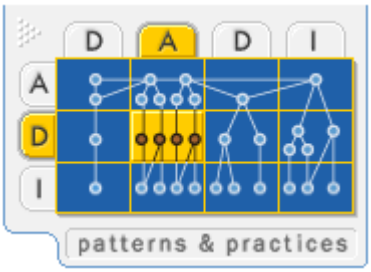
Service Interface

Retired Content
This content is outdated and is no longer being maintained. It is provided as a courtesy for individuals who are still using these technologies. This page may contain URLs that were valid when originally published, but now link to sites or pages that no longer exist. Please see the patterns & practices guidance for the most current information.



Version 1.0.0

GotDotNet community for collaboration on this pattern

Complete List of patterns & practices

Context

You are designing an enterprise application, and you need to make some of its functionality available across a network. This functionality needs to be accessible to various types of systems, so interoperability is a key aspect of the design. In addition to interoperability, you also may need to support different types of communications protocols and accommodate varying operational requirements.

Problem

How do you make pieces of your application's functionality available to other applications, while ensuring that the interface mechanics are decoupled from the application logic?

Forces

While designing your application, you must address the following forces:

- It is desirable to separate elements that are responsible for the application's business logic from the elements responsible for communication protocols, data transformation, and fulfillment of service contracts. Doing so furthers the general design objective of separation of concerns.
- Consumers of your application may want responses optimized for particular usage scenarios. For example, some consumers may want responses optimized for direct display to users, while others may want responses optimized for software processing.
-

Consumers of your application may want to communicate with the application using different technologies. For instance, consumers that are external to your company may want to access the application through SOAP over the Internet, while consumers that are internal to your company may want to access the application through .NET remoting.

- The application itself may impose different operational requirements on different consumers. For example, your application may have security requirements that authorize consumers internal to your company to perform update and delete operations, while consumers that are external to your company are only authorized to perform read-only operations. Or, for example, different consumers may need different transactional support from the application. To some clients, the context in which specific transactions occur is not important while other clients may need precise control of the transactional context. A handle to this context might then be passed to other elements of the application as needed.

- The application's ability to respond to changes in the business environment in a timely manner is greatly enhanced if changes to the business logic are isolated from the mechanisms used by consumers to interact with the application. For example, the fact that a particular set of business logic was implemented in a custom built component and then later implemented as a wrapper around a packaged solution should ideally have no impact on the consumers of the application.

Solution

Design your application as a collection of software services, each with a service interface through which consumers of the application may interact with the service.

A software service is a discrete unit of application logic that exposes a message-based interface that is suitable for being accessed by other applications. [Microsoft02-2] Each software service has an associated interface that it presents to the consumers. This interface defines and implements a contract between the consumers of the service and the provider of the service. This contract and its associated implementation are referred to as a service interface.

Figure 1 shows a service gateway consuming a service provided by a service interface. The collaboration between these two elements is governed by a contract.

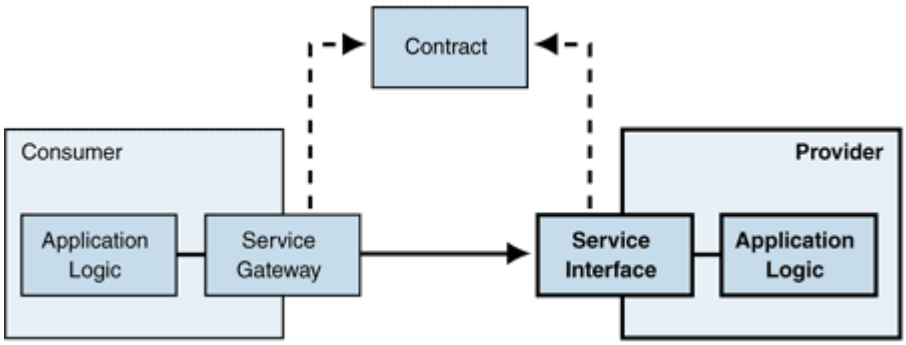


Figure 1: Service elements

Service Interface

As Figure 1 shows, *Service Interface* provides an entry point that consumers use to access the functionality exposed by the application. The *Service Interface* is usually network addressable, meaning that it is capable of being accessed by the consumer over some sort of communication network. The network address can be a well-known location or it can be obtained from a service directory such as UDDI.

A key aspect of the design of a service interface is to decouple the implementation needed to communicate with other systems from the application's business logic. The service interface provides a much more coarse-grained interface while preserving the

semantics and finer granularity of the application logic. It also provides a barrier that enables the application logic to change without affecting the consumers of the interface.

The service interface implements the contract between the consumer and provider. This contract allows them to exchange information even if they are on different systems. The service interface is responsible for all of the implementation details needed to perform this communication. Such details include but are not limited to:

- **Network protocol**. The service interface should encapsulate all aspects of the network protocol used for communication between the consumer and service. For example, suppose that a service is exposed to consumers through HTTP over a TCP/IP network. You can implement the service interface as an ASP.NET component published to a well-known URL. The ASP.NET component receives the HTTP request, extracts the information needed by the service to process the request, invokes the service implementation, packages the service response, and sends the response back to the consumer as an HTTP response. From the service perspective, the only component that understands HTTP is the service interface. The service implementation has its own contract with the service interface and should have no dependencies on the specifics of the technology that consumers use to communicate with the service interface.

- **Data formats**. The service interface translates between consumer data formats and the data formats that the service expects. For example, consumers external to the company may supply data and expect reply data to be in an XML format that conforms to an industry-standard XML schema. Consumers internal to the company may want to use an XML format optimized for this particular service. The service interface is responsible for transforming and mapping both data formats in a format that the service can use. The service implementation does not have any knowledge of the specific data formats the service interface might use to communicate with the consumers.

- **Security**. The service interface should be considered its own trust boundary. Different consumers may have different security requirements, so it is up to the service interface to implement these consumer-specific requirements. For instance, consumers external to the company will generally have more restrictive security requirements than consumers internal to the company. External consumers may have strong authentication requirements and may only be authorized to perform a very limited subset of the operations authorized for internal consumers. Internal consumer may be implicitly trusted for most operations and only require authorization for the most sensitive operations.

- **Service level agreements**. The service interface has a significant role in ensuring that the service meets its service level commitments to a specific set of consumers. Service interfaces may implement caching to increase response time and reduce bandwidth consumption. Multiple instances of a service interface may be deployed across a load-balanced set of processing nodes to achieve scalability, availability, and fault-tolerance requirements.

Minimizing the Number of Service Interfaces

In general, you will need one service interface for each unique usage scenario, technology stack, service level agreement, or operational requirement. However, the more service interfaces supported by your application, the more work is involved in building and maintaining the implementation. Therefore, you should try to minimize the number of service interfaces that an application needs to support. For example, an application may offer two service interfaces for accessing its functionality. The first service interface may be optimized for consumers that are external to the company. It may specify a few very coarse-grained sets of request-and-response pairs using SOAP over HTTP communication technology and mandate very strict security requirements. The second service interface may be optimized for consumers that are internal to the company. It may specify a somewhat larger number of request-and-response pairs that are not quite as coarse-grained as those specified in the first service interface, and emphasize performance requirements over security concerns.

Example

See *Implementing Service Interface in .NET*.

Testing Considerations

Service Interface encapsulates all the details of providing a service and decouples it from the application logic. This separation enables you to replace the application logic with mock [Mackinnon00] implementations. These mock implementations replace the real application code with dummy implementations that emulate the real code. Using mock implementations allows you to write tests that verify that the code works without having to depend on the actual application code. You can also extend the mock implementations to simulate error conditions that might be difficult or impossible to simulate with the real code.

Resulting Context

Using the *Service Interface* pattern results in the following benefits and liabilities:

Benefits

- The service interface mechanics are decoupled from the application logic. This separation allows you to easily add new interfaces and to change the implementation of the underlying application with minimal impact on consumers.

- Decoupling the service interface code from the service implementation code enables you to deploy the two code bases on separate tiers, potentially increasing the deployment flexibility of the solution.

Liabilities

- Many platforms make exposing the application functionality simple. However, this can lead to a poor decision in terms of granularity. If the interface is too fine-grained, you can end up making too many calls to the service to perform a specific action. You need to design your service interfaces to be appropriate for network or out-of-process communication.

- Each additional service interface provided by a service increases the amount of work required to make a change to the functionality exposed by a service.

- The *Service Interface* pattern adds complexity and performance overhead that may not be justified for very simple service-oriented applications.

Related Patterns

For more information, see the following related patterns:

- *Service Gateway*. *Service Gateway* performs the role of the consumer of a service interface.

- *Remote Facade* [Fowler03]. *Service Interface* is a specific type of *Remote Facade* adapted for use in service-oriented architectures. A remote facade is similar to a remote proxy, but sometimes uses encapsulation to make the remote interface more coarse-grained.

- *Service Layer* [Fowler03]. As the number and complexity of service interfaces increase, it may make sense to pool the common pieces of functionality into its own software layer.

Acknowledgments

[Fowler03] Fowler, Martin. *Patterns of Enterprise Application Architecture*. Addison-Wesley, 2003.

[Mackinnon00] Mackinnon, Tim, et al. "Endo-Testing: Unit Testing with Mock Objects." *eXtreme Programming and Flexible Processes in Software Engineering - XP2000* conference.

[Microsoft02-2] Microsoft Corporation. "Application Architecture: Conceptual View." *.NET Architecture Center*. Available from MSDN at: http://msdn.microsoft.com/architecture/default.aspx?pull=/library/en-us/dnea/html/eaappconland.asp.