Chapter 12: Designing Business Components

01/13/2010 • 13 minutes to read

For more details of the topics covered in this guide, see Contents of the Guide.

Contents

Overview

Designing business components is an important task; if you fail to design your business components correctly, the result is likely to be code that is difficult to maintain or extend. There are several types of business components you may use when designing and implementing an application. Examples of these components include business logic components, business entities, business process or workflow components, and utility or helper components. This chapter starts with an overview of the different types of business components you will find in most application designs, with the primary focus on business logic components. It shows how different aspects of your application design, transactional requirements, and processing rules affect the design you choose. Once you have an understanding of the requirements, the final step focuses on design patterns that support those requirements.

Step 1 – Identify Business Components Your Application Will Use

Within the business layer, there are different types of components that you may need to create or use to handle business logic. The goal of this step is to understand how you identify these components, and discover which components your application requires. The following guidelines will help you to decide which types of components you require:

- Consider using **business logic components** to encapsulate the business logic and state of your application. Business logic is application logic that is concerned with the implementation of the business rules and behavior of your application, and with maintaining overall consistency through processes such as data validation. Business logic components should be designed to be easily testable and independent of the presentation and the data access layers of your application.
- Consider using **business entities** as part of a domain modeling approach to encapsulate business logic and state into components that represent the real world business entities from your business domain, such as products and orders, which your application has to work with. For more information about business entities, see Chapter 13 "Designing Business Entities."
- Consider using **business workflow components** if your application must support multistep processes executed in a specific order; uses business rules that require the interaction between multiple business logic components; or you want change the behavior of the application by updating the workflow as the application evolves or requirements change. Also consider using business workflow components if your application must implement dynamic behavior based on business rules. In this case, consider storing the rules in a rules engine. Consider using Windows Workflow Foundation to implement your workflow components. Alternatively, consider an integration server environment such as BizTalk Server if your application must process multiple steps that depend on external resources, or has a process that must be executed as a long-running transaction. For more information about workflow components, see Chapter 14 "Designing Workflow Components." For more information about integration services, see Appendix D "Integration Technology Matrix."

Step 2 – Make Key Decisions for Business Components

The overall design and type of application you are creating plays a role in the business components that it will use to handle requests. For example, business components for a Web application usually deal with message-based requests, while a Windows Forms application will typically use event-based requests to interact directly with business components. In addition, there are other factors to consider when working with different application types. Some of these factors are common across types, while some are unique to an application type. Key decisions you must make with business components include:

- **Location**. Will your business components be located on the client, on an application server, or on both? Consider locating some or all business components on the client if you have a stand-alone rich client or a Rich Internet Application (RIA), if you want to improve performance, or if you are using a domain model design for business entities. Consider locating some or all business components on an application server if you must support multiple client types with common business logic, if business components require access to resources not accessible from the client, or for security reasons to protect the components within a managed and secured server environment.
- **Coupling**. How will your presentation components interact with your business components? Should you use tight coupling where the presentation components have direct knowledge of the business components, or loose coupling where an abstraction is used to hide details of the business components? For simplicity, if you have a rich client application or RIA with both sets of components located on the client, you may consider tight coupling between presentation and business components. However, loose coupling between presentation and business components will improve testability and flexibility. If you have a rich client application or RIA with business components located on an application server or Web server, design the service interface to enable their interaction to be as loosely coupled as possible.
- **Interaction**. If your business components are located on the same tier as your presentation components, consider using component-based interactions through events and methods, which maximizes performance. However, consider implementing a service interface and using message-based interactions between the presentation layer and business components if the business components are located on a separate physical tier from your Web server; if you are designing a Web application with loose coupling between the presentation and business layers; or if you have a rich

client or RIA application. If you have a rich client application or RIA that is occasionally connected to an application server or Web server, you must carefully design the service interface to allow your client to resynchronize when connected.

When you use message-based interaction, consider how you will manage duplicate requests and guarantee message delivery. **Idempotency** (the ability to ignore duplicate requests) is important if you are designing a service application, a message-based application that uses a messaging system such as Microsoft Message Queuing, or a Web application where a long running process may cause the user to attempt the same action multiple times. **Guaranteed delivery** is important if you are designing a message-based application that uses a messaging system such as Microsoft Message Queuing, a service that uses message routers between the client and service, or a service that supports fire and forget operations where the client sends a message without waiting for a response. Also, consider that cached messages, which may be stored awaiting processing, can become stale.

Step 3 – Choose Appropriate Transaction Support

Business components are responsible for coordinating and managing any transactions that may be required in your business layer. However, the first step is to determine if transaction support is required. Transactions are used to ensure that a series of actions executed against one or more resource managers, such as databases or message queues, is completed as a single unit independent of other transactions. If any single action in a series fails, all other actions must be rolled back to ensure the system is left in a consistent state. For example, you might have an operation that updates three different tables using multiple business logic components. If one of those updates fails, but two succeed, the data source will be in an inconsistent state; which means that you now have invalid data on which other operations may depend. The following options are available for implementing transactions:

- **System.Transactions** uses business logic components to initiate and manage transactions. Introduced in version 2.0 of the .NET Framework along with the Lightweight Transaction Manager (LTM), it deals with nondurable resource managers or one durable resource manager. This approach requires explicit programming against the **TransactionScope** type, and can escalate the transaction scope and delegate to a Distributed Transaction Coordinator

(DTC) if more than one durable resource manager is enlisted in the transaction. Consider using **System.Transactions** if you are developing a new application that requires transaction support, and you have transactions that span multiple nondurable resource managers.

- **WCF Transactions** were introduced in version 3.0 of the .NET Framework and are built on top of the **System.Transactions** functionality. They provide a declarative approach to transaction management implemented using a range of attributes and properties, such as **TransactionScopeRequired**, **TransactionAutoComplete**, and **TransactionFlow**. Consider using WCF Transactions if you must support transactions when interacting with WCF services. However, consider whether a declarative transaction definition is a requirement, rather than using code to manage transactions.
- **ADO.NET Transactions**, available since version 1.0 of the .NET Framework, require the use of business logic components to initiate and manage transactions. They use an explicit programming model where developers are required to manage non-distributed transactions in code. Consider using ADO.NET Transactions if you are extending an application that already uses ADO.NET Transactions, or if you are using ADO.NET providers for database access and the transactions are limited to a single resource. ADO.NET 2.0 and later additionally support distributed transactions using the System.Transactions features described earlier in this list.
- **Database** transactions are used for transaction management that can be incorporated into stored procedures, which may also simplify your business process design. If transactions are initiated by business logic components, the database transaction will be enlisted in the transaction created by the business component. Consider using database transactions if you are developing stored procedures that encapsulate all of the changes that must be managed by a transaction, or you have multiple applications that use the same stored procedure and transaction requirements can be encapsulated within that stored procedures.

Be aware that systems that use distributed transactions can increase coupling between sub-systems. Transactions that include remote systems are likely to affect performance due to increased network traffic. Transactions are expensive and should execute quickly, otherwise resources could be locked for excessive amounts of time which can lead to time outs, or deadlocks.

Allow only highly trusted services to participate in transactions because external services can to lock your internal resources through participation in the transaction. If you are calling services to perform business processes, avoid creating atomic transactions that span these calls unless you cannot avoid this.

Step 4 – Identify How Business Rules Are Handled

Managing business rules can be one of the more challenging aspects of application design. Generally, you should always keep business rules within the business layer. However, exactly where in the business layer should they go? You can use business logic or workflow components, a business rule engine, or use a domain model design with rules encapsulated in the model. Consider the following options for handling business rules:

- **Business Logic Components** can be used to handle simple rules or very complex rules, depending on the design pattern used to implement the business logic components. Consider using business logic components for tasks or document-oriented operations in Web applications or within services, if you are not implementing a domain model design for business entities, or you are using an external source that contains the business rules.
- **Workflow Components** are used when you want to decouple business rules from business entities, or the business entities you are using do not support the encapsulation of business rules, or when you have to encapsulate business logic that coordinates the interaction between multiple business entities.
- **Business Rules Engines** provides a way for non developers to establish and modify rules, but they also add complexity and overhead to an application and should only be used where appropriate. In other words, you would only use a rules engine if you have rules that must be adjusted based on different factors associated with the application. Consider using a business rules engine if you have volatile business rules that must be modified on a regular basis; to support customization and offer flexibility; or you want to allow business users to manage and update rules. Ensure that only the rules users should be able to modify are exposed, and that unauthorized users cannot modify rules that are critical to correct business logic behavior.
- **Domain Model Design** can be used to encapsulate business rules within business entities. However, a domain model design can be difficult to get right, and tends to focus on a specific viewpoint or context. Consider encapsulating rules in a domain model if you have a rich client application or RIA where parts of the business logic are deployed on the client and the domain model entities are initialized and persisted in memory, or you have a domain model that can be maintained within the session state associated with Web or service applications. If you locate parts of the domain model

on the client, you should mirror the model on the server to apply rules and behavior, and to ensure security and maintainability.

Step 5 – Identify Patterns That Fit the Requirements

Behavioral patterns are based on observing the behavior of a system in action and looking for repeatable processes. With business components, the patterns you might use are usually behavioral design patterns. In other words, patterns that are focused on the behavior of an application at the design level. Much work has been done identifying and defining patterns that occur in different types of applications and in different layers of an application design. It is not feasible to try to learn all of the patterns that have been defined; however, you should have a good understanding of different pattern types and be able to examine your scenario to identify behavior that could be expressed as a pattern. The following table describes patterns that are commonly used with business components.

Pattern	Recommendation
Adapter	Allow classes that have incompatible interfaces to work together, allowing developers to implement sets of polymorphic classes that provide alternative implementations for an existing class.
Command	Recommended for rich client applications with menus, toolbars, and keyboard shortcut interactions that are used to execute the same commands against different components. Can also be used with the Supervising Presenter pattern to implement commands.
Chain of Responsibility	Chain request handlers together so that each handler can examine the request and either handle it or pass it on to the next handler in the chain. An alternative to "if, then, else" statements, with the ability to handle complex business rules.
Decorator	Extend the behavior of an object at run time to add or modify operations that will be performed when executing a request. Requires a common interface that will be implemented by decorator classes, which can be chained together to handle complex business rules.

Pattern	Recommendation
Dependency Injection	Create and populate members (fields and properties) of objects using a separate class, which usually creates these dependencies at run time based on configuration files. Configuration files define containers that specify the mapping or registrations of object types. Application code can also define the mapping or registration of objects. Provides a flexible approach to modifying behavior and implementing complex business rule.
Façade	Provide coarse-grained operations that unify the results from multiple business logic components. Typically implemented as a remote façade for message-based interfaces into the business layer, and used to provide loose coupling between presentation and business layers.
Factory	Create object instances without specifying the concrete type. Requires objects that implement a common interface or extend a common base class.
Transaction Script	Recommended for basic CRUD operations with minimal business rules. Transaction script components also initiate transactions, which means all operations performed by the component should represent an atomic unit of work. With this pattern, the business logic components interact with other business components and data components to complete the operation.

Although this list represents many of the common patterns you might use with business components, there are many other patterns associated with business components. The main goal when choosing a pattern is to ensure that it fits the scenario and does not add more complexity than necessary.

Additional Resources

For more information, see the following resources:

- For more information on business component design, see "*Application Architecture for .NET: Designing Applications and Services*" http://msdn.microsoft.com/en-us/library/ms954595.aspx .
- For more information on performance in business layers and components, see the following resources:

- "*Architecture and Design Review of a .NET Application for Performance and Scalability*" at http://msdn.microsoft.com/en-us/library/ms998544.aspx.
- "*Design Guidelines for Application Performance*" at http://msdn.microsoft.com/en-us/library/ms998541.aspx.

- For more information on implementing transactions in business components, see the following resources:
  - "*Introducing System.Transactions in the .NET Framework 2.0*" at http://msdn.microsoft.com/en-us/library/ms973865.aspx.
  - "*Transactions in WCF*" at http://msdn.microsoft.com/en-us/library/ms730266.aspx.
  - "*Transaction Processing in .NET 3.5*" at http://msdn.microsoft.com/en-us/library/w97s6fw4.aspx.
- For more information on implementing workflow in business components, see the following resources:
  - "*Introduction to the Windows Workflow Foundation Rules Engine*" at http://msdn.microsoft.com/en-us/library/aa480193.aspx.
  - "*Windows Workflow Foundation*" at http://msdn.microsoft.com/en-us/library/ms735967.aspx.