Chapter 3: Architectural Patterns and Styles

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

Contents

Overview

This chapter describes and discusses high level patterns and principles commonly used for applications today. These are often referred to as the *architectural styles*, and include patterns such as client/server, layered architecture, component-based architecture, message bus architecture, and service-oriented architecture (SOA). For each style, you will find an overview, key principles, major benefits, and information that will help you choose the appropriate architectural styles for your application. It is important to understand that the styles describe different aspects of applications. For example, some architectural styles describe deployment patterns, some describe structure and design issues, and others describe communication factors. Therefore, a typical application will usually use a combination of more than one of the styles described in this chapter.

What Is An Architectural Style?

An architectural style, sometimes called an architectural pattern, is a set of principles—a coarse grained pattern that provides an abstract framework for a family of systems. An architectural style improves partitioning and promotes design reuse by providing solutions to frequently recurring problems. You can think of architecture styles and patterns as sets of principles that shape an application. Garlan and Shaw define an architectural style as:

"&#2026a; family of systems in terms of a pattern of structural organization. More specifically, an architectural style determines the vocabulary of components and connectors that can be used in instances of that style, together with a set of constraints on how they can be combined. These can include topological constraints on architectural descriptions (e.g., no cycles). Other constraints—say, having to do with execution semantics—might also be part of the style definition."

[David Garlan and Mary Shaw, January 1994, CMU-CS-94-166, see "*An Introduction to Software Architecture*" athttp://www.cs.cmu.edu/afs/cs/project/able/ftp/intro_softarch/intro_softarch.pdf]
An understanding of architectural styles provides several benefits. The most important benefit is that they provide a common language. They also provide opportunities for conversations that are technology agnostic. This facilitates a higher level of conversation that is inclusive of patterns and principles, without getting into specifics. For example, by using architecture styles, you can talk about client/server versus *n*-tier. Architectural styles can be organized by their key focus area. The following table lists the major areas of focus and the corresponding architectural styles.

Category	Architecture styles
Communication	Service-Oriented Architecture (SOA), Message Bus
Deployment	Client/Server, N-Tier, 3-Tier
Domain	Domain Driven Design
Structure	Component-Based, Object-Oriented, Layered Architecture

Summary of Key Architectural Styles

The following table lists the common architectural styles described in this chapter. It also contains a brief description of each style. Later sections of this chapter contain more details of each style, as well as guidance to help you choose the appropriate ones for your application.

Architecture style	Description
Client/Server	Segregates the system into two applications, where the client makes requests to the server. In many cases, the server is a database with application logic represented as stored procedures.
Component-Based Architecture	Decomposes application design into reusable functional or logical components that expose well-defined communication interfaces.
Domain Driven Design	An object-oriented architectural style focused on modeling a business domain and defining business objects based on entities within the business domain.
Layered Architecture	Partitions the concerns of the application into stacked groups (layers).
Message Bus	An architecture style that prescribes use of a software system that can receive and send messages using one or more

	communication channels, so that applications can interact without needing to know specific details about each other.
N-Tier / 3-Tier	Segregates functionality into separate segments in much the same way as the layered style, but with each segment being a tier located on a physically separate computer.
Object-Oriented	A design paradigm based on division of responsibilities for an application or system into individual reusable and self-sufficient objects, each containing the data and the behavior relevant to the object.
Service-Oriented Architecture (SOA)	Refers to applications that expose and consume functionality as a service using contracts and messages.

Combining Architectural Styles

The architecture of a software system is almost never limited to a single architectural style, but is often a combination of architectural styles that make up the complete system. For example, you might have a SOA design composed of services developed using a layered architecture approach and an object-oriented architecture style.

A combination of architecture styles is also useful if you are building a public facing Web application, where you can achieve effective separation of concerns by using the layered architecture style. This will separate your presentation logic from your business logic and your data access logic. Your organization's security requirements might force you to deploy the application using either the 3-tier deployment approach, or a deployment of more than three tiers. The presentation tier may be deployed to the perimeter network, which sits between an organization's internal network and an external network. On your presentation tier, you may decide to use a separated presentation pattern (a type of layered design style), such as Model-View-Controller (MVC), for your interaction model. You might also choose a SOA architecture style, and implement message-based communication, between your Web server and application server.

If you are building a desktop application, you may have a client that sends requests to a program on the server. In this case, you might deploy the client and server using the client/server architecture style, and use the component-based architecture style to decompose the design further into independent components that expose the appropriate communication interfaces. Using the object-oriented design approach for these components will improve reuse, testability, and flexibility.

Many factors will influence the architectural styles you choose. These factors include the capacity of your organization for design and implementation; the capabilities and experience of your developers; and your infrastructure and organizational constraints. The following sections will help you to determine the appropriate styles for your applications.

Client/Server Architectural Style

The client/server architectural style describes distributed systems that involve a separate client and server system, and a connecting network. The simplest form of client/server system involves a server application that is accessed directly by multiple clients, referred to as a 2-Tier architectural style.

Historically, client/server architecture indicated a graphical desktop UI application that communicated with a database server containing much of the business logic in the form of stored procedures, or with a

dedicated file server. More generally, however, the client/server architectural style describes the relationship between a client and one or more servers, where the client initiates one or more requests (perhaps using a graphical UI), waits for replies, and processes the replies on receipt. The server typically authorizes the user and then carries out the processing required to generate the result. The server may send responses using a range of protocols and data formats to communicate information to the client. Today, some examples of the client/server architectural style include Web browser—based programs running on the Internet or an intranet; Microsoft Windows® operating system—based applications that access networked data services; applications that access remote data stores (such as e-mail readers, FTP clients, and database query tools); and tools and utilities that manipulate remote systems (such as system management tools and network monitoring tools).
Other variations on the client/server style include:

- **Client-Queue-Client systems**. This approach allows clients to communicate with other clients through a server-based queue. Clients can read data from and send data to a server that acts simply as a queue to store the data. This allows clients to distribute and synchronize files and information. This is sometimes known as a *passive queue* architecture.
- **Peer-to-Peer (P2P) applications**. Developed from the Client-Queue-Client style, the P2P style allows the client and server to swap their roles in order to distribute and synchronize files and information across multiple clients. It extends the client/server style through multiple responses to requests, shared data, resource discovery, and resilience to removal of peers.
- **Application servers**. A specialized architectural style where the server hosts and executes applications and services that a thin client accesses through a browser or specialized client installed software. An example is a client executing an application that runs on the server through a framework such as Terminal Services.

The main benefits of the client/server architectural style are:

- **Higher security**. All data is stored on the server, which generally offers a greater control of security than client machines.
- **Centralized data access**. Because data is stored only on the server, access and updates to the data are far easier to administer than in other architectural styles.
- **Ease of maintenance**. Roles and responsibilities of a computing system are distributed among several servers that are known to each other through a network. This ensures that a client remains unaware and unaffected by a server repair, upgrade, or relocation.

Consider the client/server architectural style if your application is server based and will support many clients, you are creating Web-based applications exposed through a Web browser, you are implementing business processes that will be used by people throughout the organization, or you are creating services for other applications to consume. The client/server architectural style is also suitable, like many networked styles, when you want to centralize data storage, backup, and management functions, or when your application must support different client types and different devices.
However, the traditional 2-Tier client/server architectural style has numerous disadvantages, including the tendency for application data and business logic to be closely combined on the server, which can negatively impact system extensibility and scalability, and its dependence on a central server, which can negatively impact system reliability. To address these issues, the client-server architectural style has evolved into the more general 3-Tier (or N-Tier) architectural style, described below, which overcomes some of the disadvantages inherent in the 2-Tier client-server architecture and provides additional benefits.

Component-Based Architectural Style

Component-based architecture describes a software engineering approach to system design and development. It focuses on the decomposition of the design into individual functional or logical components that expose well-defined communication interfaces containing methods, events, and properties. This provides a higher level of abstraction than object-oriented design principles, and does not focus on issues such as communication protocols and shared state.
The key principle of the component-based style is the use of components that are:

- **Reusable**. Components are usually designed to be reused in different scenarios in different applications. However, some components may be designed for a specific task.
- **Replaceable**. Components may be readily substituted with other similar components.
- **Not context specific**. Components are designed to operate in different environments and contexts. Specific information, such as state data, should be passed to the component instead of being included in or accessed by the component.
- **Extensible**. A component can be extended from existing components to provide new behavior.
- **Encapsulated**. Components expose interfaces that allow the caller to use its functionality, and do not reveal details of the internal processes or any internal variables or state.
- **Independent**. Components are designed to have minimal dependencies on other components. Therefore components can be deployed into any appropriate environment without affecting other components or systems.

Common types of components used in applications include user interface components such as grids and buttons (often referred to as *controls*), and helper and utility components that expose a specific subset of functions used in other components. Other common types of components are those that are resource intensive, not frequently accessed, and must be activated using the just-in-time (JIT) approach (common in remoting or distributed component scenarios); and queued components whose method calls may be executed asynchronously using message queuing and store and forward.
Components depend upon a mechanism within the platform that provides an environment in which they can execute, often referred to as *component architecture*. Examples are the component object model (COM) and the distributed component object model (DCOM) in Windows; and Common Object Request Broker Architecture (CORBA) and Enterprise JavaBeans (EJB) on other platforms. Component architectures manage the mechanics of locating components and their interfaces, passing messages or commands between components, and—in some cases—maintaining state.
However, the term component is often used in the more basic sense of *a constituent part, element, or ingredient*. The Microsoft .NET Framework provides support for building applications using such a component based approach. For example, this guide discusses business and data components, which are commonly code classes compiled into .NET Framework assemblies. They execute under the control of the .NET Framework runtime, and there may be more than one such component in each assembly.
The following are the main benefits of the component-based architectural style:

- **Ease of deployment**. As new compatible versions become available, you can replace existing versions with no impact on the other components or the system as a whole.
- **Reduced cost**. The use of third-party components allows you to spread the cost of development and maintenance.
- **Ease of development**. Components implement well-known interfaces to provide defined functionality, allowing development without impacting other parts of the system.

- **Reusable**. The use of reusable components means that they can be used to spread the development and maintenance cost across several applications or systems.
- **Mitigation of technical complexity**. Components mitigate complexity through the use of a component container and its services. Example component services include component activation, lifetime management, method queuing, eventing, and transactions.

Design patterns such as the Dependency Injection pattern or the Service Locator pattern can be used to manage dependencies between components, and promote loose coupling and reuse. Such patterns are often used to build composite applications that combine and reuse components across multiple applications.
Consider the component-based architectural style if you already have suitable components or can obtain suitable components from third-party suppliers; your application will predominantly execute procedural-style functions, perhaps with little or no data input; or you want to be able to combine components written in different code languages. Also, consider this style if you want to create a pluggable or composite architecture that allows you to easily replace and update individual components.

Domain Driven Design Architectural Style

Domain Driven Design (DDD) is an object-oriented approach to designing software based on the business domain, its elements and behaviors, and the relationships between them. It aims to enable software systems that are a realization of the underlying business domain by defining a domain model expressed in the language of business domain experts. The domain model can be viewed as a framework from which solutions can then be rationalized.
To apply Domain Driven Design, you must have a good understanding of the business domain you want to model, or be skilled in acquiring such business knowledge. The development team will often work with business domain experts to model the domain. Architects, developers, and subject matter experts have diverse backgrounds, and in many environments will use different languages to describe their goals, designs and requirements. However, within Domain Driven Design, the whole team agrees to only use a single language that is focused on the business domain, and which excludes any technical jargon.
As the core of the software is the domain model, which is a direct projection of this shared language, it allows the team to quickly find gaps in the software by analyzing the language around it. The creation of a common language is not merely an exercise in accepting information from the domain experts and applying it. Quite often, communication problems within development teams are due not only to misunderstanding the language of the domain, but also due to the fact that the domain's language is itself ambiguous. The Domain Driven Design process holds the goal not only of implementing the language being used, but also improving and refining the language of the domain. This in turn benefits the software being built, since the model is a direct projection of the domain language.
In order to help maintain the model as a pure and helpful language construct, you must typically implement a great deal of isolation and encapsulation within the domain model. Consequently, a system based on Domain Driven Design can come at a relatively high cost. While Domain Driven Design provides many technical benefits, such as maintainability, it should be applied only to complex domains where the model and the linguistic processes provide clear benefits in the communication of complex information, and in the formulation of a common understanding of the domain.
The following are the main benefits of the Domain Driven Design style:

- **Communication**. All parties within a development team can use the domain model and the entities it defines to communicate business knowledge and requirements using a common business domain language, without requiring technical jargon.

- **Extensible**. The domain model is often modular and flexible, making it easy to update and extend as conditions and requirements change.
- **Testable**. The domain model objects are loosely coupled and cohesive, allowing them to be more easily tested.

Consider DDD if you have a complex domain and you wish to improve communication and understanding within your development team, or where you must express the design of an application in a common language that all stakeholders can understand. DDD can also be an ideal approach if you have large and complex enterprise data scenarios that are difficult to manage using other techniques.
For a summary of domain driven design techniques, see "*Domain Driven Design Quickly*" at http://www.infoq.com/minibooks/domain-driven-design-quickly. Alternatively, see "*Domain-Driven Design: Tackling Complexity in the Heart of Software*" by Eric Evans (Addison-Wesley, ISBN: 0-321-12521-5) and "*Applying Domain-Driven Design and Patterns*" by Jimmy Nilsson (Addison-Wesley, ISBN: 0-321-26820-2).

Layered Architectural Style

Layered architecture focuses on the grouping of related functionality within an application into distinct layers that are stacked vertically on top of each other. Functionality within each layer is related by a common role or responsibility. Communication between layers is explicit and loosely coupled. Layering your application appropriately helps to support a strong separation of concerns that, in turn, supports flexibility and maintainability.
The layered architectural style has been described as an *inverted pyramid of reuse* where each layer aggregates the responsibilities and abstractions of the layer directly beneath it. With strict layering, components in one layer can interact only with components in the same layer or with components from the layer directly below it. More relaxed layering allows components in a layer to interact with components in the same layer or with components in any lower layer.
The layers of an application may reside on the same physical computer (the same tier) or may be distributed over separate computers (*n*-tier), and the components in each layer communicate with components in other layers through well-defined interfaces. For example, a typical Web application design consists of a presentation layer (functionality related to the UI), a business layer (business rules processing), and a data layer (functionality related to data access, often almost entirely implemented using high-level data access frameworks). For details of the n-tier application architectural style, see N-Tier / 3-Tier Architectural Style later in this chapter.
Common principles for designs that use the layered architectural style include:

- **Abstraction**. Layered architecture abstracts the view of the system as whole while providing enough detail to understand the roles and responsibilities of individual layers and the relationship between them.
- **Encapsulation**. No assumptions need to be made about data types, methods and properties, or implementation during design, as these features are not exposed at layer boundaries.
- **Clearly defined functional layers**. The separation between functionality in each layer is clear. Upper layers such as the presentation layer send commands to lower layers, such as the business and data layers, and may react to events in these layers, allowing data to flow both up and down between the layers.
- **High cohesion**. Well-defined responsibility boundaries for each layer, and ensuring that each layer contains functionality directly related to the tasks of that layer, will help to maximize cohesion within the layer.

- **Reusable**. Lower layers have no dependencies on higher layers, potentially allowing them to be reusable in other scenarios.
- **Loose coupling**. Communication between layers is based on abstraction and events to provide loose coupling between layers.

Examples of layered applications include line-of-business (LOB) applications such as accounting and customer-management systems; enterprise Web-based applications and Web sites, and enterprise desktop or smart clients with centralized application servers for business logic.
A number of design patterns support the layered architectural style. For example, **Separated Presentation** patterns encompass a range of patterns that the handling of the user's interactions from the UI, the presentation and business logic, and the application data with which the user works. Separated Presentation allows graphical designers to create a UI while developers generate the code to drive it. Dividing the functionality into separate roles in this way provides increased opportunities to test the behavior of individual roles. The following are the key principles of the Separated Presentation patterns:

- **Separation of concerns**. Separated Presentation patterns divide UI processing concerns into distinct roles; for example, MVC has three roles: the Model, the View, and the Controller. The Model represents data (perhaps a domain model that includes business rules); the View represents the UI; and the Controller handles requests, manipulates the model, and performs other operations.
- **Event-based notification**. The Observer pattern is commonly used to provide notifications to the View when data managed by the Model changes.
- **Delegated event handling**. The controller handles events triggered from the UI controls in the View.

Other examples of Separated Presentation patterns are the Passive View pattern and the Supervising Presenter (or Supervising Controller) pattern.
The main benefits of the layered architectural style, and the use of a Separated Presentation pattern, are:

- **Abstraction**. Layers allow changes to be made at the abstract level. You can increase or decrease the level of abstraction you use in each layer of the hierarchical stack.
- **Isolation**. Allows you to isolate technology upgrades to individual layers in order to reduce risk and minimize impact on the overall system.
- **Manageability**. Separation of core concerns helps to identify dependencies, and organizes the code into more manageable sections.
- **Performance**. Distributing the layers over multiple physical tiers can improve scalability, fault tolerance, and performance.
- **Reusability**. Roles promote reusability. For example, in MVC, the Controller can often be reused with other compatible Views in order to provide a role specific or a user-customized view on to the same data and functionality.
- **Testability**. Increased testability arises from having well-defined layer interfaces, as well as the ability to switch between different implementations of the layer interfaces. Separated Presentation patterns allow you to build mock objects that mimic the behavior of concrete objects such as the Model, Controller, or View during testing.

Consider the layered architectural style if you have existing layers that are suitable for reuse in other applications, you already have applications that expose suitable business processes through service interfaces, or your application is complex and the high-level design demands separation so that teams can

focus on different areas of functionality. The layered architectural style is also appropriate if your application must support different client types and different devices, or you want to implement complex and/or configurable business rules and processes.
Consider a Separated Presentation pattern if you want improved testability and simplified maintenance of UI functionality, or you want to separate the task of designing the UI from the development of the logic code that drives it. These patterns are also appropriate when your UI view does not contain any request processing code, and does not implement any business logic.

Message Bus Architectural Style

Message bus architecture describes the principle of using a software system that can receive and send messages using one or more communication channels, so that applications can interact without needing to know specific details about each other. It is a style for designing applications where interaction between applications is accomplished by passing messages (usually asynchronously) over a common bus. The most common implementations of message bus architecture use either a messaging router or a Publish/Subscribe pattern, and are often implemented using a messaging system such as Message Queuing. Many implementations consist of individual applications that communicate using common schemas and a shared infrastructure for sending and receiving messages. A message bus provides the ability to handle:

- **Message-oriented communications**. All communication between applications is based on messages that use known schemas.
- **Complex processing logic**. Complex operations can be executed by combining a set of smaller operations, each of which supports specific tasks, as part of a multistep itinerary.
- **Modifications to processing logic**. Because interaction with the bus is based on common schemas and commands, you can insert or remove applications on the bus to change the logic that is used to process messages.
- **Integration with different environments**. By using a message-based communication model based on common standards, you can interact with applications developed for different environments, such as Microsoft .NET and Java.

Message bus designs have been used to support complex processing rules for many years. The design provides a pluggable architecture that allows you to insert applications into the process, or improve scalability by attaching several instances of the same application to the bus. Variations on the message bus style include:

- **Enterprise Service Bus (ESB)**. Based on message bus designs, an ESB uses services for communication between the bus and components attached to the bus. An ESB will usually provide services that transform messages from one format to another, allowing clients that use incompatible message formats to communicate with each other
- **Internet Service Bus (ISB)**. This is similar to an enterprise service bus, but with applications hosted in the cloud instead of on an enterprise network. A core concept of ISB is the use of Uniform Resource Identifiers (URIs) and policies to control the routing of logic through applications and services in the cloud.

The main benefits of the message-bus architectural style are:

- **Extensibility**. Applications can be added to or removed from the bus without having an impact on the existing applications.

- **Low complexity**. Application complexity is reduced because each application only needs to know how to communicate with the bus.
- **Flexibility**. The set of applications that make up a complex process, or the communication patterns between applications, can be changed easily to match changes in business or user requirements, simply through changes to the configuration or parameters that control routing.
- **Loose coupling**. As long as applications expose a suitable interface for communication with the message bus, there is no dependency on the application itself, allowing changes, updates, and replacements that expose the same interface.
- **Scalability**. Multiple instances of the same application can be attached to the bus in order to handle multiple requests at the same time.
- **Application simplicity**. Although a message bus implementation adds complexity to the infrastructure, each application needs to support only a single connection to the message bus instead of multiple connections to other applications.

Consider the message bus architectural style if you have existing applications that interoperate with each other to perform tasks, or you want to combine multiple tasks into a single operation. This style is also appropriate if you are implementing a task that requires interaction with external applications, or applications hosted in different environments.

N-Tier / 3-Tier Architectural Style

N-tier and 3-tier are architectural deployment styles that describe the separation of functionality into segments in much the same way as the layered style, but with each segment being a tier that can be located on a physically separate computer. They evolved through the component-oriented approach, generally using platform specific methods for communication instead of a message-based approach.
N-tier application architecture is characterized by the functional decomposition of applications, service components, and their distributed deployment, providing improved scalability, availability, manageability, and resource utilization. Each tier is completely independent from all other tiers, except for those immediately above and below it. The nth tier only has to know how to handle a request from the n+1th tier, how to forward that request on to the n-1th tier (if there is one), and how to handle the results of the request. Communication between tiers is typically asynchronous in order to support better scalability.
N-tier architectures usually have at least three separate logical parts, each located on a separate physical server. Each part is responsible for specific functionality. When using a layered design approach, a layer is deployed on a tier if more than one service or application is dependent on the functionality exposed by the layer.
An example of the N-tier/3-tier architectural style is a typical financial Web application where security is important. The business layer must be deployed behind a firewall, which forces the deployment of the presentation layer on a separate tier in the perimeter network. Another example is a typical rich client connected application, where the presentation layer is deployed on client machines and the business layer and data access layer are deployed on one or more server tiers.
The main benefits of the N-tier/3-tier architectural style are:

- **Maintainability**. Because each tier is independent of the other tiers, updates or changes can be carried out without affecting the application as a whole.
- **Scalability**. Because tiers are based on the deployment of layers, scaling out an application is reasonably straightforward.
- **Flexibility**. Because each tier can be managed or scaled independently, flexibility is increased.
- **Availability**. Applications can exploit the modular architecture of enabling systems using easily scalable components, which increases availability.

Consider either the N-tier or the 3-tier architectural style if the processing requirements of the layers in the application differ such that processing in one layer could absorb sufficient resources to slow the processing in other layers, or if the security requirements of the layers in the application differ. For example, the presentation layer should not store sensitive data, while this may be stored in the business and data layers. The N-tier or the 3-tier architectural style is also appropriate if you want to be able to share business logic between applications, and you have sufficient hardware to allocate the required number of servers to each tier.
Consider using just three tiers if you are developing an intranet application where all servers are located within the private network; or an Internet application where security requirements do not restrict the deployment of business logic on the public facing Web or application server. Consider using more than three tiers if security requirements dictate that business logic cannot be deployed to the perimeter network, or the application makes heavy use of resources and you want to offload that functionality to another server

Object-Oriented Architectural Style

Object-oriented architecture is a design paradigm based on the division of responsibilities for an application or system into individual reusable and self-sufficient objects, each containing the data and the behavior relevant to the object. An object-oriented design views a system as a series of cooperating objects, instead of a set of routines or procedural instructions. Objects are discrete, independent, and loosely coupled; they communicate through interfaces, by calling methods or accessing properties in other objects, and by sending and receiving messages. The key principles of the object-oriented architectural style are:

- **Abstraction**. This allows you to reduce a complex operation into a generalization that retains the base characteristics of the operation. For example, an abstract interface can be a well-known definition that supports data access operations using simple methods such as **Get** and **Update**. Another form of abstraction could be metadata used to provide a mapping between two formats that hold structured data.
- **Composition**. Objects can be assembled from other objects, and can choose to hide these internal objects from other classes or expose them as simple interfaces.
- **Inheritance**. Objects can inherit from other objects, and use functionality in the base object or override it to implement new behavior. Moreover, inheritance makes maintenance and updates easier, as changes to the base object are propagated automatically to the inheriting objects.
- **Encapsulation**. Objects expose functionality only through methods, properties, and events, and hide the internal details such as state and variables from other objects. This makes it easier to update or replace objects, as long as their interfaces are compatible, without affecting other objects and code.
- **Polymorphism**. This allows you to override the behavior of a base type that supports operations in your application by implementing new types that are interchangeable with the existing object.
- **Decoupling**. Objects can be decoupled from the consumer by defining an abstract interface that the object implements and the consumer can understand. This allows you to provide alternative implementations without affecting consumers of the interface.

Common uses of the object-oriented style include defining an object model that supports complex scientific or financial operations, and defining objects that represent real world artifacts within a business domain (such as a customer or an order). The latter is a process commonly implemented using the more specialized domain driven design style, which takes advantage of the principles of the object-oriented style. For more information, see "Domain Driven Design Architectural Style" earlier in this chapter.

The main benefits of the object-oriented architectural style are that it is:

- **Understandable**. It maps the application more closely to the real world objects, making it more understandable.
- **Reusable**. It provides for reusability through polymorphism and abstraction.
- **Testable**. It provides for improved testability through encapsulation.
- **Extensible**. Encapsulation, polymorphism, and abstraction ensure that a change in the representation of data does not affect the interfaces that the object exposes, which would limit the capability to communicate and interact with other objects.
- **Highly Cohesive**. By locating only related methods and features in an object, and using different objects for different sets of features, you can achieve a high level of cohesion.

Consider the object-oriented architectural style if you want to model your application based on real world objects and actions, or you already have suitable objects and classes that match the design and operational requirements. The object-oriented style is also suitable if you must encapsulate logic and data together in reusable components or you have complex business logic that requires abstraction and dynamic behavior.

Service-Oriented Architectural Style

Service-oriented architecture (SOA) enables application functionality to be provided as a set of services, and the creation of applications that make use of software services. Services are loosely coupled because they use standards-based interfaces that can be invoked, published, and discovered. Services in SOA are focused on providing a schema and message-based interaction with an application through interfaces that are application scoped, and not component or object-based. An SOA service should not be treated as a component-based service provider.

The SOA style can package business processes into interoperable services, using a range of protocols and data formats to communicate information. Clients and other services can access local services running on the same tier, or access remote services over a connecting network.

The key principles of the SOA architectural style are:

- **Services are autonomous**. Each service is maintained, developed, deployed, and versioned independently.
- **Services are distributable**. Services can be located anywhere on a network, locally or remotely, as long as the network supports the required communication protocols.
- **Services are loosely coupled**. Each service is independent of others, and can be replaced or updated without breaking applications that use it as long as the interface is still compatible.
- **Services share schema and contract, not class**. Services share contracts and schemas when they communicate, not internal classes.
- **Compatibility is based on policy**. Policy in this case means definition of features such as transport, protocol, and security.

Common examples of service-oriented applications include sharing information, handling multistep processes such as reservation systems and online stores, exposing industry specific data or services over an extranet, and creating mashups that combine information from multiple sources.

The main benefits of the SOA architectural style are:

- **Domain alignment**. Reuse of common services with standard interfaces increases business and technology opportunities and reduces cost.

- **Abstraction**. Services are autonomous and accessed through a formal contract, which provides loose coupling and abstraction.
- **Discoverability**. Services can expose descriptions that allow other applications and services to locate them and automatically determine the interface.
- **Interoperability**. Because the protocols and data formats are based on industry standards, the provider and consumer of the service can be built and deployed on different platforms.
- **Rationalization**. Services can be granular in order to provide specific functionality, rather than duplicating the functionality in number of applications, which removes duplication.

Consider the SOA style if you have access to suitable services that you wish to reuse; can purchase suitable services provided by a hosting company; want to build applications that compose a variety of services into a single UI; or you are creating Software plus Services (S+S), Software as a Service (SaaS), or cloud-based applications. The SOA style is suitable when you must support message-based communication between segments of the application and expose functionality in a platform independent way, when you want to take advantage of federated services such as authentication, or you want to expose services that are discoverable through directories and can be used by clients that have no prior knowledge of the interfaces.

Additional Resources

Evans, Eric. Domain-Driven Design: Tackling Complexity in the Heart of Software. Addison-Wesley, 2004.
Nilsson, Jimmy. Applying Domain-Driven Design and Patterns: With Examples in C# and NET. Addison-Wesley, 2006.
For more information about architectural styles, see the following resources:

- "*An Introduction To Domain-Driven Design*" at http://msdn.microsoft.com/en-us/magazine/dd419654.aspx.
- "*Domain Driven Design and Development in Practice*" at http://www.infoq.com/articles/ddd-in-practice.
- "*Fear Those Tiers*" at http://msdn.microsoft.com/en-us/library/cc168629.aspx.
- "*Layered Versus Client-Server*" at http://msdn.microsoft.com/en-us/library/bb421529.aspx.
- "*Message Bus*" at http://msdn.microsoft.com/en-us/library/ms978583.aspx.
- "*Microsoft Enterprise Service Bus (ESB) Guidance*" at http://www.microsoft.com/biztalk/en/us/esb-guidance.aspx.
- "*Separated Presentation*" at http://martinfowler.com/eaaDev/SeparatedPresentation.html.
- "*Services Fabric: Fine Fabrics for New-Era Systems*" at http://msdn.microsoft.com/en-us/library/cc168621.aspx.