Chapter 2: Key Principles of Software Architecture

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

Contents

Overview

In this chapter, you will learn about the key design principles and guidelines for software architecture. Software architecture is often described as the organization or structure of a system, where the *system* represents a collection of components that accomplish a specific function or set of functions. In other words, architecture is focused on organizing components to support specific functionality. This organization of functionality is often referred to as grouping components into "areas of concern." Figure 1 illustrates common application architecture with components grouped by different areas of concern.

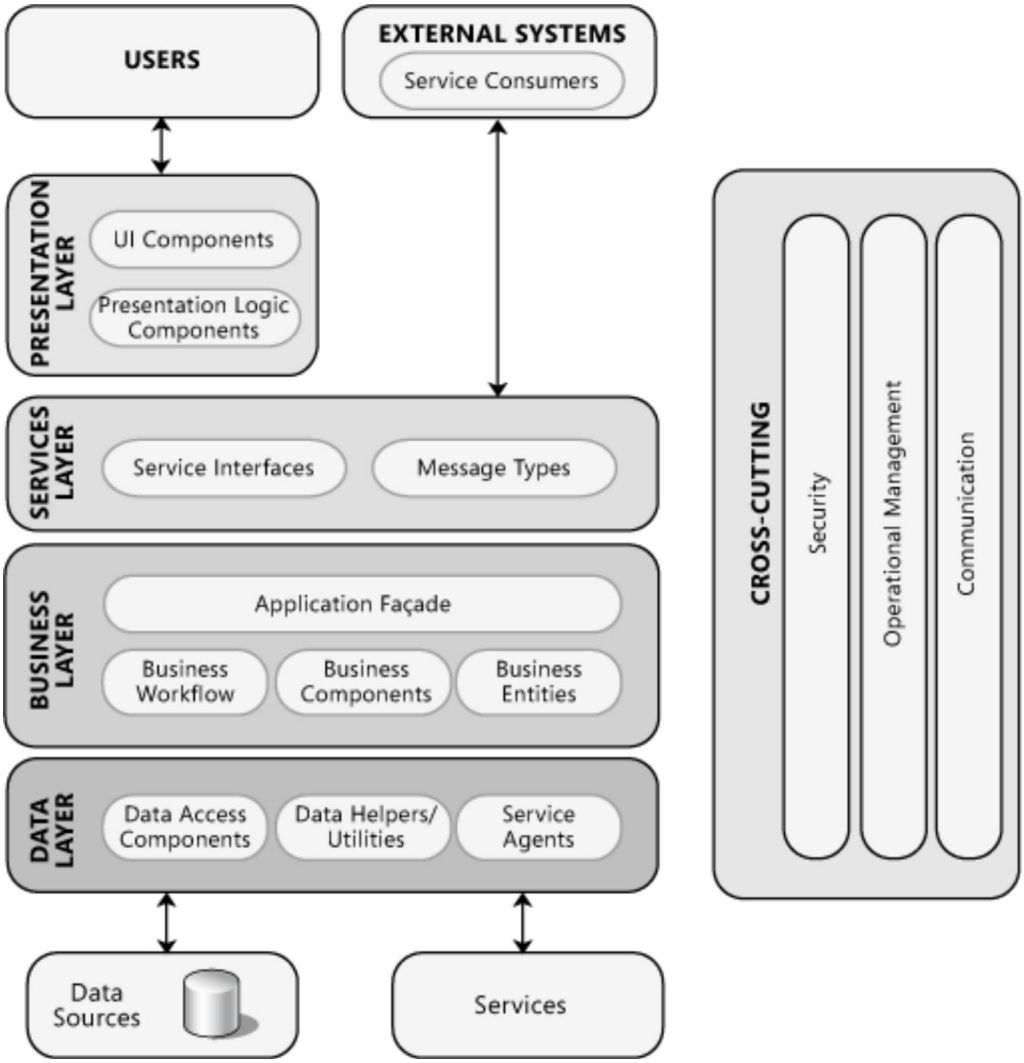

Figure 1

Common application architecture

In addition to the grouping of components, other areas of concern focus on interaction between the components and how different components work together. The guidelines in this chapter examine different areas of concern that you should consider when designing the architecture of your application.

Key Design Principles

When getting started with your design, keep in mind the key principles that will help you to create an architecture that adheres to proven principles, minimizes costs and maintenance requirements, and promotes usability and extendibility. The key principles are:

- **Separation of concerns**. Divide your application into distinct features with as little overlap in functionality as possible. The important factor is minimization of interaction points to achieve high cohesion and low coupling. However, separating functionality at the wrong boundaries can result in high coupling and complexity between features even though the contained functionality within a feature does not significantly overlap.
- **Single Responsibility principle**. Each component or module should be responsible for only a specific feature or functionality, or aggregation of cohesive functionality.
- **Principle of Least Knowledge** (also known as the Law of Demeter or LoD). A component or object should not know about internal details of other components or objects.
- **Don't repeat yourself (DRY)**. You should only need to specify intent in one place. For example, in terms of application design, specific functionality should be implemented in only one component; the functionality should not be duplicated in any other component.
- **Minimize upfront design.** Only design what is necessary. In some cases, you may require upfront comprehensive design and testing if the cost of development or a failure in the design is very high. In other cases, especially for agile development, you can avoid big design upfront (BDUF). If your application requirements are unclear, or if there is a possibility of the design evolving over time, avoid making a large design effort prematurely. This principle is sometimes known as YAGNI ("You ain't gonna need it").

When designing an application or system, the goal of a software architect is to minimize the complexity by separating the design into different areas of concern. For example, the user interface (UI), business processing, and data access all represent different areas of concern. Within each area, the components you design should focus on that specific area and should not mix code from other areas of concern. For example, UI processing components should not include code that directly accesses a data source, but instead should use either business components or data access components to retrieve data.

However, you must also make a cost/value determination on the investment you make for an application. In some cases, you may need to simplify the structure to allow, for example, UI data binding to a result set. In general, try to consider the functional boundaries from a business viewpoint as well. The following high level guidelines will help you to consider the wide range of factors that can affect the ease of designing, implementing, deploying, testing, and maintaining your application.

Design Practices

- **Keep design patterns consistent within each layer**. Within a logical layer, where possible, the design of components should be consistent for a particular operation. For example, if you choose to use the Table Data Gateway pattern to create an object that acts as a gateway to tables or views in a database, you should not include another pattern such as Repository, which uses a different paradigm for accessing data and initializing business entities. However, you may need to use different patterns for tasks in a layer that have a large variation in requirements, such as an application that contains business transaction and reporting functionality.
- **Do not duplicate functionality within an application**. There should be only one component providing a specific functionality—this functionality should not be duplicated in any other component. This makes your components cohesive and makes it easier to optimize the components if a specific feature or functionality changes. Duplication of functionality within an application can make it difficult to implement changes, decrease clarity, and introduce potential inconsistencies.
- **Prefer composition to inheritance**. Wherever possible, use composition over inheritance when reusing

functionality because inheritance increases the dependency between parent and child classes, thereby limiting the reuse of child classes. This also reduces the inheritance hierarchies, which can become very difficult to deal with.

- **Establish a coding style and naming convention for development**. Check to see if the organization has established coding style and naming standards. If not, you should establish common standards. This provides a consistent model that makes it easier for team members to review code they did not write, which leads to better maintainability.
- **Maintain system quality using automated QA techniques during development**. Use unit testing and other automated Quality Analysis techniques, such as dependency analysis and static code analysis, during development. Define clear behavioral and performance metrics for components and sub-systems, and use automated QA tools during the build process to ensure that local design or implementation decisions do not adversely affect the overall system quality.
- **Consider the operation of your application**. Determine what metrics and operational data are required by the IT infrastructure to ensure the efficient deployment and operation of your application. Designing your application's components and sub-systems with a clear understanding of their individual operational requirements will significantly ease overall deployment and operation. Use automated QA tools during development to ensure that the correct operational data is provided by your application's components and sub-systems.

Application Layers

- **Separate the areas of concern**. Break your application into distinct features that overlap in functionality as little as possible. The main benefit of this approach is that a feature or functionality can be optimized independently of other features or functionality. In addition, if one feature fails, it will not cause other features to fail as well, and they can run independently of one another. This approach also helps to make the application easier to understand and design, and facilitates management of complex interdependent systems.
- **Be explicit about how layers communicate with each other**. Allowing every layer in an application to communicate with or have dependencies upon all of the other layers will result in a solution that is more challenging to understand and manage. Make explicit decisions about the dependencies between layers and the data flow between them.
- **Use abstraction to implement loose coupling between layers**. This can be accomplished by defining interface components such as a façade with well known inputs and outputs that translate requests into a format understood by components within the layer. In addition, you can also use Interface types or abstract base classes to define a common interface or shared abstraction (dependency inversion) that must be implemented by interface components.
- **Do not mix different types of components in the same logical layer**. Start by identifying different areas of concern, and then group components associated with each area of concern into logical layers. For example, the UI layer should not contain business processing components, but instead should contain components used to handle user input and process user requests.
- **Keep the data format consistent within a layer or component**. Mixing data formats will make the application more difficult to implement, extend, and maintain. Every time you need to convert data from one format to another, you are required to implement translation code to perform the operation and incur a processing overhead.

Components, Modules, and Functions

- **A component or an object should not rely on internal details of other components or objects**. Each component or object should call a method of another object or component, and that method should have information about how to process the request and, if appropriate, how to route it to appropriate subcomponents or other components. This helps to create an application that is more maintainable and adaptable.
- **Do not overload the functionality of a component**. For example, a UI processing component should not contain data access code or attempt to provide additional functionality. Overloaded components often have many functions and properties providing business functionality mixed with crosscutting functionality such as logging and exception handling. The result is a design that is very error prone and difficult to maintain. Applying the single responsibility and separation of concerns principles will help you to avoid this.
- **Understand how components will communicate with each other**. This requires an understanding of the deployment scenarios your application must support. You must determine if all components will run within the same process, or if communication across physical or process boundaries must be supported—perhaps by implementing message-based interfaces.

- **Keep crosscutting code abstracted from the application business logic as far as possible**. Crosscutting code refers to code related to security, communications, or operational management such as logging and instrumentation. Mixing the code that implements these functions with the business logic can lead to a design that is difficult to extend and maintain. Changes to the crosscutting code require touching all of the business logic code that is mixed with the crosscutting code. Consider using frameworks and techniques (such as aspect oriented programming) that can help to manage crosscutting concerns.
- **Define a clear contract for components**. Components, modules, and functions should define a contract or interface specification that describes their usage and behavior clearly. The contract should describe how other components can access the internal functionality of the component, module, or function; and the behavior of that functionality in terms of pre-conditions, post-conditions, side effects, exceptions, performance characteristics, and other factors.

Key Design Considerations

This guide describes the major decisions that you must make, and which help to ensure that you consider all of the important factors as you begin and then iteratively develop your architecture design. The major decisions, briefly described in the following sections, are:

- Determine the Application Type
- Determine the Deployment Strategy
- Determine the Appropriate Technologies
- Determine the Quality Attributes
- Determine the Crosscutting Concerns

For a more detailed description of the design process, see Chapter 4 "A Technique for Architecture and Design."

Determine the Application Type

Choosing the appropriate application type is the key part of the process of designing an application. Your choice is governed by your specific requirements and infrastructure limitations. Many applications must support multiple types of client, and may make use of more than one of the basic archetypes. This guide covers the following basic application types:

- Applications designed for mobile devices.
- Rich client applications designed to run primarily on a client PC.
- Rich Internet applications designed to be deployed from the Internet, which support rich UI and media scenarios.
- Service applications designed to support communication between loosely coupled components.
- Web applications designed to run primarily on the server in fully connected scenarios.

In addition, it provides information and guidelines for some more specialist application types. These include the following:

- Hosted and cloud-based applications and services.
- Office Business Applications (OBAs) that integrate Microsoft Office and Microsoft server technologies.
- SharePoint Line of Business (LOB) applications that provide portal style access to business information and functions.

For more information about application archetypes, see Chapter 20 "Choosing an Application Type."

Determine the Deployment Strategy

Your application may be deployed in a variety of environments, each with its own specific set of constraints such as physical separation of components across different servers, a limitation on networking protocols, firewall and router configurations, and more. Several common deployment patterns exist, which describe the benefits and considerations for a range of distributed and non-distributed scenarios. You must balance the requirements of the application with the

appropriate patterns that the hardware can support, and the constraints that the environment exerts on your deployment options. These factors will influence your architecture design.

For more information about deployment issues, see Chapter 19 "Physical Tiers and Deployment."

Determine the Appropriate Technologies

When choosing technologies for your application, the key factors to consider are the type of application you are developing and your preferred options for application deployment topology and architectural styles. Your choice of technologies will also be governed by organization policies, infrastructure limitations, resource skills, and so on. You must compare the capabilities of the technologies you choose against your application requirements, taking into account all of these factors before making decisions.

For more information about technologies available on the Microsoft platform, see Appendix A "The Microsoft Application Platform."

Determine the Quality Attributes

Quality attributes—such as security, performance, and usability—can be used to focus your thinking on the critical problems that your design should solve. Depending on your requirements, you might need to consider every quality attribute covered in this guide, or you might only need to consider a subset. For example, every application design must consider security and performance, but not every design needs to consider interoperability or scalability. Understand your requirements and deployment scenarios first so that you know which quality attributes are important for your design. Keep in mind that quality attributes may conflict; for example, security often requires a tradeoff against performance or usability.

When designing to accommodate quality attributes, consider the following guidelines:

- Quality attributes are system properties that are separate from the functionality of the system.
- From a technical perspective, implementing quality attributes can differentiate a good system from a bad one.
- There are two types of quality attributes: those that are measured at run time, and those that can only be estimated through inspection.
- Analyze the tradeoffs between quality attributes.

Questions you should ask when considering quality attributes include:

- What are the key quality attributes required for your application? Identify them as part of the design process.
- What are the key requirements for addressing these attributes? Are they actually quantifiable?
- What are the acceptance criteria that will indicate that you have met the requirements?

For more information about quality attributes, see Chapter 16 "Quality Attributes."

Determine the Crosscutting Concerns

Crosscutting concerns represent key areas of your design that are not related to a specific layer in your application. For example, you should consider implementing centralized or common solutions for the following:

- A logging mechanism that allows each layer to log to a common store, or log to separate stores in such a way that the results can be correlated afterwards.
- A mechanism for authentication and authorization that passes identities across multiple layers to permit granting access to resources.
- An exception management framework that will work within each layer, and across the layers as exceptions are propagated to the system boundaries.
- A communication approach that you can use to communicate between the layers.
- A common caching infrastructure that allows you to cache data in the presentation layer, the business layer, and the data access layer.

The following list describes some of the key crosscutting concerns that you must consider when architecting your applications:

- **Instrumentation and logging**. Instrument all of the business-critical and system-critical events, and log sufficient details to recreate events in your system without including sensitive information.
- **Authentication**. Determine how to authenticate your users and pass authenticated identities across the layers.
- **Authorization**. Ensure proper authorization with appropriate granularity within each layer, and across trust boundaries.
- **Exception management**. Catch exceptions at functional, logical, and physical boundaries; and avoid revealing sensitive information to end users.
- **Communication**. Choose appropriate protocols, minimize calls across the network, and protect sensitive data passing over the network.
- **Caching**. Identify what should be cached, and where to cache, to improve your application's performance and responsiveness. Ensure that you consider Web farm and application farm issues when designing caching.

For more information about crosscutting concerns, see Chapter 17 "Crosscutting Concerns."