Chapter 10. Layered Architecture Style

The *layered* architecture style, also known as *n-tiered*, is one of the most common architecture styles. It's the de facto standard for many legacy applications because of its simplicity, familiarity, and low cost.

The layered architecture style can fall into several architectural antipatterns, including the *Architecture by Implication* and the *Accidental Architecture* antipatterns. When developers or architects "just start coding," unsure which architecture style they are using, chances are good that they're implementing the layered architecture style.

Topology

Components within the layered architecture style are organized into logical horizontal layers, as shown in Figure 10-1, with each layer performing a specific role within the application (such as presentation logic or business logic). Although there are no specific restrictions in terms of the number and types of layers that must exist, most layered architectures consist of four standard layers: Presentation, Business, Persistence, and Database. Some architectures combine the Business and Persistence layers, particularly when the persistence logic (such as SQL or HSQL) is embedded within the Business layer components. Smaller applications may have only three layers, whereas larger and more complex business applications may contain five or more.

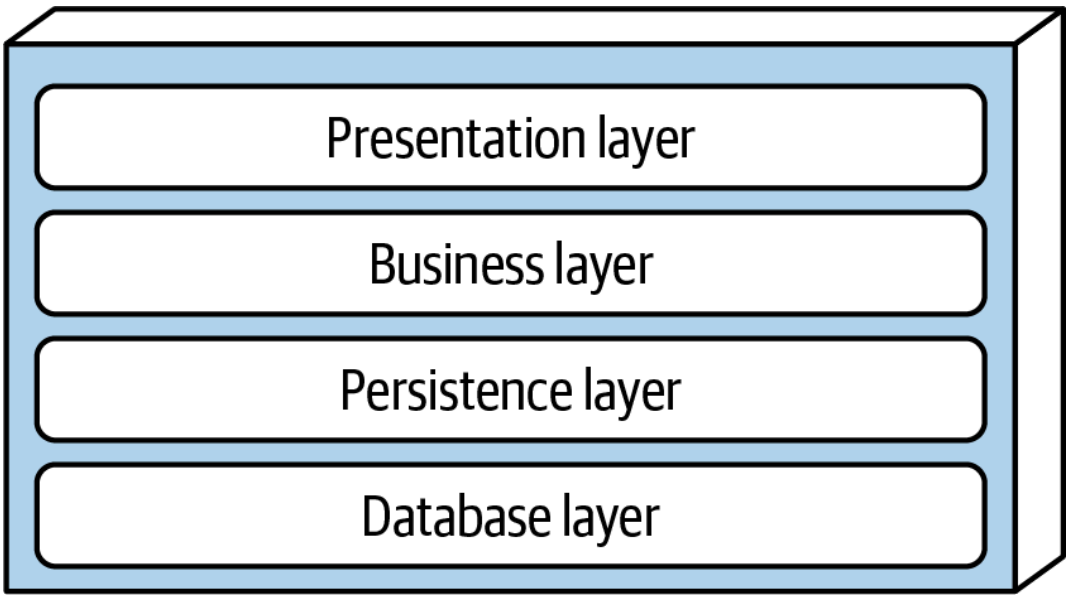

Figure 10-1. Standard logical layers within the layered architecture style

Figure 10-2 illustrates topology variants from a physical layering (deployment) perspective. The first variant combines the Presentation, Business, and Persistence layers into a single deployment unit, with the Database layer typically represented

as a separate external physical database (or filesystem). The second variant physically separates the Presentation layer into its own deployment unit, with the Business and Persistence layers combined into a second deployment unit. Again, with this variant, the Database layer is usually physically separated through an external database or filesystem. A third variant combines all four standard layers into a single deployment, including the Database layer. This variant might be useful for smaller applications with an internally embedded database or an in-memory database, such as mobile device applications. Many on-premises (on-prem) products are built and delivered to customers using this third variant.

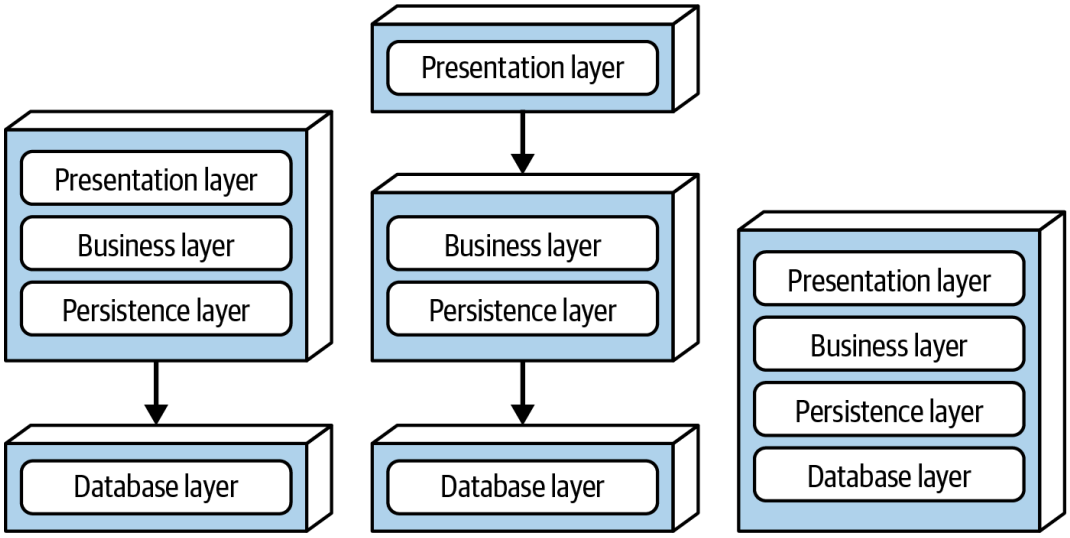

Figure 10-2. Physical topology (deployment) variants

Each layer has a specific role and responsibility, and forms an abstraction around the work that needs to be done to satisfy a particular business request. For example, the Presentation layer is responsible for handling all UI and browser communication logic, whereas the Business layer is responsible for executing specific business rules associated with the request. The Presentation layer doesn't need to know or worry about how to get customer data; it only needs to display that information on a screen in a particular format. Similarly, the Business layer doesn't need to be concerned about how to format customer data for display on a screen or even where that data is coming from; it only needs to get the data from the Persistence layer, perform business logic against it (such as calculating values or aggregating data), and pass that information up to the Presentation layer.

This separation of concerns within the layered architecture style makes it easy to build effective role and responsibility models. Components within a specific layer are limited in scope, dealing only with the logic that pertains to that layer. For example, components in the Presentation layer only handle presentation logic, whereas components residing in the Business layer only handle business logic. This allows developers to leverage their particular technical expertise to focus on the technical aspects of the domain (such as presentation logic or persistence logic). The trade-off of this benefit, however, is a lack of overall *holistic agility* (the entire system's ability to respond quickly to change).

The layered architecture is a *technically partitioned* architecture (as opposed to *domain-partitioned* architecture). This means, as you learned in Chapter 9, that components are separated by their technical role in the architecture (such as presentation or business) rather than by domain (such as customer). As a result, any particular business domain is spread throughout all of the layers of the architecture. For example, the domain of "customer" is contained in the Presentation layer, Business layer, Rules layer, Services layer, and Database layer,

making it difficult to apply changes to that domain. As a result, a DDD approach does not fit particularly well with the layered architecture style.

Style Specifics

Layers in this architectural style encapsulate specific areas of technical responsibility, but the layers themselves may exhibit other characteristics.

Layers of Isolation

Each layer can be either closed or open. If a layer is *closed*, then as a request moves from the top layer down to the bottom layer, the request cannot skip any layers. It must go through the layer immediately beneath to get to the next layer (see Figure 10-3). For example, in an architecture where all the layers are closed, a request originating from the Presentation layer must first go through the Business layer and then to the Persistence layer before finally making it to the Database layer.

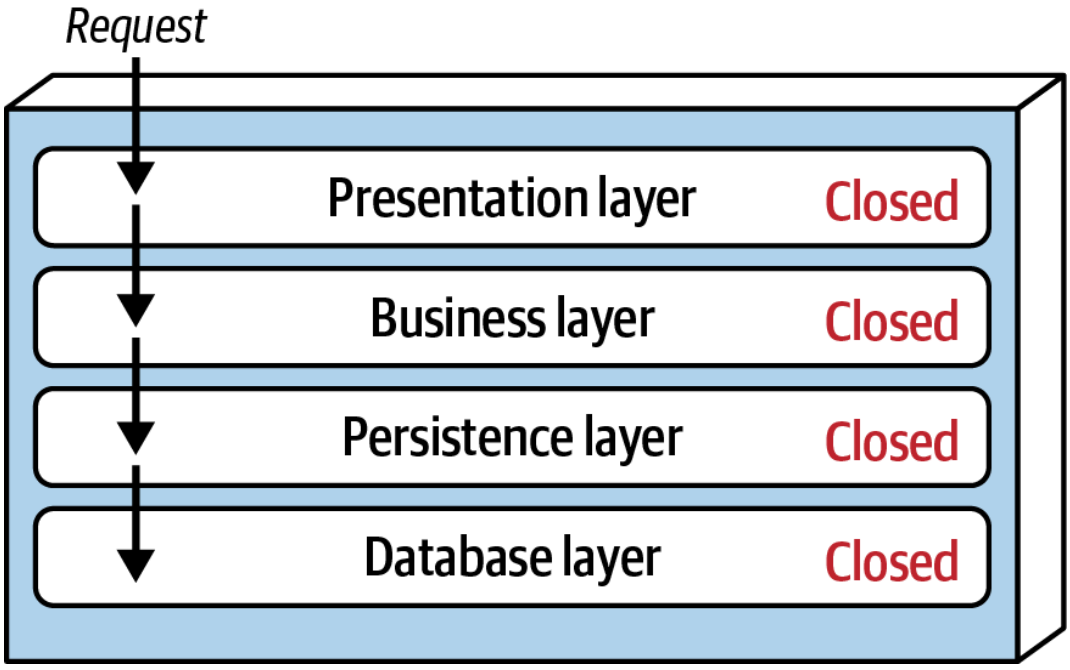

Figure 10-3. Closed layers within a layered architecture

Notice that in Figure 10-3 it would be much faster and easier for the Presentation layer to access the database directly for simple retrieval requests, bypassing any unnecessary layers (what used to be known in the early 2000s as the *Fast-Lane Reader* pattern). For this to happen, the Business and Persistence layers would have to be *open*, allowing requests to bypass other layers. Which is better—open layers or closed layers? The answer to this question lies in a key concept known as *layers of isolation*.

The *layers of isolation* concept means that changes made in one layer of the architecture generally don't impact or affect components in other layers, provided the contracts between those layers remain unchanged. Each layer is independent of the other layers, with little or no knowledge of their inner workings. However, to support layers of isolation, layers involved with the major flow of a request have to be closed. If the Presentation layer can access the Persistence layer directly, then changes made to the Persistence layer would impact both the Business layer *and* the Presentation layer, producing a very tightly coupled

application with layer interdependencies between components. This makes a layered architecture very brittle, as well as difficult and expensive to change.

Using layers of isolation also allows any layer in the architecture to be replaced without impacting any other layer (again, assuming well-defined contracts and the use of the *Business Delegate* pattern).[1] For example, you can leverage layers of isolation to replace your older UI framework with a newer one, all within the Presentation layer.

Adding Layers

While closed layers facilitate layers of isolation and therefore help isolate change, there are times when it makes sense for certain layers to be open. For example, suppose your layered architecture's Business layer has shared objects that contain common functionality for business components (such as date and string utility classes, auditing classes, logging classes, and so on). You make an architecture decision restricting the Presentation layer from using these shared business objects. This constraint is illustrated in Figure 10-4, with the dotted line going from a presentation component to a shared business object in the Business layer. This scenario is difficult to govern and control because, *architecturally*, the Presentation layer has access to the Business layer, and hence access to the shared objects within that layer.

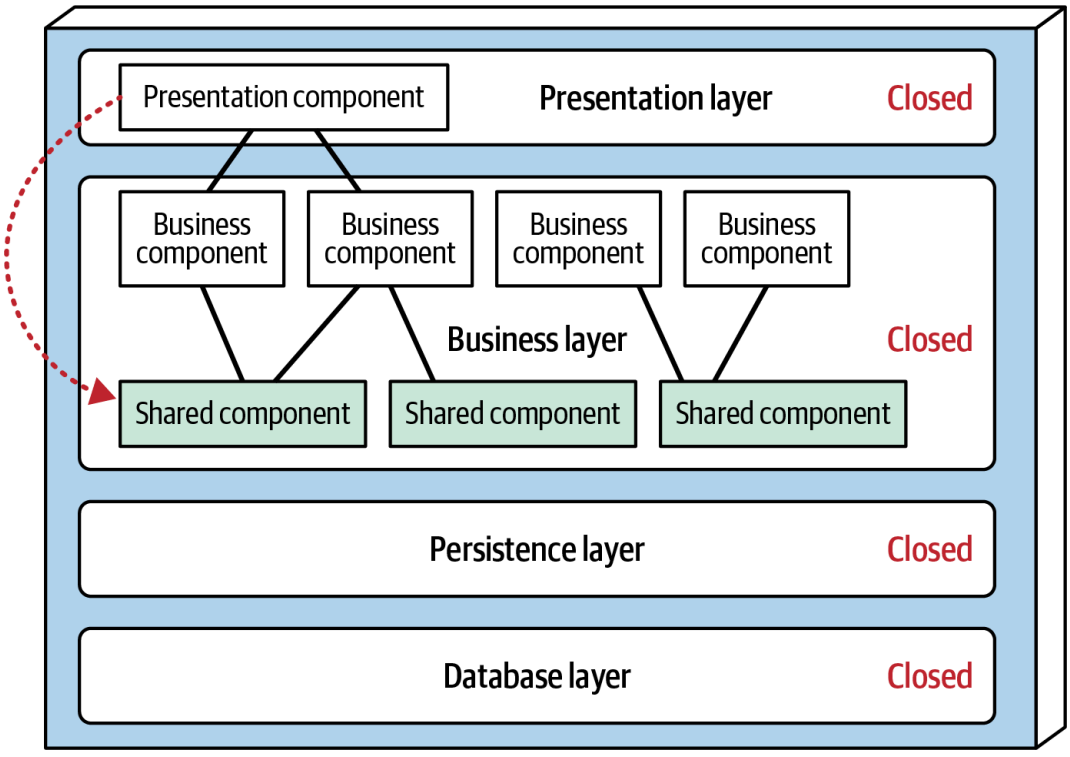

Figure 10-4. Shared objects within the Business layer

One way to mandate this restriction architecturally would be to add a new Services layer containing all of the shared business objects (see Figure 10-5). Adding this new layer would architecturally restrict the Presentation layer from accessing the shared business objects because the Business layer is closed. However, you must mark the new Services layer as *open*; otherwise, the Business layer would be forced to go through the Services layer to access the Persistence layer. Marking the Services layer as open allows the Business layer to either access that layer (as indicated by the solid arrow) or bypass it and go to the next one down (as indicated by the dotted arrow).

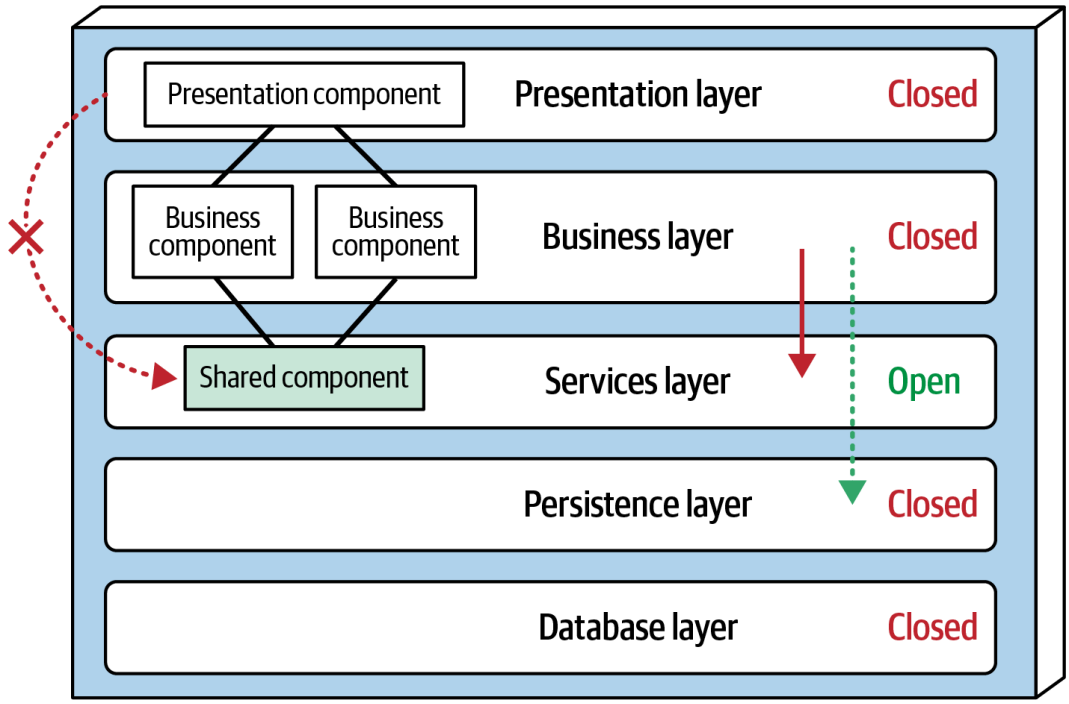

Figure 10-5. Adding a new Services layer to the architecture

Leveraging the concept of open and closed layers helps define the relationship between architecture layers and request flows. It also provides developers with the necessary information and guidance to understand layer access restrictions within the architecture. Failure to document or properly communicate which layers in the architecture are open and closed (and why) usually results in tightly coupled, brittle architectures that are very difficult to test, maintain, and deploy.

Every layered architecture will have at least some scenarios that fall into the *Architecture Sinkhole* antipattern. This antipattern occurs when requests are simply passed through from layer to layer, with no business logic performed. For example, suppose the Presentation layer responds to a user's simple request to retrieve basic customer data (such as name and address). The Presentation layer passes the request to the Business layer, which does nothing but pass the request on to the Rules layer, which in turn does nothing but pass it on to the Persistence layer, which then makes a simple SQL call to the Database layer to retrieve the customer data. The data is then passed all the way back up the stack with no additional processing or logic to aggregate, calculate, apply rules to, or transform any of it. This results in unnecessary object instantiation and processing, draining both memory consumption and performance.

The key to determining whether this antipattern is at play is to analyze the percentage of requests that fall into this category. The 80-20 rule is usually a good practice to follow. For example, it is acceptable if only 20% of the requests are sinkholes; however, if it's 80%, that's a good indicator that the layered architecture is not the correct architecture style for the problem domain. Another approach to solving the Architecture Sinkhole antipattern is to make all the layers in the architecture open—realizing, of course, that the trade-off is increased difficulty in managing change.

Data Topologies

Traditionally, layered architectures form a monolithic system alongside a single, monolithic database. The common Persistence layer is often used to map object

hierarchies between favored object-oriented languages and the set-based realm of relational databases.

Cloud Considerations

Because layered architectures are typically monolithic and partitioned into layers, the cloud options are limited to deploying one or more layers via a cloud provider. The technical partitioning inherent in this architecture is a good fit for separated deployments via the cloud. However, communication latency between on-premises servers and the cloud may create issues, because workflows typically go through most of the layers in this architecture.

Common Risks

Layered architectures don't support fault tolerance, due to their monolithic deployments and lack of architectural modularity. If one small part of a layered architecture causes an out-of-memory condition to occur, the entire application unit crashes. Overall availability is also impacted due to most monolithic applications' high mean time to recover (MTTR): startup times can range from 2 minutes for smaller applications, to 15 minutes or more for most large applications.

Governance

The news is excellent for this architecture style's governance: because it is so common, the architects who built some of the original structural testing tools did so with this architecture in mind. In fact, the example fitness function in Figure 6-4 was created for the layered architecture (see Example 10-1).

Example 10-1. ArchUnit fitness function to govern layers

```
layeredArchitecture()
    .layer("Controller").definedBy("..controller..")
    .layer("Service").definedBy("..service..")
    .layer("Persistence").definedBy("..persistence..")

    .whereLayer("Controller").mayNotBeAccessedByAnyLayer()
    .whereLayer("Service").mayOnlyBeAccessedByLayers("Controller")
    .whereLayer("Persistence").mayOnlyBeAccessedByLayers("Service")
```

In Example 10-1, the architect defines the layers in the architecture, providing convenient names such as `Controller` for the component represented by the package name. (In ArchUnit syntax, two periods (`..`) on either side of a package name indicate its ownership.) Then the architect defines the communication permitted between layers, governing their open and closed layers.

Fitness function libraries support the layered architecture style extremely well, allowing architects to automate governance of the relationships they design between layers during implementation.

Team Topology Considerations

Unlike some of the architectural styles described in this book, the layered architecture style is generally independent of team topologies and will work with any team configuration:

Stream-aligned teams

> Because the layered architecture is typically small and self-contained and represents a single journey or flow through the system, it works well with stream-aligned teams. With this team topology, teams generally own the flow through the system through each layer from beginning to end, creating workflows as part of their solutions.

Enabling teams

> Because the layered architecture is highly modular and separated by technical concerns, it pairs well with enabling team topologies. Specialists and cross-cutting team members can interface with one or more layers to make suggestions and perform experiments, without affecting the rest of the flow. For example, the team could experiment with a new UI library by adding new behaviors to the Presentation layer, while isolating the other layers from the changes.

Complicated-subsystem teams

> Because each layer performs a very specific task, this style works well with the complicated-subsystem team topology. For example, the Persistence layer provides the perfect hook for a team that requires access to operational data for analytical purposes. If allowed access at the Persistence layer, the complicated-subsystem team can work without affecting any other layers that the stream-aligned teams still owns.

Platform teams

> Platform teams working on a layered architecture can leverage its high degree of modularity by utilizing the many tools available for it.

The biggest challenge with layered architectures for most platform teams tracks the overall issue with monoliths in general: as they grow, they become increasingly unwieldy. If teams keep adding features to a monolith, no matter how well partitioned and governed, it will eventually start straining at some constraints: database connections, memory, performance, concurrent users, or a host of other impending problems. Keeping the system operational will require the platform team to do increasingly difficult work.

Style Characteristics

A one-star rating in the characteristics ratings table (shown in Figure 10-6) means the specific architecture characteristic isn't well supported in the architecture, whereas a five-star rating means the architecture characteristic is one of the strongest features in the architecture style. The definition for each characteristic identified in the scorecard can be found in Chapter 4.

	Architectural characteristic	Star rating
	Overall cost	$
Structural	Partitioning type	Technical
Structural	Number of quanta	1
Structural	Simplicity	★★★★★
Structural	Modularity	★
Engineering	Maintainability	★
Engineering	Testability	★★
Engineering	Deployability	★
Engineering	Evolvability	★
Operational	Responsiveness	★★★
Operational	Scalability	★
Operational	Elasticity	★
Operational	Fault tolerance	★

Figure 10-6. Layered architecture characteristics ratings

Overall cost and simplicity are the primary strengths of the layered architecture style. Being monolithic, layered architectures aren't as complex as distributed architecture styles; they're simpler, easier to understand, and relatively low cost to build and maintain. Use caution, though, because these ratings diminish quickly as the monolithic layered architecture gets bigger and consequently more complex.

Neither deployability nor testability rate well for this architecture style. Deployability is low because deployments are high risk, infrequent, and involve a lot of ceremony and effort. For example, to make a simple three-line change to a class file, the entire deployment unit must be redeployed—introducing the potential for changes to the database, configuration, or other aspects of the code to sneak in alongside the original change. Furthermore, this simple three-line change is usually bundled with dozens of other changes, each of which further increases the risk and frequency of deployments. The low testability rating also reflects this scenario; with a simple three-line change, most developers are not going to spend hours executing the entire regression test suite for a simple three-line change (assuming they even have such a test suite). We give testability a two-star rating (rather than one star) because this style offers the ability to mock or stub components or even an entire layer, which eases the overall testing effort.

The engineering characteristics of the layered architecture style reflect the dynamic mentioned above: they all start well, but degrade as the size of the code base grows.

Elasticity and scalability rate very low (one star) for the layered architecture, primarily due to its monolithic deployments and lack of architectural modularity. Although it is possible to make certain functions within a monolith scale more than other functions, this effort usually requires very complex design techniques for which this architecture isn't well suited, such as multithreading, internal messaging, and other parallel processing practices. However, because a layered system's architecture quantum is always 1 (due to its monolithic UI and database and its backend processing), applications can only scale to a certain point.

Architects can achieve high responsiveness in a layered architecture with careful design, and increase it further through techniques such as caching and multithreading. We give this style three stars overall, because it does suffer from a lack of inherent parallel processing, as well as from closed layering and the Architecture Sinkhole antipattern.

When to Use

The layered architecture style is a good choice for small, simple applications or websites. It is also a good starting point for situations with very tight budget and time constraints. Because of its simplicity and its familiarity to developers and architects, this is perhaps one of the lowest-cost styles, promoting ease of development for smaller applications. The layered architecture style is also a good choice when architects are still determining whether more complex architectures would be more suitable, but must begin development.

When using this technique, keep code reuse at a minimum and keep object hierarchies (the depth of the inheritance tree) fairly shallow to maintain a good level of modularity. This will facilitate moving to another architecture style later on.

When Not to Use

As we've shown, characteristics like maintainability, agility, testability, and deployability are adversely affected as applications using the layered architecture style grow. For this reason, large applications and systems might be better off using other, more modular architecture styles.

Examples and Use Cases

The layered architecture is one of the most common architecture styles and shows up in numerous contexts.

Designers of operating systems (like Linux or Windows) tend to use layers for the same reasons application architects do—separation of concerns. Common layers in operating systems include:

Hardware layer

: Includes physical hardware like CPU, memory, and I/O devices

Kernel layer

: Provides hardware abstraction, memory management, and process scheduling

System Call Interface layer

Interacts with the kernel to provide system services

User layer

Includes applications and utilities that users interact with

The *networking Open Systems Interconnection* (OSI) model conceives of how networks should delineate responsibility. For example, the base protocol of the internet, TCP/IP, includes the following layers:

Physical layer

Physically transmits data

Data Link layer

Utilizes error detection and frame synchronization

Network layer

Handles routing (such as IP)

Transport layer

Ensures reliable data transmission (such as TCP)

Application layer

Provides services like email (SMTP), file transfer (FTP), and web browsing (HTTP)

Layered architecture promotes separation of concerns, improves maintainability, and allows for independent development of each layer, so any architecture that values these features will benefit from it.

Layered architecture is also extremely common for teams trying to achieve the architectural characteristics of *feasibility*: can we actually deliver the stated scope in the allotted time with the resources available? For example, if an organization is fueled by investors and needs to deliver something as quickly as possible, the simplicity of layered architecture often makes it a good choice, even if parts need to be rewritten later to achieve different capabilities.

[1] *Business Delegate* is a pattern designed to reduce coupling between business services and the user interface. Business delegates act as adapters to invoke business objects from the presentation tier.