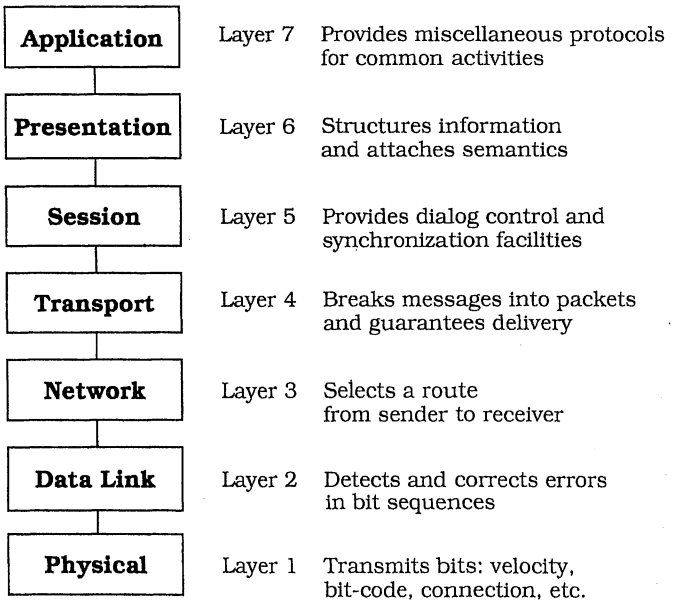
Layers

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 format, contents, and meaning of all messages are defined. All scenarios are described in detail, usually by giving sequence charts. 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 subprotocols and arrange them in layers. Each layer deals with a specific aspect of communication and uses the services of the next lower layer. The International Standardization Organization (ISO) defined the following architectural model, the OSI 7-Layer Model [Tan92]:



A layered approach is considered better practice than implementing the protocol as a monolithic block, since implementing conceptually-different issues separately reaps several benefits, for example aiding development by teams and supporting 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 OSI is an important reference model, TCP/IP, also known as the 'Internet protocol suite', is the prevalent networking protocol. We use TCP/IP to illustrate another important reason for layering: the reuse of individual layers in different contexts. TCP for example can be used 'as is' by diverse distributed applications such as `telnet` or `ftp`.

Context A large system that requires decomposition.

Problem Imagine that you are designing a system whose dominant characteristic is a mix of low- and high-level issues, where high-level operations rely on the lower-level ones. Some parts of the system handle low-level issues such as hardware traps, sensor input, reading bits from a file or electrical signals from a wire. At the other end of the spectrum there may be user-visible functionality such as the interface of a multi-user 'dungeon' game or high-level policies such as telephone billing tariffs. A typical pattern of communication flow consists of requests moving from high to low level, and answers to requests, incoming data or notification about events traveling in the opposite direction.

Such systems often also require some horizontal structuring that is orthogonal to their vertical subdivision. This is the case where several operations are on the same level of abstraction but are largely independent of each other. You can see examples of this where the word 'and' occurs in the diagram illustrating the OSI 7-layer model.

The system specification provided to you describes the high-level tasks to some extent, and specifies the target platform. Portability to other platforms is desired. Several external boundaries of the system are specified a priori, such as a functional interface to which your system must adhere. The mapping of high-level tasks onto the platform is not straightforward, mostly because they are too complex to be implemented directly using services provided by the platform.

In such a case you need to balance the following *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 exchangeable. Components should be able to be replaced by alternative implementations without affecting the rest of the system. A low-level platform may be given but may be subject to change in the future. While such fundamental changes usually require code changes and recompilation, reconfiguration of the system can also be done at run-time using an administration interface. Adjusting cache or buffer sizes are examples of such a change. 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.
- 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. Grouping and coherence are conflicting at times.
- 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—a requirement that is often overlooked at the architectural design stage.

Solution From a high-level viewpoint the solution is extremely simple. 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. It is however essential that within an individual layer all constituent components work at the same level of abstraction.

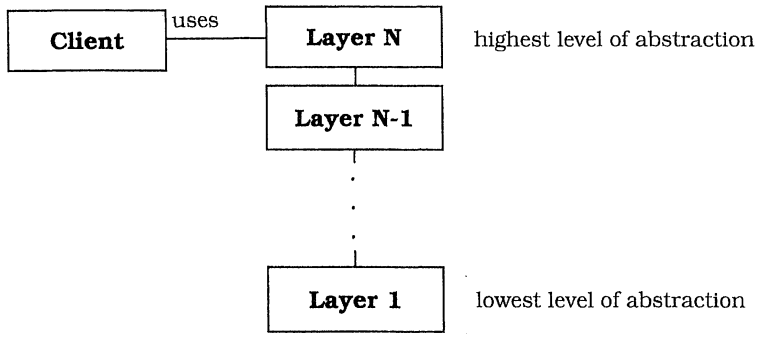
Most of the services that Layer J provides are composed of services provided by Layer J-1. In other words, the services of each layer implement a strategy for combining the services of the layer below in a meaningful way. In addition, Layer J's services may depend on other services in Layer J.

Structure An individual layer can be described by the following CRC card:

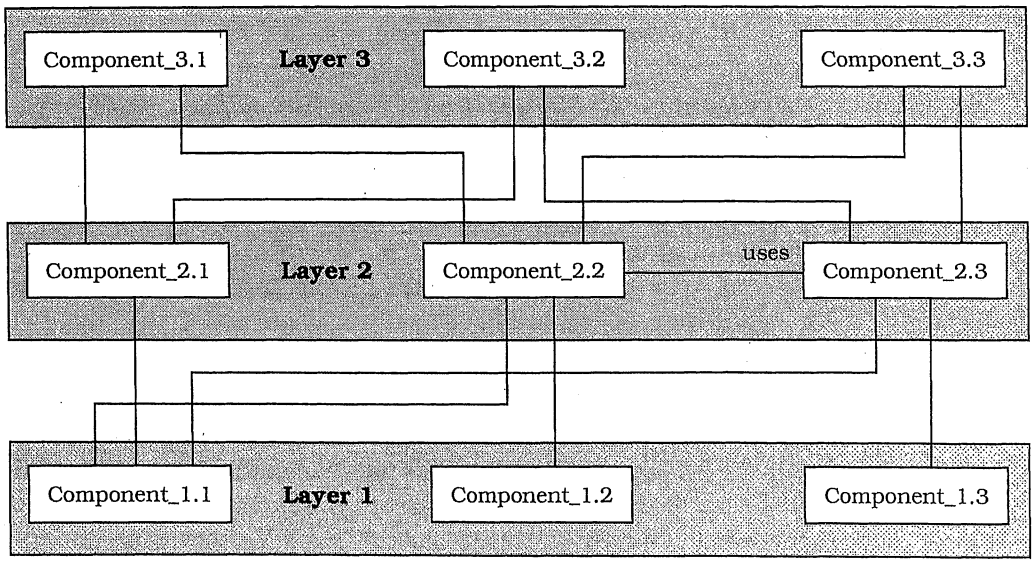
Class Layer J	***Collaborator*** • Layer J-1
Responsibility • Provides services used by Layer J+1. • Delegates subtasks to Layer J-1.	

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. This structure can be compared

with a stack, or even an onion. Each individual layer shields all lower layers from direct access by higher layers.



Examining individual layers in more detail may reveal that they are complex entities consisting of different components. In the following figure, each layer consists of three components. In the middle layer two components interact. Components in different layers call each other directly—other designs shield each layer by incorporating a unified interface. In such a design, `Component_2.1` no longer calls `Component_1.1` directly, but calls a Layer 1 interface object that forwards the request instead. In the Implementation section, we discuss the advantages and disadvantages of direct addressing.



Dynamics The following scenarios are archetypes for the dynamic behavior of layered applications. This does not mean that you will encounter every scenario in every architecture. In simple layered architectures you will only see the first scenario, but most layered applications involve Scenarios I and II. Due to space limitations we do not give object message sequence charts in this pattern.

Scenario I is probably the best-known one. 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. The example code in the Implementation section illustrates this.

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.

Scenario II illustrates 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'.

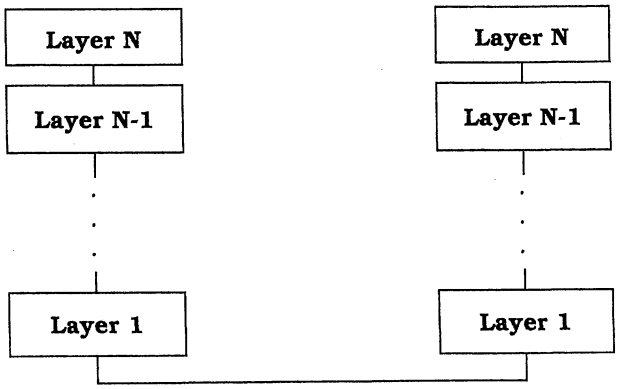
As mentioned in Scenario I, one top-down request often fans out to several requests in lower layers. In contrast, several bottom-up notifications may either be condensed into a single notification higher in the structure, or remain in a 1:1 relationship.

Scenario III describes the situation where 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 re-entrancy.

Scenario IV describes a situation similar to Scenario III. An event is detected in Layer 1, but stops at Layer 3 instead of 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.

Scenario V involves two stacks of N layers communicating with each other. This scenario is well-known from communication protocols where the stacks are known as 'protocol stacks'. In the following diagram, 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.



For more details about protocol stacks, see the Example Resolved section, where we discuss several communication protocol issues using TCP/IP as an example.