for example, examining SOAP headers for context information). The following figure illustrates:



In the above figure, an ASP.NET application running IIS receives an HTTP message and then invokes Service Interface 1 which inspects some SOAP header values and sets the correct principal object based on the authentication of the Web Service. Service Interface 1 then invokes the Business Facade component to validate, authorize, and audit the call. The facade then invokes the business component to perform the actual business work. At a later stage in the application lifecycle, it was determined that business components should also be available via MSMQ. A custom MSMQ listener is used to pick up messages and then invokes Service Interface 2 to extract some data off the message and set the correct principal object on the thread based on the message signature. Then the facade is used to invoke the business component as usual. The main point here is that *by factoring the code of the business facade outside of the service interface, the application was able to add a communication mechanism without having to change the business component*.

Transaction Management in Service Interfaces

The service interface will need to deal with channels that do not provide transactional capabilities (XML Web Service) and channels that do provide transactional capabilities (Message Queuing). It is very important that transaction boundaries can be designed so that operations can be retried in an error. To do so, make sure that all your resources are transactions and that root components ***require transactions*** while all other sub components ***require/support transactions***:

- **Transactional Messaging**
  The service interface starts the transactions first and then processes the message. If the transaction fails, the message is 'unreceived' and placed back in the message queue for a retry. For example, when using MSMQ you can define a message queue-and-receive operation as transactional to achieve this automatically.
- **Non-transactional Messaging**
  The service interface code must call the root of the transaction. In case of failure, you can design the service interface functionality to retry the operation or return to the caller an appropriate exception, or may be just set some data flag to represent failure.

Business Entities

Business entities exhibit the following behavior:

- Business entities contain snapshot data. They are effectively a local cache of data - data can only be guaranteed to be consistent if it is read in the context of an active transaction.
- Business entities store data values and expose them via properties, therefore, they essentially follow a stateful programming model.
- A business entity typically has a schema that is a denormalization of underlying schemas. In other words, it may represent data that has been aggregated from many sources. In general, you should not map one business entity to one database table
- Business entities should be serializable to persist the current state of the entity. For example, the application may need to store the business entity on desk if the application is working offline.
- Business entities do not initiate any transactions. Transactions are the responsibility of the application or business components using these business entities.

The following business entity topics are further discussed:

Representing Business Entities

When passing data around from one layer to another, or even within different components in the business layer, you can do so in many formats. These formats vary from data-centric like XML to more object-oriented like a Typed DataSet. Common formats for passing data in .NET applications are:

- XML
- DataSet
- Custom objects with data that map to fields and methods that perform data modifications using the data access layer.
- Typed DataSet
- DataReader

Generally, it is recommended that data-centric formats like DataSets be used to pass data between tiers. DataSets can then be used to hydrate custom business entities if you want to work with data in an object-oriented approach. In many cases, it will also be simpler to work with business data directly from the DataSet.

Guidelines	Advantages	Disadvantages
XML		
• Decide whether the XML document should contain a single business entity or a collection of business entities. • Use a namespace to uniquely identify the document. • Retrieve business entities in XML using SQL Server FOR XML clause, or a DataSet that is tranformed into XML, or by building the business entity XML from scratch	• Standards support. • Flexibility as XML can represent hierarchies and collections • Interoperability with external partners and services	• Type fidelity is not preserved, but you can use XSD for simple data typing. • Parsing XML (manually or via XSD) is slow. • Cannot automatically bind XML data to UI. You will need to write an XSLT transformation to transform the data into a .NET bindable type. • Cannot automatically sort XML. You may need to use XSLT or transform the data into a sortable .NET type like a DataView .

DataSet		
	• Very flexible. • Supports serialization. • Supports data binding to any UI element. • Interchangeable with XML • Availability of metadata. • Supports optimistic concurrency.	• High instantiation and marshalling costs. • Do not have the option of hiding information through private fields.
Custom Business Entity Component		
• Contains private fields to cache the entity's data locally. • Contains public properties to access the state of the entity. • Contains methods to perform localized processing. • Contains events to signal changes in the entity's internal state.	• Code readability via typed methods and properties. • Can contain methods to encapsulate simple business rules such as `IncreasePriceNPercent(int n)`. • Can perform simple validation tests. • Can hide information via private fields.	• A custom business entity represents a *single* business entity. Calling application must use a collection to hold multiple business entities. • You must implement serialization • You must implement your own mechanism for representing hierarchies and relationship of data in a business entity component. • You must implement `IComparable` to allow entity components to be held in a sortable container. • You must deploy assembly containing business entity to all physical tiers. • If database schema is modified, the custom business entity needs to be modified and redeployed.

In most cases you should work with data directly using XML documents or data structures provided by ADO.NET - `DataSet`, `DataTable`, etc. *This allows you to pass structured data between the layers of your application without having to write any custom code*. Business entities are custom components used to represent data and they are considered as part of the business layer. Business entities can be consumed by other business components or by presentation components.

Business Entity Consumers

So who consumes business entities? Consumers include:

- **UI components for rich clients**
  These UI components may bind to the data in business entities or to the data returned by any queries that the component may expose.
- **User process components**
  User process components may hold one or more business entity as part of its internal business-specific state.
- **Business components**
  A business components may pass a business entity as a parameter to a data access logic component.

Business Entity Interface Design

You only need to write custom business entity components if:

- You want to encapsulate all the details about working with a particular format, or
- You want to add behaviors to your data, or,
- You want to tighten control over what other application components can do with the data.
- You want to abstract internal data formats from the schema that the application is using.

Business entity components must expose the following functionality:

- Property accessors (get/set) for attributes of the entity.
- Collection accessors for sub-collections of related data. There are two ways to represent collections - using a .NET collection such as ArrayList or using a DataSet.
- Control functions to manage the entity, i.e., functions like `Load`, `Save`, `Validate`, etc.
- Methods to access metadata for the entity. This can be useful in improving maintainability of the user interface.
- Events to signal changes in the underlying data. Exposing events to UI allows the UI to refresh its data.
- Methods to perform business tasks or get date for complex queries. These methods may act on local data or on business components and processes.
- Methods and interfaces for data binding.

Designing Business Entities

Consider the following recommendation when designing business entities:

- Business entities should not update the data store each time any of its properties change. Instead, provide an `Update` method to propagate all local changes to the data store.
- Do not access data directly. Use data access logic components instead. In other words, business entities has no knowledge of Data Access Logic Components.
- Do not initiate any transactions. Transactions should only be initiated by business components or UI process components.
- Implement business entities by deriving them from a base class that provides boilerplate functionality.
- If the business entity contains any internal collections, use DataSets and XML documents rather than structs, arrays and so on.
- Implement a common set of interfaces across all business:
  - `IBusinessEntityControl`
    Control methods and properties such as `Load`, `Save`, `Delete`, `Update`, `IsDirty`, etc.
  - `IBusinessEntityMetadata`
    Methods to retrieve information about the underlying metadata such as `GetChildDataSetMetadata`, etc.
- Business entities should validate their encapsulated data through continuous and point-in-time validation rules.
- You may need to implement an implicit enforcement of relationships between different business entities based on data schema and business rules. For example, a business entity representing an `Order` may limit the maximum number of `Items` - another business entity -that it (`Order`) can hold.
- Make business entities serializable. You can either use XML serialization via the `XmlSerializer` class or formatted serialization via the `BinaryFormatter` / `SoapFormatter`.

Business Entity Example

The following represents a sample code of a business entity component called `CustomerBusinessEntity` that also exposes events as its state changes:

```
// Define a common event for all business entities
public class EntityEventArgs : EventArgs
{
    // Define event class members, like datetime, business entity type name
    // (obtained via reflection) and others
}
public delegate void EntityEventHandler (object sender, EntityEventArgs args);
```

```
/* A data access logic component called CustomerDALC will create the business entity and populate all its
fields. */
public class CustomerBusinessEntity
{
    /* Events fired by this class */
    public event EntityEventHandler BeforeChange;
    public event EntityEventHandler AfterChange;

    /* Data members - private fields to hold state of the business entity */
    private string     m_strFirstName   = "";
    private string     m_strLastName    = "";
    private string     m_strAddress     = "";
    private DateTime   m_dtDOB;
    private DataTable  m_dtCurrentOrders;

    /* Public properties to get/set state of object */
    public string FirtName
    {
        get { return m_strFirstName; }
        set
        {
            BeforeChanges( this, new EntityEvnetArgs();
            m_strFirstName = (string)value;
            AfterChanges( this, new EntityEvnetArgs();
        }
    }

    public string LastName
    {
        get { return m_strLastName; }
        set
        {
            BeforeChanges( this, new EntityEvnetArgs();
            m_strLastName = (string)value;
            AfterChanges( this, new EntityEvnetArgs();
        }
    }

    public string Address
    {
```

```
        get { return m_strAddress; }
        set
        {
            BeforeChanges( this, new EntityEvnetArgs();
            m_strAddress = (string)value;
            AfterChanges( this, new EntityEvnetArgs();

        }
    }

    public DateTime DOB
    {
        get { return m_dtDOB; }
        set
        {
            BeforeChanges( this, new EntityEvnetArgs();
            m_dtDOB = (DateTime)value;
            AfterChanges( this, new EntityEvnetArgs();
        }
    }

    public DataTable CurrentOrders
    {
        get { return m_dtCurrentOrders; }
        set
        {
            BeforeChanges( this, new EntityEvnetArgs();
            m_dtCurrentOrders = (DataTable)value;
            AfterChanges( this, new EntityEvnetArgs();
        }
    }

    /* Public methods to perform some localized processing */
    public bool IsCustomerInAgeGroup( int nAgeGroup )
    {
        // Determine age group based on customer's DOB
    }
}
```

Exception Handling in Business Entities

Business entity components should propagate exceptions to their direct callers. Business entity components may also raise exceptions if they are performing any validations or if, for example, the caller supplies missing data. The following example illustrates:

```
public class CustomerEntity
{
```

```
    public void Update( string strFirstName, string strLastName )
    {
        try
        {
            // Check that names are valid
            if (strFirstName == null) || (strLastName == null)
                throw new InvalidArgumentException ("Customer names are not valid" );


            if (strFirstName.Length == 0) || (strLastName.Length == 0)
                throw new InvalidArgumentException ("Customer names are empty" );

            // Call customer data access logic component to update customer
            ...

        }
        catch( Exception ex )
        {
            throw;        // Propagate exception to caller
        }
        finally
        {

        }
    }
}
```

Designing Data Layers

The following sections discuss the choice of data stores, design of data access components, and choices available for representing data

- Data Stores
- Data Access Logic Components
- Data Access Helper Components
- Integrating with Services

Data Stores

Common types of data stores include:

- **Relational Databases**
  Relational databases provide high-volume, transactional, high-performance data management with security, operations, and data transformation services.
- **Messaging Databases**
  Messaging databases like Exchange Server are useful if the application is groupware-, workgroup-, or messaging-centric and you do not want to rely on

other data stores that may need to be managed separately.

- **File Systems**
  You may decide to store data in your own files in the file system.
- **Others**
  There are many other data stores like XML databases, object databases, data warehouses, and so on. Consult MSDN for more information.

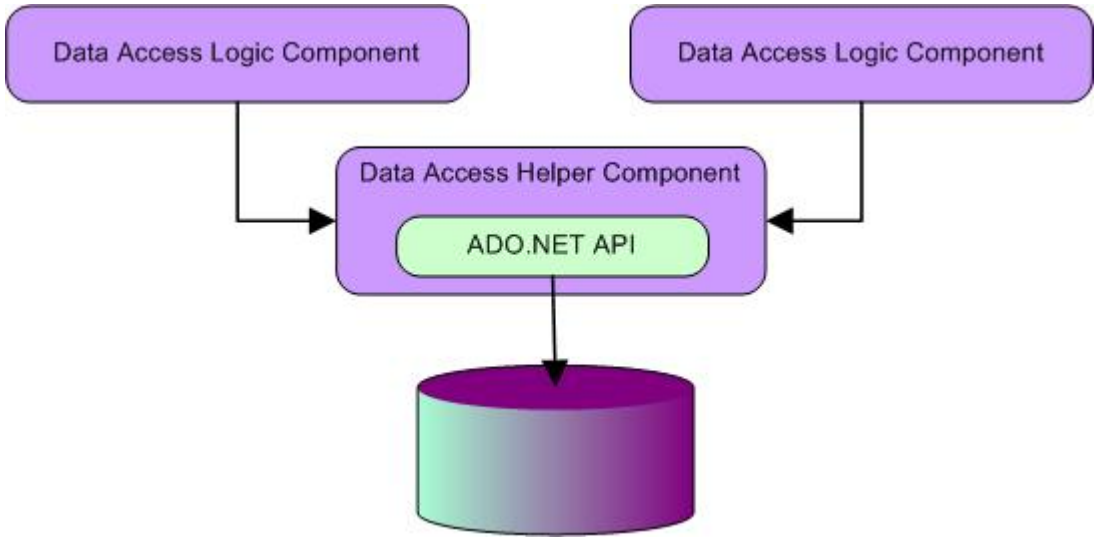
Data Access Logic Components

Data Access Logic Components abstract the semantics of the underlying data store (i.e., SQL Server) and the underlying data access technology (i.e., ADO.NET) and provide a simple programmatic interface for retrieving and operating on data. They are not to be confused with Data Access Helper Components which centralize generic data access functionality.

Data access logic components usually implement a *stateless design pattern* that separates the business processing from data access logic. *Each data access logic component typically provides CRUD operations (Create, Read, Update, and Delete) relating to a specific business entity*. They also include paging functionality when retrieving large records. When an application contains multiple data access logic components, it becomes important to centralize data access logic in a data access helper component that manages database connections, transactions, command execution, and so on.

The difference between data access logic component and data access helper components is that the former provides logic to *access specific business data*, while the latter centralizes data access API (i.e., ADO.NET) and data connection configuration. Note: See the Data Access Application Block provided by Microsoft, This block can be used as a generic data helper component.

The following figure shows the relation between data access logic components and data access helper components:



The following code show a partial skeletal outline of a data access logic component for accessing Order data:

```
public class ProductData
{
    // Constructors
```

```
    public ProductData                          { // Acquire connection string from a secure location }

    // Public interface
    public DataSet GetProducts()                { // Acquire all products }
    public DataRow GetProduct( long lID }       { // Acquire a specific products }
    public UpdataProduct( DataRow drProduct )   { // Update a specific product in the database }
}
```

The following sections provide more details on data access logic components:

Data access logic component functionality

Data access logic components should have the following functionality:

- Perform simple mappings and transformations of input and output arguments.
- Access data from one data source only. This improves maintainability by moving all data aggregation functionality to the business component where data can be aggregated according to the specific business process being performed.
- [Optional] Use a custom utility component to manage and encapsulate locking mechanisms..
- [Optional] Use a custom utility component to implement data caching strategy.
- [Optional] Implement dynamic data routing functionality for very large scale systems that provide scalability by distributing data across multiple data servers.

Data access logic components should *not* have the following functionality:

- Invoke other data access logic components.
- Initiate heterogeneous transactions.
- Maintain state across method calls.

To support a wide range of business processes and applications, you can pass and return data to/fromData Access Logic Components in several different formats - arrays, XML, DataSets or custom business entity component.

Mapping Relational Data to Business Entities

One of the first steps in writing business entities is to decide how to map tables to business entities. For example, typical operations in a hypothetical retails would be to get/update information about a customer (including address), get a list of orders for a customer, get/update information about a product and so on. To fulfill these requirements, you can come up with three logical business entities - Customer, Product, and Order. For each business entity, a separate Data Access Logic Component (DALC) will be defined as follows:

- CustomerDALC: Retrieves and modifies data in the `[Customer]` and `[Address]` tables.
- OrderDALC: Retrieves and modifies data in the `[Order]` and `[OrderDetails]` tables.
- ProductDALC: Retrieves and modifies data in the `[Product]` and `[ProductOffers]` tables.

To map relational data to business entities, observe the following guidelines:

- Do not define a separate business entity for each table. Take the time to analyze and understand the logical entities of the application.
- Do not define separate business entities to represent many-to-many relations. These methods can be exposed through methods in the Data Access Logic Components. For example, in the preceding example, `[OrderDetails]` is not mapped to a separate business entity, instead the OrderDALC encapsulates the `[OrderDetails]` table to achieve the many-to-many relations between `[Order]` and `[Product]` tables.
- If you have methods that retrieve a particular type of business entity, place those methods in the Data Access Logic Component for that type. For example, when retrieving all *orders* for customer, implement this function in the OrderDALC.
- DALCs typically access data from a single source. If aggregation from multiple sources is required, define a separate DALC to access each data source. These DALCs are then called from higher-level business components that can perform the aggregation. Reasons for this approach are:
  - Centralizing transaction management in the business process components.
  - By separating access to each data source, returned data can be either stand alone or part of aggregated data.

Data access logic component interface design

Data access logic component often expose their functionality to the following consumers:

- **Business components and workflows**
  These consumers often consume disconnected business documents and/or scalars in a stateless manner.
- **User Interface components**
  These consumers often consume disconnected business documents for data rendering operations.

Data access logic component often expose the following functionality to their consumers:

- CRUD functions to manage business entities.
- Queries that may involve getting data from different data sources for read-only purposes.
- Relating meta-data relating to entity schema, query parameters, or result set schemas.
- Paging for user interfaces that require subsets of data (usually when scrolling through a large result set.)

Data access logic component design

When designing data access logic components, consider the following design recommendations:

- Return only the data you need - improves performance and enhances scalability.
- Use stored procedures to abstract data access from the underlying data schema.
- Relay on the database to do data-intensive work.
- Expose functionality that is common across all data access logic components in a base class or in a separately defined interface.
- Design consistent interfaces for different clients:
  - ASP.NET-based clients will benefit from using `DataReader`s rather than `DataSet`s. `DataReader`s are best for read-only forward-only operations in which processing for each row is fast.

  - Windows Forms-based clients will benefit from using `DataSet`s rather than `DataReader`s. `DataSet`s are best for disconnected scenarios.
- Let data access logic components expose metadata for the data and operations it deals with. If a method already returns one of the ADO.NET data structures (`DataTable` or `DataSet` for example), then metadata should already be available.
- Do not necessarily build a data access logic components per table. Data access logic components should be designed to expose a slightly higher level of abstraction that is consumable from your business components.
- If you store encrypted data, data access logic components should decrypt data before sending it back to consumers.
- Use data access helper components to centralize generic data access functionality.

Data Access Logic Component Example

The following represents a sample code of a data access logic component called `CustomerDALC` for the `CustomerBusinessEntity` class shown previously:

```
public class CustomerDALC
{
    /* Data members */
    private string m_strConnectionString = "";

    /* Constructors */
    public CustomerDALC()
    {
        // Acquire the connection string from a secure location
    }

    /* Data access methods */
    public CustomerBusinessEntity GetCustomer( int nID )
    {
        // Use data access helper components to retrieve data
    }

    public int CreateCustomer( string strName, string strAddress, string DateTime dtDOB )
    {
        // Use data access helper components to create a new customer entry in the database
        // and return customer ID
    }

    public void DeleteCustomer( int nCustomerID )
    {
        // Use data access helper components to delete customer from the database
    }

    public void UpdateCustomer( DataSet dsCustomer )
    {
        // Use data access helper components to update customer in the database
    }
}
```

```
// Code in some business component
CustomerDALC             dalc = new CustomerDALC();
CustomerBusinessEntity   be = dalc.GetCustomer( 123 );
...
```

Exception Handling in Data Access Logic Components

Data Access Logic Components should propagate exceptions back to direct callers. The following example illustrates:

```
public class CustomerDALC
{
    public void LoadCustomers()
    {
        try
        {
            // Load all customers from database
        }
        catch ( Exception ex )
        {
            threw new DataAccessException ( ex.Message );
        }
        finally
        {
            // Cleanup
        }
    }

    public UpdateCustomers() { ... }
    public DeleteCustomer() { ... }
    public AddCustomers ( CustomerBusinessEntities collBE ) { ... }
}
```

Data Access Helper Components

Centralize generic data access functionality in a data access helper component can help produce cleaner code and a more manageable design. Consider data access helper component as a generic caller-side facade to the data store. They are typically agnostics to the application logic being performed.

If you are designing your application to be agnostic to the data source being used (for example, to be able to access Oracle, SQL Server and so on), you would have two data access helper components, one that use the ADO.NET provider classes for Oracle (`OracleCommand`, ...), and another helper component that uses ADO.NET provider for SQL Server (`SqlCommand`, ...). You can then use the Factory pattern to create the appropriate component while providing a uniform interface to both components.

In general, the goals of a data access helper components are:

- Abstract data access API programming from the data-related business logic encapsulated by the data access logic components.
- Isolate the following:
  - Connection management semantics.
  - Data source location (through connection string management).
  - Data source authentication.
  - Transaction enlistment
  - ADO.NET versioning dependencies from data access logic components.
- Centralize data access logic for easier maintenance
- Provide a single point of interception for data access monitoring and testing.

The Data Access Application Block for .NET should be used as a model on how to design data access helper components for .NET

Integrating with Services

If your business process talks to external services, then you will need to handle the semantics of calling each service that you want to communicate with. Specifically:

1. You will need to use the correct communication API and perform any necessary translation between the data formats used by the service and those used by your business process.
2. If calling an external service also involves long-running conversations, then you will also need to keep intermediate state while waiting for a response.

Think of service agents as data access logic components for services rather than for data stores. You can also think of service agents as *proxies* to other services. The goals of using a service agent are therefore:

- Encapsulate access to services.
- Isolate business process implementation from the service implementation in terms of data formats or schema changes.
- Provide input and output data formats that are compatible with the business components.
- [Optional] Perform basic validation of data exchanged with the service.
- [Optional] Cache data for common queries.
- [Optional] Authorize access to the service.
- [Optional] Encrypt outgoing/decrypt incoming data.
- [Optional] Provide monitoring information.

In certain scenarios, service agents may receive asynchronous data from business components. Service agents may also send asynchronous data to services. When messages are exchanged asynchronously, you need to keep track of the conversation state -which conversation does a set of message belong to. There are two options:

- Use business data in the message to identify the conversation. For example, the `OrderID` can be used to correlate a set of messages to a specific conversation.
- Provide an infrastructure component that provides GUIDs for specific conversations and attach them to messages.

NEXT CHAPTER

This chapter described general recommendation for designing different kinds of component commonly found in distributed systems. The next chapter discusses the effect of various policies (security, communication, etc.) on the design of the application.