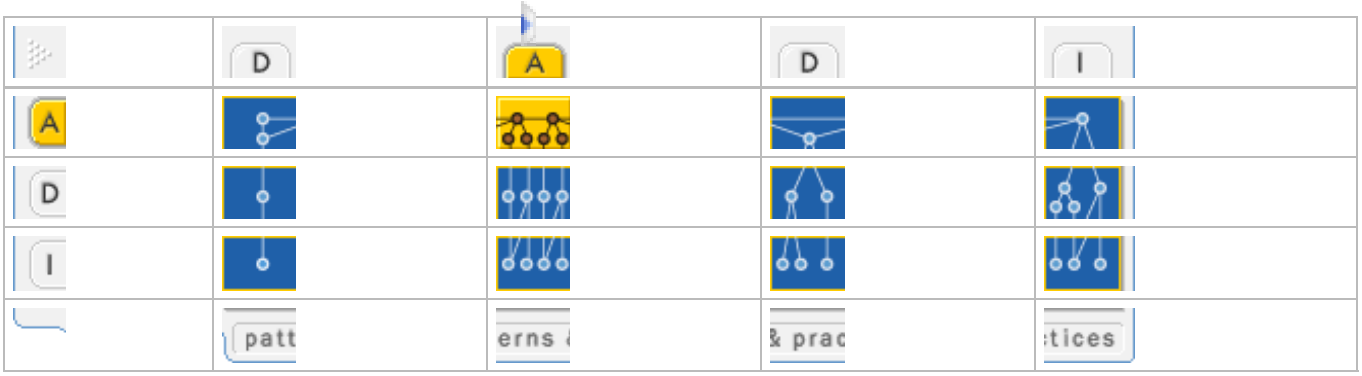
Layered Application

Layered Application



Version 1.0.1

GotDotNet community for collaboration on this pattern

Complete List of patterns & practices

Context

You are designing a complex enterprise application that is composed of a large number of components across multiple levels of abstraction.

Problem

How do you structure an application to support such operational requirements as maintainability, reusability, scalability, robustness, and security?

Forces

When structuring your application, you must reconcile the following forces within the context of your environment:

- Localizing changes to one part of the solution minimizes the impact on other parts, reduces the work involved in debugging and fixing bugs, eases application maintenance, and enhances overall application flexibility.
- Separation of concerns among components (for example, separating the user interface from the business logic, and separating the business logic from the database) increases flexibility, maintainability, and scalability.
- Components should be reusable by multiple applications.
- Independent teams should be able to work on parts of the solution with minimal dependencies on other teams and should be able to develop against well-defined interfaces.
- Individual components should be cohesive.
- Unrelated components should be loosely coupled.
- Various components of the solution are independently deployed, maintained, and updated, on different time schedules.

- Crossing too many component boundaries has an adverse effect on performance.
- To make a Web application both secure and accessible, you need to distribute the application over multiple physical tiers. This enables you to secure portions of the application behind the firewall and make other components accessible from the Internet.
- To ensure high performance and reliability, the solution must be testable.

Solution

Separate the components of your solution into layers. The components in each layer should be cohesive and at roughly the same level of abstraction. Each layer should be loosely coupled to the layers underneath. *Pattern-Oriented Software Architecture, Vol 1* [Buschmann96] describes the layering process as follows:

> Start at the lowest level of abstraction - call it Layer 1. This is the base of your system. Work your way up the abstraction ladder by putting Layer J on top of Layer J-1 until you reach the top level of functionality - call it Layer N.

Figure 1 shows how this layering scheme would look.

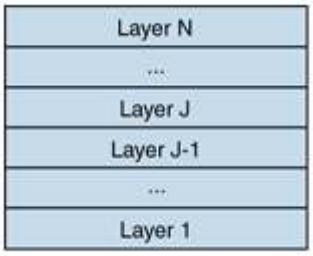


Figure 1: Layers

Structure

The key to *Layered Application* is dependency management. Components in one layer can interact only with peers in the same level or components from lower levels. This helps reduce the dependencies between components on different levels. There are two general approaches to layering: strictly layered and relaxed layered.

A strictly layered approach constrains components in one layer to interacting only with peers and with the layer directly below. If the application is layered as shown in Figure 1, for example, Layer J can only interact with components from Layer J-1, Layer J-1 can only interact with Layer J-2, and so on.

A relaxed layered application loosens the constraints such that a component can interact with components from any lower layer. Therefore, in Figure 1, not only can Layer J interact with Layer J-1, but with layers J-2 and J-3.

The relaxed approach can improve efficiency because the system does not have to forward simple calls from one layer to the next. On the other hand, the relaxed approach does not provide the same level of isolation between the layers and makes it more difficult to swap out a lower layer without affecting higher layers.

For large solutions involving many software components, it is common to have a large number of components at the same level of abstraction that are not cohesive. In this case, each layer may be further decomposed into one or more cohesive subsystems. Figure 2 demonstrates a possible UML notation for representing layers that are composed of multiple subsystems.

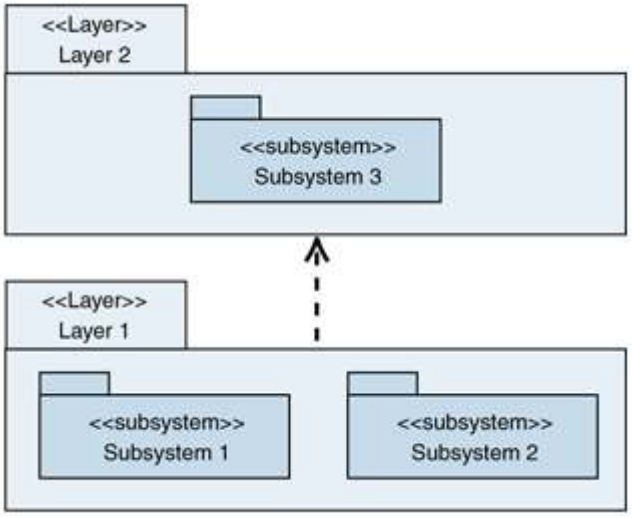


Figure 2: UML representation of layers composed of subsystems

The basic *Layered Application* pattern is often augmented with the following techniques:

- ***Layer Supertype***[Fowler03]. If the components in the layer share a set of common behaviors, you extract those behaviors into a common class or component from which all the components in the layer inherit. Not only does this ease maintenance and promote reuse, it also reduces the dependencies between layers by allowing the common behaviors to be invoked through a runtime reference to the supertype instead of a specific component.
- **Abstract interface**. An abstract interface is defined for each component in a layer that is called by components in a higher level. The higher layers access the lower-level components through the abstract interfaces instead of calling the components directly. This allows the implementation of the lower-level components to change without affecting the higher-level components.
- **Layer Facade**. For larger systems, it is common to use the *Facade* patternto provide a single unified interface to a layer or subsystem instead of developing an abstract interface for each exposed component [Gamma95]. This gives you the lowest coupling between layers, because higher-level components only reference the facade directly. Be sure to design your facade carefully. It will be difficult to change in the future, because so many components will depend on it.

Dynamics

There are basically two modes of interaction within the layered application:

- Top-down
- Bottom-up

In the *top-down* mode, an external entity interacts with the topmost layer of the stack. The topmost layer uses one or more services of the lower-level layers. In turn, each lower level uses the layers below it until the lowest layer is reached.

For the sake of discussion, this pattern assumes that the external entity is a client application and the layered application is for a server-based application that exposes its functionality as a set of services. Figure 3 is a UML sequence diagram that depicts a common top-down scenario.

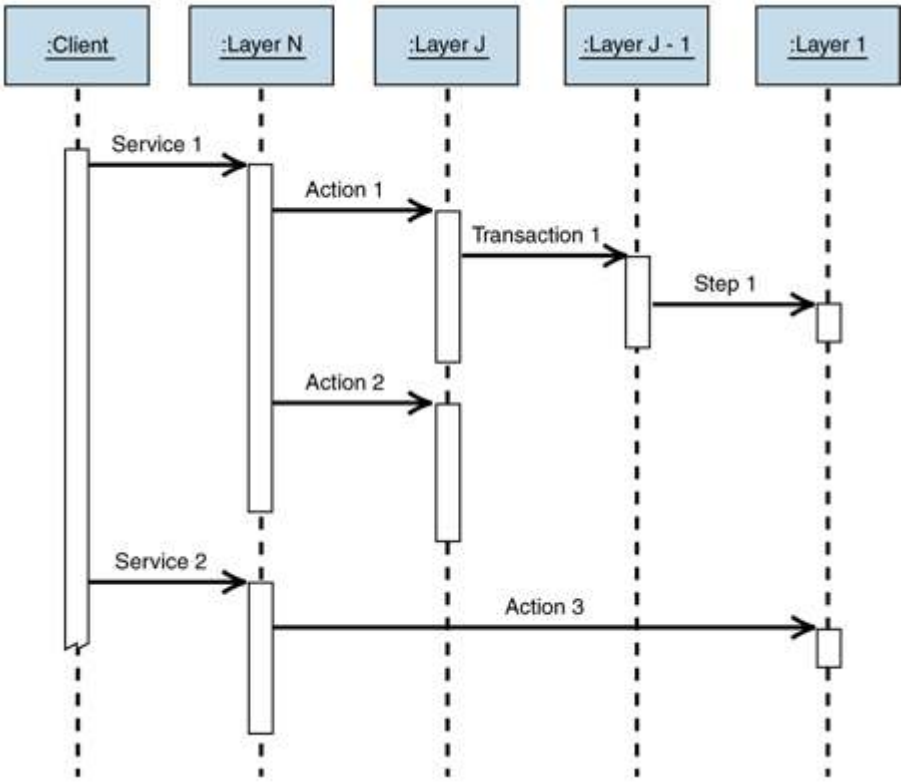


Figure 3: Sequence diagram of a top-down scenario

In this scenario, the client application consumes a set of services offered by a server-based application. These services are exposed by the topmost layer of the server application. Therefore, the client only must interact with the topmost layer and has no direct knowledge of any lower layers. A couple of factors are worth noting.

First, a single incoming invocation can result in multiple outgoing invocations. The invocation of Service 1 on Layer N illustrates this case. This quite often occurs when a higher-level service aggregates the results of several lower-level services or coordinates the execution of multiple lower services that must be executed in a particular order. For example, an ASP.NET page might feed the output of a customer domain component to an order component whose output is in turn fed to an invoice component.

Second, this scenario illustrates the relaxed layered approach. The implementation of Service 2 bypasses all the intermediate layers and calls Layer 1 directly. A common example of this is a presentation layer accessing a data access layer directly, bypassing any intermediate business logic layers. Data maintenance applications often use this approach.

Third, an invocation of the service at the top layer does not necessarily invoke all of the layers. This concept is illustrated by the Service 1 to Action 2 sequence, which occurs when a higher level can process an invocation on its own or has cached the results of an earlier request. For example, domain components quite often cache results of database queries, which remove the need to invoke the data access layer for future invocations.

In the *bottom-up* mode, Layer 1 detects a situation that affects higher levels. The following scenario assumes that Layer 1 is monitoring the state of some external entity such as the file system of the server on which the server application is running. Figure 4 depicts a typical bottom-up scenario as a UML sequence diagram.

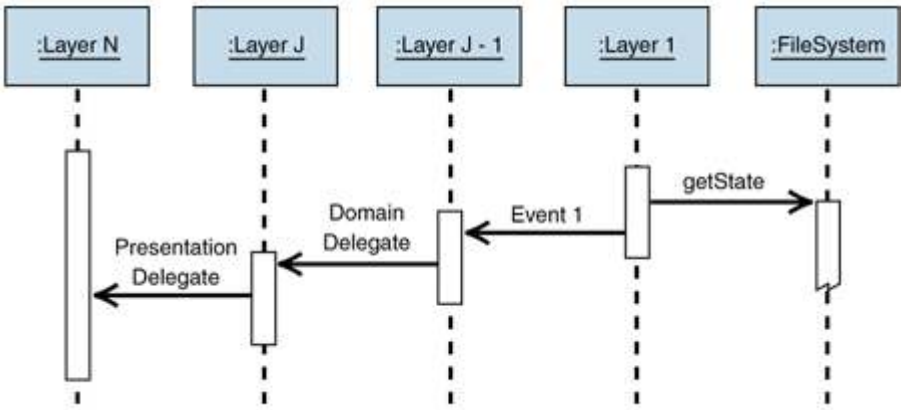


Figure 4: Sequence diagram of a bottom-up scenario

In this scenario, Layer 1 is monitoring the state of the local file system. When it detects a change, it fires an event exposed by a component from the J-1 layer. This component then invokes a callback delegate of Layer J, where the state of the domain layer is updated. The domain component then notifies Layer N that it has been updated by a delegate provided by Layer N for that purpose.

As in the first scenario, an input at one level can result in multiple outputs. A lower layer can notify any layer higher than it, not just the next higher layer. And finally, a notification does not necessarily have to percolate up the entire chain.

Take careful notice of how differently the layers interact in the bottom-up scenario as opposed to the top-down scenario. In the top-down scenario, higher layers call lower layers directly and thus are dependent on them. In the bottom-up scenario, however, lower layers communicate with higher layers through events, callbacks, and delegates. This level of indirection is required to keep lower layers from being dependent on higher layers. Making lower layers dependent on higher layers reduces quite a few of the benefits that the layered architecture provides.

Implementation

There are basically two approaches to implementing the *Layered Application* pattern:

- Create your own layering scheme
- Reuse an existing layering scheme

Creating Your Own Layering Scheme

Buschmann provides a great discussion about implementing your own layered application. A brief overview is provided here, but it is highly recommended that you study the *Layers* pattern in Buschmann if you need to define your own layered application. The outline of the process is as follows:

- Use a well-defined set of criteria to group the functionality of the solution into a set of layers and define the services that each layer provides. This is an iterative process in which you will likely try multiple combinations of criteria, numbers of levels, functionality decomposition, and service assignments. UML sequence diagrams describing the interactions of layers and solution components are an ideal tool for understanding the tradeoffs involved with each candidate layering scheme.
- Define the interfaces between each level and the protocols they require to communicate with each other. To avoid making lower levels dependent on higher levels, use techniques such as asynchronous messaging, callbacks, and events for communications that need to travel up the stack. Again, UML sequence diagrams are a great tool for ensuring that you have a complete and consistent set of interfaces. The diagrams give you a visual clue of the granularity or chattiness of

your interface and protocols. Be particularly aware of the number of times you cross a layer boundary for a given scenario and look for opportunities to refactor your design to reduce the number of boundary crossings. A key design decision is to determine how much coupling should exist between levels. Do components in Layer J directly access components in Layer J-1? This makes higher levels dependent on lower-level implementation details. Patterns such as *Facade* and other decoupling techniques should be explored to minimize this type of coupling.

- Design the implementation of the layers. Traditional object-oriented design techniques work quite well for this task. Be sure to consider patterns such as *Adapter*, *Bridge*, and *Strategy* [Gamma95] to enable the switching out of multiple implementations of a given layer's interface. This capability is especially valuable when it comes to testing the interfaces and level implementations. Another critical design decision is how to handle errors. A consistent error-handling strategy must be defined across all the levels. Consider the following when designing your error-handling strategy:
    - Try to deal with the error at the lowest level possible.
    - Avoid exposing lower-level abstractions to higher levels through the exception handling mechanism.
    - If you must escalate an exception up the stack, convert lower-level exceptions to exceptions that have some meaning to the handling layer.

Reusing an Existing Layering Scheme

The other approach is to reuse an existing reference layered application to provide structure for your applications. The canonical three-layered application consists of the following three layers: presentation, domain, and data source. Even something as simple as this goes a long way towards achieving the benefits of the *Layered Application* pattern. An enhanced version of the canonical model is discussed in *Layered Services Application*.

Martin Fowler has found the use of mediating layers between the presentation and domain layers as well as between the domain and data source layers useful at times. For more information, see Fowler's book, *Patterns of Enterprise Application Architecture* [Fowler03].

Testing Considerations

Layered Application enhances testability in several ways:

- Because each layer interacts with the other layers only through well-defined interfaces, it is easy to plug in alternative implementations of a layer. This allows some testing on a layer before the layers it depends on are complete. In addition, an alternative implementation that immediately returns a set of known good data can be substituted for a layer that takes a long time to compute a correct answer, thus speeding up test execution. This ability is greatly enhanced if the layered supertype, abstract interface, and layer facade techniques are used, because they further decrease the dependencies between layers.
- It is easier to test individual components, because the dependencies between components are constrained such that components in higher levels can only call components in lower levels. This helps to isolate individual components for testing and facilitates swapping out lower-level components with special-purpose testing components.

Example

It is quite common for enterprise application architects to compose their solutions into the following three layers:

- **Presentation**. This layer is responsible for interacting with the user.

- **Business**. This layer implements the business logic of the solution.
- **Data**. This layer encapsulates the code that accesses the persistent data stores such as a relational database.

For more information, see the *Three-Layered Services Application* pattern.

Resulting Context

Layered Application generally results in the following benefits and liabilities:

Benefits

- Maintenance of and enhancements to the solution are easier due to the low coupling between layers, high cohesion between the layers, and the ability to switch out varying implementations of the layer interfaces.
- Other solutions should be able to reuse functionality exposed by the various layers, especially if the layer interfaces are designed with reuse in mind.
- Distributed development is easier if the work can be distributed at layer boundaries.
- Distributing the layers over multiple physical tiers can improve scalability, fault-tolerance, and performance. For more information, see the *Tiered Distribution* pattern.
- Testability benefits from having well-defined layer interfaces as well as the ability to switch out various implementations of the layer interfaces.

Liabilities

- The extra overhead of passing through layers instead of calling a component directly can negatively affect performance. To help offset the performance hit, you can use the relaxed layers approach, in which higher layers can directly call lower layers.
- Development of user-intensive applications can sometime take longer if the layering prevents the use of user interface components that directly interact with the database.
- The use of layers helps to control and encapsulate the complexity of large applications, but adds complexity to simple applications.
- Changes to lower-level interfaces tend to percolate to higher levels, especially if the relaxed layered approach is used.

Acknowledgments

[Buschmann96] Buschmann, Frank, et al. *Pattern-Oriented Software Architecture, Vol 1*. Wiley & Sons, 1996.

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

[Gamma95] Gamma, Helm, Johnson, and Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.