Layers Pattern

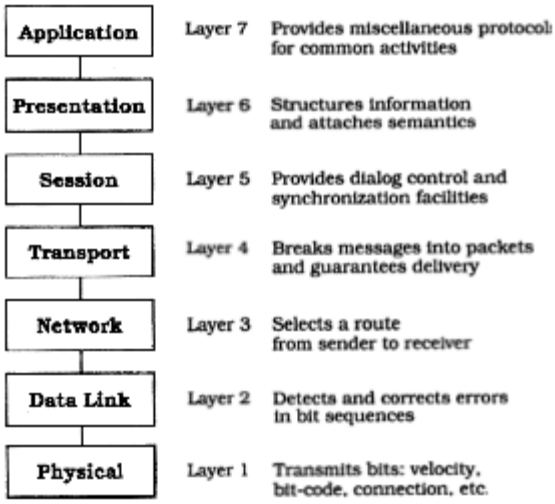
The Layers architectural pattern helps to structure applications that can be decomposed into groups of subtasks in which each group of subtasks is at a particular level of abstraction.

Example

- Networking protocols are probably the best-known example of layered architectures.
- Such a protocol consists of a set of rules and conventions that describe how computer programs communicate across machine boundaries.
- The protocol specifies agreements at a variety of abstraction levels, ranging from the details of bit transmission to high-level application logic. Therefore designers use several sub-protocols and arrange them in layers. Each layer deals with a specific aspect of communication and uses the services of the next lower layer.

Example (2)

- The International Standardization Organization (ISO) defined the following architectural model, the OSI 7-Layer Model:



Example (3)

- A layered approach is considered better practice than implementing the protocol as a monolithic block.
  - It aids development by teams and supports incremental coding and testing.
  - Using semi-independent parts also enables the easier exchange of individual parts at a later date.
  - Better implementation technologies such as new languages or algorithms can be incorporated by simply rewriting a delimited section of code.
- While ISO is an important reference model, TCP/IP is the prevalent networking protocol. TCP/IP illustrates another important benefit layering.
  - Individual layers may be reused in different contexts. TCP for example can be used 'as is' by diverse distributed applications such as telnet or ftp.

Context

- When transforming a *Domain Model* into a technical software architecture, or when realizing *Broker*, *Data Access Layer*, or *Microkernel* we must support the independent development and evolution of different system parts.
- Regardless of the interactions and coupling between different parts of a software system, there is a need to develop and evolve them independently, for example due to system size and time- to-market requirements. However, without a clear and reasoned separation of concerns in the system's software architecture, the interactions between the parts cannot be supported appropriately, nor can their independent development.

Context (2)

- The challenge is to find a balance between a design that partitions the application into meaningful, tangible parts that can be developed and deployed independently, but does not lose itself in a myriad of detail so that the architecture vision is lost and operational issues such as performance and scalability are not addressed.
- An ad hoc, monolithic design is not a feasible way to resolve the challenge. Although it allows quality of service aspects to be addressed more directly, it is likely to result in a spaghetti structure that degrades developmental qualities such as comprehensibility and maintainability.

Problem

- How should a large system be decomposed into subsystems?

Forces

- Late source code changes should not ripple through the system They should be confined to one component and not affect others.
- Interfaces should be stable, and may even be prescribed by a standards body.
- Parts of the system should be interchangeable.
  - Components should be able to be replaced by alternative implementations without affecting the rest of the system.
  - An extreme form of exchangeability might be a client component dynamically switching to a different implementation of a service that may not have been available at start-up.
  - Design for change in general is a major facilitator of graceful system evolution.

Forces (2)

- It may be necessary to build other systems at a later date with the same low level issues as the system you are currently designing.
- Similar responsibilities should be grouped to help understandability and maintainability.
  - Each component should be coherent.
  - If one component implements divergent issues its integrity may be lost.
- There is no 'standard' component granularity.
- Complex components need further decomposition.
- Crossing component boundaries may impede performance, for example when a substantial amount of data must be transferred over several boundaries, or where there are many boundaries to cross.
- The system will be built by a team of programmers, and work has to be subdivided along clear boundaries.

Forces (3)

- A key challenge is to find the ‘right’ number of layers.
  - Too few layers may not separate sufficiently the different issues in the system that can evolve independently.
  - Too many layers can fragment a software architecture into bits and pieces without a clear vision and scope, which makes it hard to evolve them at all.
  - In addition, the more layers are defined, the more levels of indirection must cross in an end-to-end control flow, which can introduce performance penalties—especially when layers are remote.

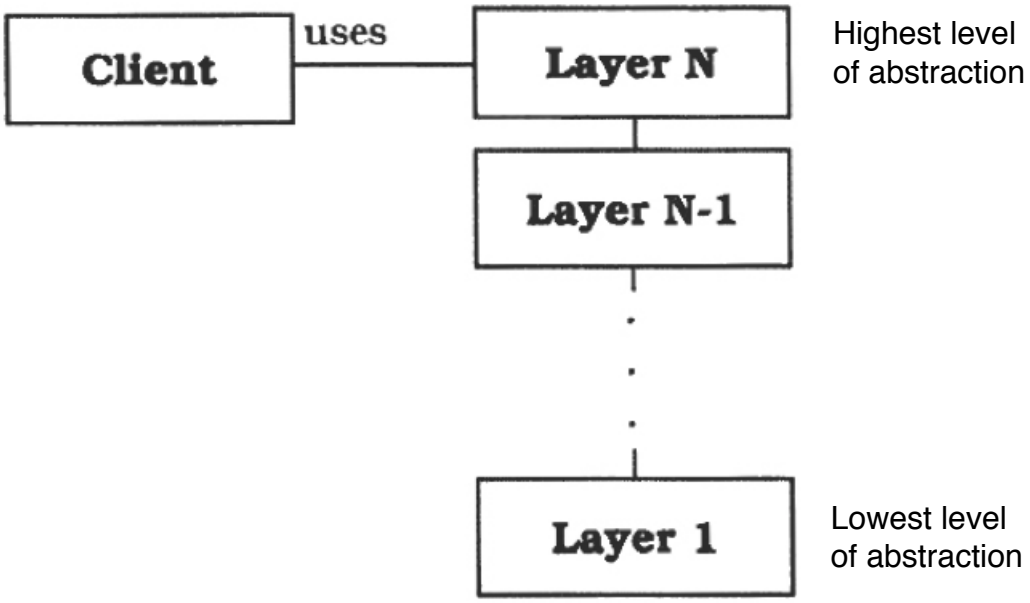
Solution

- Structure your system into an appropriate number of layers and place them on top of each other.
  - 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.
- Note that this does not prescribe the order in which to actually design layers, it just gives a conceptual view.
- It also does not prescribe whether an individual Layer J should be a complex subsystem that needs further decomposition, or whether it should just translate requests from Layer J + 1 to requests to Layer J -1 and make little contribution of its own.

Solution (2)

- It is essential that within an individual layer all constituent components work at the same level of abstraction.
- Most of the services that Layer J provides are composed of services provided by Layer J - 1.
  - Layer J provides services to layer J + 1 and delegates subtasks to layer J - 1.
- The main structural characteristic of the Layers pattern is that the services of Layer J are only used by Layer J + 1.
  - There are no further direct dependencies between layers.
  - Each individual layer shields all lower layers from direct access by higher layers.

Solution (3)



Behavior

- The following scenarios describe typical behaviors of layered applications. This does not mean that you will encounter every scenario in every application.
- Scenario 1. A client issues a request to Layer N.
  - Since Layer N cannot carry out the request on its own, it calls the next Layer N -1 for supporting subtasks.
  - Layer N -1 provides these, in the process sending further requests to Layer N-2, and so on until Layer 1 is reached. Here, the lowest-level services are finally performed.
  - If necessary, replies to the different requests are passed back up from Layer 1 to Layer 2, from Layer 2 to Layer 3, and so on until the final reply arrives at Layer N.
  - A characteristic of such top-down communication is that Layer J often translates a single request from Layer J + 1 into several requests to Layer J -1. This is due to the fact that Layer J is on a higher level of abstraction than Layer J - 1 and has to map a high-level service onto more primitive ones.

Behavior (2)

- Scenario 2. Bottom-up communication.
  - A chain of actions starts at Layer 1, for example when a device driver detects input.
  - The driver translates the input into an internal format and reports it to Layer 2, which starts interpreting it, and so on.
  - In this way data moves up through the layers until it arrives at the highest layer. While top-down information and control flow are often described as 'requests', bottom-up calls can be termed 'notifications'.
  - As mentioned in Scenario 1, one top-down request often fans out to several requests in lower layers. In contrast, several bottom-up notifications may either be condensed into a single notification higher in the structure, or remain in a 1 to 1 relationship.

Behavior (3)

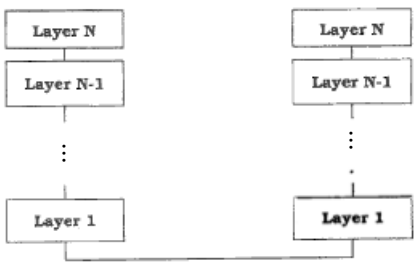
- Scenario 3. Top-down requests only travel through a subset of the layers.
  - A top-level request may only go to the next lower level N-1 if this level can satisfy the request. An example of this is where level N -1 acts as a cache, and a request from level N can be satisfied without being sent all the way down to Layer 1 and from here to a remote server.
  - Note that such caching layers maintain state information, while layers that only forward requests are often stateless.
  - Stateless layers usually have the advantage of being simpler to program, particularly with respect to reentrancy.

Behavior (4)

- Scenario 4. Bottom-up notifications only travel through a subset of the layers.
  - An event is detected in Layer 1, but stops before traveling all the way up to Layer N.
  - In a communication protocol, for example, a re- send request may arrive from an impatient client who requested data some time ago. In the meantime the server has already sent the answer, and the answer and the re-send request cross. In this case, Layer 3 of the server side may notice this and intercept the re-send request without further action.

Behavior (5)

- Scenario 5. Two stacks of N layers communicate with each other.
  - This scenario is well-known from communication protocols where the stacks. are known as 'protocol stacks'.
  - Layer N of the left stack issues a request. The request moves down through the layers until it reaches Layer 1, is sent to Layer 1 of the right stack, and there moves up through the layers of the right stack. The response to the request follows the reverse path until it arrives at Layer N of the left stack.



Design Steps (1)

- Define the abstraction criterion for grouping tasks into layers.
  - This criterion is often the conceptual distance from the platform. Sometimes you encounter other abstraction paradigms, for example the degree of customization for specific domains, or the degree of conceptual complexity.
  - For example, a chess game application may consist of the following layers, listed from bottom to top:
    - Elementary units of the game, such as a bishop.
    - Basic moves, such as castling.
    - Medium-term tactics, such as the Sicilian defense.
    - Overall game strategies.

Design Steps (2)

- Determine the number of abstraction levels according to your abstraction criterion.
  - Each abstraction level corresponds to one layer of the pattern.
  - Sometimes this mapping from abstraction levels to layers is not obvious.
  - Think about the trade-offs when deciding whether to split particular aspects into two layers or combine them into one. Having too many layers may impose unnecessary overhead, while too few layers can result in a poor structure.
- Name the layers and assign tasks to each of them.
  - The task of the highest layer is the overall system task, as perceived by the client.
  - The tasks of all other layers are to be helpers to higher layers.
  - If we take a bottom-up approach, then lower layers provide an infrastructure on which higher layers can build.

Design Steps (3)

- Specify the services.
  - The most important implementation principle is that layers are strictly separated from each other, in the sense that no component may spread over more than one layer.
  - Argument, return, and error types of functions offered by Layer J should be built-in types of the programming language, types defined in Layer J, or types taken from a shared data definition module.
  - Note that modules that are shared between layers relax the principles of strict layering.
  - It is often better to locate more services in higher layers than in lower layers. This is because developers should not have to learn a large set of slightly different low-level primitives, which may even change during concurrent development.
  - Instead the base layers should be kept 'slim' while higher layers can expand to cover a broader spectrum of applicability. This phenomenon is also called the 'inverted pyramid of reuse'.

Design Steps (4)

- Refine the layering.
  - Iterate over steps 1 to 4.
  - It is usually not possible to define an abstraction criterion precisely before thinking about the implied layers and their services.
  - Alternatively, it is usually wrong to define components and services first and later impose a layered structure on them according to their usage relationships. Since such a structure does not capture an inherent ordering principle, it is very likely that system maintenance will destroy the architecture. For example, a new component may ask for the services of more than one other layer, violating the principle of strict layering.
  - The solution is to perform the first four steps several times until a natural and stable layering evolves.
  - Like almost all other kinds of design, finding layers does not proceed in an orderly, logical way, but consists of both top-down and bottom-up steps, and certain amount of inspiration...'

Design Steps (5)

- Specify an interface for each layer.
  - If Layer J should be a 'black box' for Layer J+1, design a flat interface that offers all Layer J's services, and perhaps encapsulate this interface in a Facade object.
  - A 'white- box' approach is that in which Layer J+ 1 sees the internals of Layer J.
  - In a gray-box' approach, Layer J+ 1 is aware of the fact that Layer J consists of several components, and addresses them separately, but does not see the internal workings of individual components.
  - Good design practice tells us to use the black-box approach whenever possible because it supports system evolution better than other approaches.
  - Exceptions to this rule can be made for reasons of efficiency, or a need to access the innards of another layer.

Design Steps (6)

- Structure individual layers.
  - Traditionally, the focus was on the proper relationships between layers, but inside individual layers there was often free-wheeling chaos. When an individual layer is complex it should be broken into separate components.
  - This subdivision can be helped by using finer-grained patterns. For example, you can use the Bridge pattern to support multiple implementations of services provided by a layer.

Design Steps (7)

- Specify the communication between adjacent layers.
  - The most often used mechanism for inter-layer communication is the push model. When Layer J invokes a service of Layer J -1, any required information is passed as part of the service call. The reverse is known as the pull model and occurs when the lower layer fetches available information from the higher layer at its own discretion.
  - The Publisher-Subscriber and Pipes and Filters patterns give details about push and pull model information transfer.
  - However, such models may introduce additional dependencies between a layer and its adjacent higher layer. If you want to avoid dependencies of lower layers on higher layers introduced by the pull model, use callbacks, as described in the next step.

Design Steps (8)

- Decouple adjacent layers.
  - There are many ways to do this. Often an upper layer is aware of the next lower layer, but the lower layer is unaware of the identity of its users.
  - This implies a one-way coupling only: changes in Layer J can ignore the presence and identity of Layer J + 1 provided that the interface and semantics of the Layer J services being changed remain stable. Such a one-way coupling is perfect when requests travel top-down, as illustrated in Scenario 1, as return values are sufficient to transport the results in the reverse direction.
  - For bottom-up communication, you can use callbacks and still preserve a top-down one-way coupling. Here the upper layer registers callback functions with the lower layer. This is especially effective when only a fixed set of possible events is sent from lower to higher layers. During start-up the higher layer tells the lower layer what functions to call when specific events occur.

Design Steps (9)

- Design an error-handling strategy.
  - Error handling can be rather expensive for layered architectures with respect to processing time and, notably, programming effort.
  - An error can either be handled in the layer where it occurred or be passed to the next higher layer. In the latter case, the lower layer must transform the error into an error description meaningful to the higher layer.
  - As a rule of thumb, try to handle errors at the lowest layer possible. This prevents higher layers from being swamped with many different errors and voluminous error-handling code.
  - As a minimum, try to condense similar error types into more general error types, and only propagate these more general errors.

Variations

- Relaxed Layered System
  - This is a variant of the Layers pattern that is less restrictive about the relationship between layers.
  - In a Relaxed Layered System each layer may use the services of all layers below it, not only of the next lower layer.
  - A layer may also be partially opaque. This means that some of its services are only visible to the next higher layer, while others are visible to all higher layers.
  - The gain of flexibility and performance in a Relaxed Layered System is paid for by a loss of maintainability. This is often a high price to pay, and you should consider carefully before giving in to the demands of developers asking for shortcuts.
  - These shortcuts may be appropriate in systems that are very stable (i.e. rarely modified) and where performance is more important than maintainability.

Variations (2)

- Layering Through Inheritance
  - In this variant lower layers are implemented as base classes in object-oriented systems.
  - A higher layer requesting services from a lower layer inherits from the lower layer's implementation and hence can issue requests to the base class services.
  - An advantage of this scheme is that higher layers can modify lower-layer services according to their needs.
  - A drawback is that such an inheritance relationship closely ties the higher layer to the lower layer. If for example the definition of a base class changes, upper layers might need to be recompiled.

Layers Pattern Benefits

- Reuse of layers.
  - If an individual layer embodies a well-defined abstraction and has a well-defined and documented interface, the layer can be reused in multiple contexts.
  - However, despite the higher costs of not reusing such existing layers, developers often prefer to rewrite this functionality. They argue that the existing layer does not fit their purposes exactly and that layering would cause high performance penalties.
  - Empirical studies suggest that black-box reuse of existing layers can dramatically reduce development effort and decrease the number of defects.
- Support for standardization.
  - Clearly defined and commonly accepted levels of abstraction enable the development of standardized tasks and interfaces. Different implementations of the same interface can then be used interchangeably. This allows you to use products from different vendors in different layers.

Benefits (2)

- Dependencies are kept local.
  - Standardized interfaces between layers usually confine the effect of code changes to the layer that is changed.
  - Changes of the hardware, the operating system, the window system, special data formats and so on often affect only one layer, and you can adapt affected layers without altering the remaining layers.
  - This supports the portability of a system.
  - Testability is supported as well, since you can test particular layers independently of other components in the system.

Benefits (3)

- Exchangeability.
  - Individual layer implementations can be replaced by semantically equivalent implementations without too great an effort.
  - Hardware exchanges or additions are prime examples for illustrating exchangeability.
    - A new hardware I/0 device, for example, can be put in operation by installing the right driver program which may be a plug-in or replace an old driver program.
    - A transport medium such as Ethernet could be replaced by Token Ring. In such a case, upper layers do not need to change their interfaces, and can continue to request services from lower layers as before.
  - However, if you want to be able to switch between two layers that do not match closely in their interfaces and services, you must build an insulating layer on top of these two layers.
    - The benefit of exchangeability comes at the price of increased programming effort and possibly decreased run-time performance.

Layers Pattern Liabilities

- Lower efficiency
  - A layered architecture is usually less efficient than, say, a monolithic structure or a 'sea of objects'.
  - If high-level services in the upper layers rely heavily on the lowest layers, all relevant data must be transferred through a number of intermediate layers, and may be transformed several times.
  - The same is true of all results or error messages produced in lower levels that are passed to the highest level.
  - Communication protocols, for example, transform messages from higher levels by adding message headers and trailers.

Liabilities (2)

- Unnecessary work.
  - If some services provided by lower layers perform excessive or duplicate work not actually required by the higher layer, this has a negative impact on performance.
  - De-multiplexing in a communication protocol stack is an example of this phenomenon. Several high-level requests may cause the same incoming bit sequence to be read many times because every high-level request is interested in a different subset of the bits.
  - Another example is error correction in ftp transfer. A general purpose low-level transmission system is written first and provides a very high degree of reliability, but it can be more economical or even mandatory to build reliability into higher layers, for example by using checksums.

Liabilities (3)

- Difficulty of establishing the correct granularity of layers.
  - A layered architecture with too few layers does not fully exploit this pattern's potential for reusability, changeability and portability.
  - On the other hand, too many layers introduce unnecessary complexity and overhead in the separation of layers and the transformation of arguments and return values.
  - The decision about the granularity of layers and the assignment of tasks to layers is difficult, but is critical for the quality of the architecture.
  - A standardized architecture can only be used if the scope of potential client applications fits the defined layers.

Examples

- Information Systems (IS)
  - Systems from the business software domain often use a two-layer architecture.
    - The bottom layer is a database that holds company-specific data.
    - Many applications work concurrently on top of this database to fulfill different tasks.
  - Because the tight coupling of user interface and data representation causes its share of problems, a third layer is introduced between them – the domain layer – which models the conceptual structure of the problem domain.
  - As the top level still mixes user interface and application, this level is also split, resulting in a four-layer architecture:
    - Presentation
    - Application logic
    - Domain layer
    - Database

Examples (2)

- Virtual Machines.
  - We can speak of lower levels as a virtual machine that insulates higher levels from low-level details or varying hardware.
  - For example, the Java Virtual Machine (JVM) defines a binary code format.
  - Code written in the Java programming language is translated into a platform-neutral binary code, called byte-codes, and delivered to the JVM for interpretation.
  - The JVM itself is platform-specific. There are implementations of the JVM for different operating systems and processors.
  - Such a two-step translation process allows platform-neutral source code and the delivery of binary code not readable to humans, while maintaining platform independency.

Examples (3)

- Application Programming Interfaces (APIs)
  - An API is a layer that encapsulates lower layers of frequently-used functionality. An API is usually a flat collection of function specifications, such as the UNIX system calls. 'Flat' means here that the system calls for accessing the UNIX file system, for example, are not separated from system calls for storage allocation.
  - Above system calls we find other layers, such as the C standard library with operations like printf() or fopen().
    - These libraries provide the benefit of portability between different operating systems, and provide additional higher-level services such as output buffering or formatted output.
    - They tend to be less error prone due to the narrower gap between high-level application abstractions and library calls.
    - They often carry the liability of lower efficiency and less flexibility than conventional system calls.