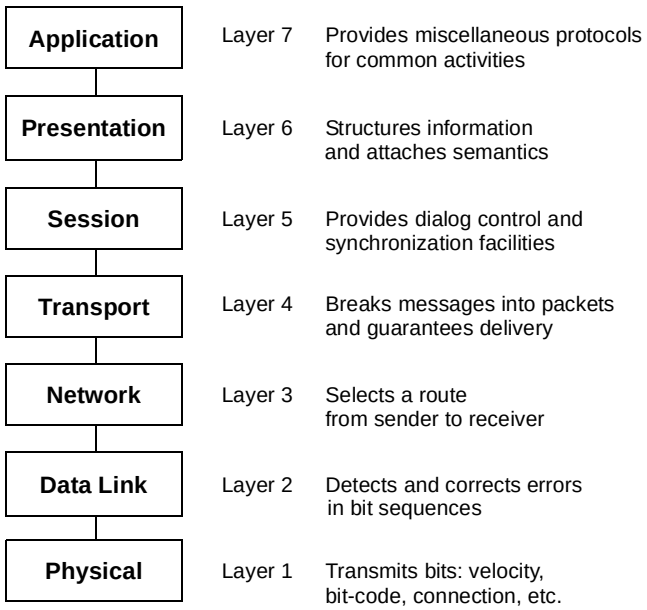
# Layers

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.

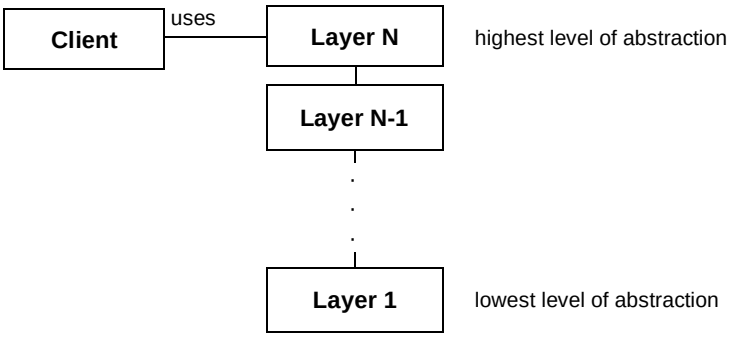
Most of the services that Layer J provides are composed of services provided by Layer J-1. In other words, the services of each layer implement a strategy for combining the services of the layer below in a meaningful way. In addition, Layer J's services may depend on other services in Layer J.

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

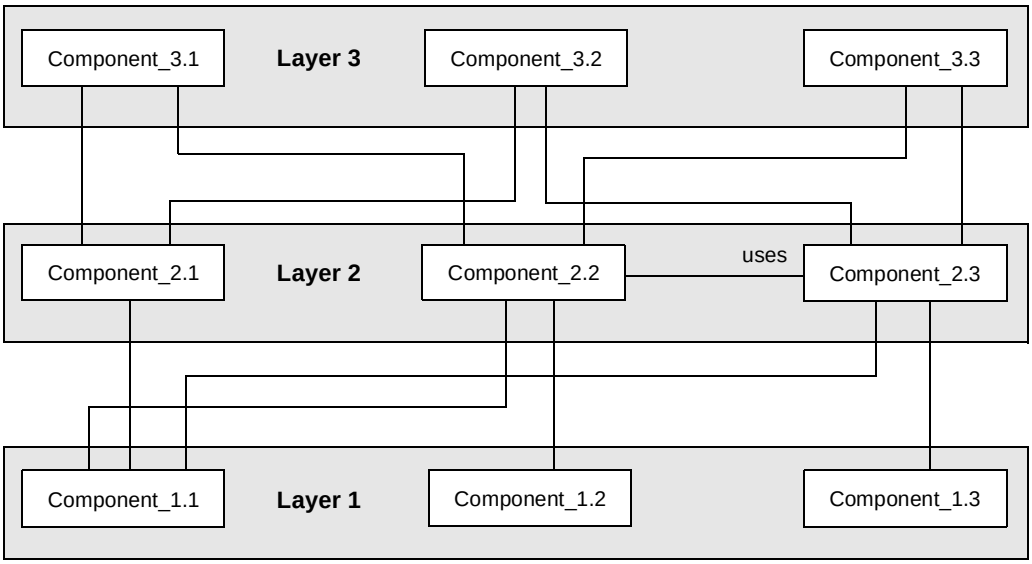
| ***Class*** Layer J | ***Collaborator*** • Layer J-1 |
|---|---|
| ***Responsibility*** • Provides services used by Layer J+1. • Delegates subtasks to Layer J-1. | |

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'.

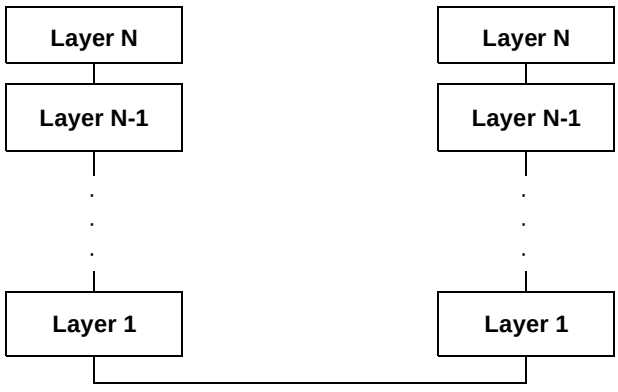
As mentioned in Scenario I, one top-down request often fans out to several requests in lower layers. In contrast, several bottom-up notifications may either be condensed into a single notification higher in the structure, or remain in a 1:1 relationship.

**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.

**Implementation**

The following steps describe a step-wise refinement approach to the definition of a layered architecture. This is not necessarily the best method for all applications—often a bottom-up or 'yo-yo' approach is better. See also the discussion in step 5.

Not all the following steps are mandatory—it depends on your application. For example, the results of several implementation steps can be heavily influenced or even strictly prescribed by a standards specification that must be followed.

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

In American Football these levels may correspond respectively to linebacker, blitz, a sequence of plays for a two-minute drill, and finally a full game plan.

In the real world of software development we often use a mix of abstraction criterions. For example, the distance from the hardware can shape the lower levels, and conceptual complexity governs the higher ones. An example layering obtained using a mixed-mode layering principle like this is as follows, ordered from top to bottom:

- User-visible elements
- Specific application modules
- Common services level
- Operating system interface level
- Operating system (being a layered system itself, or structured according to the Microkernel pattern (171))
- Hardware

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.

3 *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. However, this approach requires considerable experience and foresight in the domain to find the right abstractions for the lower layers before being able to define specific requests from higher layers.

4 *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’.

5 *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...' [Joh95]. Performing both top-down and bottom-up steps alternately is often called 'yo-yo' development, mentioned at the start of the Implementation section.

6 *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 [GHJV95]. The Known Uses section describes flat interfaces further. A 'white-box' approach is that in which Layer J+1 sees the internals of Layer J. The last figure in the Structure section shows a 'gray-box' approach, a compromise between black and white box approaches. Here Layer J+1 is aware of the fact that Layer J consists of three components, and addresses them separately, but does not see the internal workings of individual components.

Good design practise 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. The latter occurs rarely, and may be helped by the Reflection pattern (193), which supports more controlled access to the internal functioning of a component. Arguments over efficiency are debatable, especially when inlining can simply do away with a thin layer of indirection.

7 *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 [GHJV95] to support multiple implementations of services provided by a layer. The Strategy pattern [GHJV95] can support the dynamic exchange of algorithms used by a layer.

8 *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 (339) and Pipes and Filters patterns (53) 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 or message queues, as described in the next step.

9 *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. The lower layer maintains the mapping from events to callback functions in a registry. The Reactor pattern [Sch94] illustrates an object-oriented implementation of the use of callbacks in conjunction with event demultiplexing. The Command pattern [GHJV95] shows how to encapsulate callback functions into first-class objects.

You can also decouple the upper layer from the lower layer to a certain degree. Here is an example of how this can be done using object-oriented techniques. The upper layer is decoupled from specific implementation variants of the lower layer by coding the upper layer against an interface. In the following C++ code, this interface is a base class. The lower-level implementations can then be easily exchanged, even at run-time. In the example code, a Layer 2 component talks to a Level 1 provider but does not know which implementation of Layer 1 it is talking to. The ‘wiring’ of the layers is done here in the main program, but will usually be factored out into a connection-management component. The main program also takes the role of the client by calling a service in the top layer.

```
#include <iostream.h>

class L1Provider {
public:
     virtual void L1Service() = 0;
};
class L2Provider {
public:
     virtual void L2Service() = 0;
     void setLowerLayer(L1Provider *l1) {level1 = l1;}
protected:
     L1Provider *level1;
};
class L3Provider {
public:
     virtual void L3Service() = 0;
     void setLowerLayer(L2Provider *l2) {level2 = l2;}
protected:
     L2Provider *level2;
};

class DataLink : public L1Provider {
public:
     virtual void L1Service(){
          cout << "L1Service doing its job" << endl;}
};
class Transport : public L2Provider {
public:
     virtual void L2Service() {
          cout << "L2Service starting its job" << endl;
          level1->L1Service();
          cout << "L2Service finishing its job" << endl;}
};
class Session : public L3Provider {
public:
     virtual void L3Service() {
          cout << "L3Service starting its job" << endl;
          level2->L2Service();
          cout << "L3Service finishing its job" << endl;}
};

main() {
     DataLink dataLink;
     Transport transport;
     Session session;

     transport.setLowerLayer(&dataLink);
     session.setLowerLayer(&transport);

     session.L3Service();
}
```

The output of the program is as follows:

```
L3Service starting its job
L2Service starting its job
L1Service doing its job
L2Service finishing its job
L3Service finishing its job
```

For communicating stacks of layers where messages travel both up and down, it is often better explicitly to connect lower levels to higher levels. We therefore again introduce base classes, for example classes `L1Provider`, `L2Provider`, and `L3Provider`, as in the code example, and additionally `L1Parent`, `L2Parent`, and `L1Peer`. Class `L1Parent` provides the interface by which level 1 classes access the next higher layer, for example to return results, send confirmations or pass data streams. An analogous argument holds for `L2Parent`. `L1Peer` provides the interface by which a message is sent to the level 1 peer module in the other stack. A Layer 1 implementation class therefore inherits from two base classes: `L1Provider` and `L1Peer`. A second-level implementation class inherits from `L2Provider` and `L1Parent`, as it offers the services of Layer 2 and can serve as the parent of a Layer 1 object. A third-level implementation class finally inherits from `L3Provider` and `L2Parent`.

If your programming language separates inheritance and subtyping at the language level, as for example Sather [Omo93] and Java [AG96] do, the above base classes can be transformed into interfaces by pushing data into subclasses and implementing all methods there.

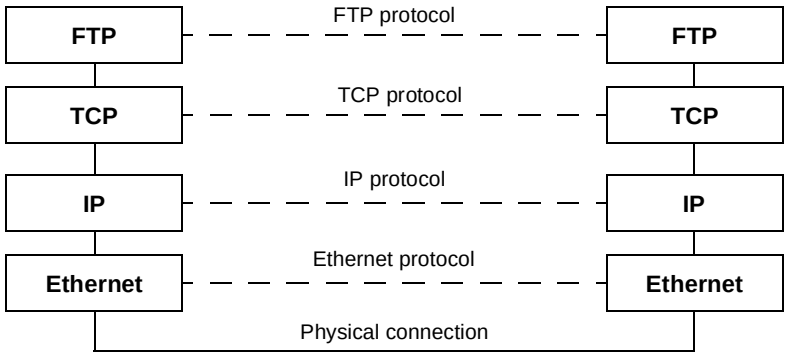
A yet other alternative for decoupling layers are message queues. They provide a complete decoupling for both top-down and bottom-up inter-layer communication. Instead of invoking services or interfaces of lower layers, or callback functions of higher layers, adjacent layers communicate solely by exchanging messages and data. The only components a layer knows are the message queues that connect it with its surrounding layers.

10 *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. If you do not do this, higher layers can be confronted with error messages that apply to lower-level abstractions that the higher layer does not understand. And who hasn't seen totally cryptic error messages being popped up to the highest layer of all—the user?

**Example Resolved**

The most widely-used communication protocol, TCP/IP, does not strictly conform to the OSI model and consists of only four layers: TCP and IP constitute the middle layers, with the application at the top and the transport medium at the bottom. A typical configuration, that for the UNIX `ftp` utility, is shown below:




TCP/IP has several interesting aspects that are relevant to our discussion. Corresponding layers communicate in a peer-to-peer fashion using a *virtual protocol*. This means that, for example, the two TCP entities send each other messages that follow a specific format. From a conceptual point of view, they communicate using the dashed line labeled 'TCP protocol' in the diagram above. We refer to this protocol as 'virtual' because in reality a TCP message traveling from left to right in the diagram is handled first by the IP entity on the left. This IP entity treats the message as a data packet, prefixes it with a header, and forwards it to the local Ethernet interface. The Ethernet interface then adds its own control information and sends the data over the physical connection. On the receiving side the local Ethernet and IP entities strip the Ethernet and IP headers respectively. The

TCP entity on the right-hand side of the diagram then receives the TCP message from its peer on the left as if it had been delivered over the dashed line.

A notable characteristic of TCP/IP and other communication protocols is that standardizing the functional interface is a secondary concern, partly driven by the fact that TCP/IP implementations from different vendors differ from each other intentionally. The vendors usually do not offer single layers, but full implementations of the protocol suite. As a result, every TCP implementation exports a fixed set of core functions but is free to offer more, for example to increase flexibility or performance. This looseness has no impact on the application developer for two reasons. Firstly, different stacks understand each other because the virtual protocols are strictly obeyed. Secondly, application developers use a layer on top of TCP, or its alternative, UDP. This upper layer has a fixed interface. Sockets and TLI are examples of such a fixed interface.

Assume that we use the Socket API on top of a TCP/IP stack. The Socket API consists of system calls such as `bind()`, `listen()` or `read()`. The Socket implementation sits conceptually on top of TCP/UDP, but uses lower layers as well, for example IP and ICMP. This violation of strict layering principles is worthwhile to tune performance, and can be justified when all the communication layers from sockets to IP are built into the OS kernel.

The behavior of the individual layers and the structure of the data packets flowing from layer to layer are much more rigidly defined in TCP/IP than the functional interface. This is because different TCP/IP stacks must understand each other—they are the workhorses of the increasingly heterogeneous Internet. The protocol rules describe exactly how a layer behaves under specific circumstances. For example, its behavior when handling an incoming re-transmit message after the original has been sent is exactly prescribed. The data packet specifications mostly concern the headers and trailers added to messages. The size of headers and trailers is specified, as well as the meaning of their subfields. In a header, for example, the protocol stack encodes information such as sender, destination, protocol used, time-out information, sequence number, and checksums. For more information on TCP/IP, see for example [Ste90]. For even more detail, study the series started in [Ste94].

**Variants**

*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. We see these shortcuts more often in infrastructure systems, such as the UNIX operating system or the X Window System, than in application software. The main reason for this is that infrastructure systems are modified less often than application systems, and their performance is usually more important than their maintainability.

*Layering Through Inheritance*. This variant can be found in some object-oriented systems and is described in [BuCa96]. In this variant lower layers are implemented as base classes. 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 data layout of a C++ base class changes, all subclasses must be recompiled. Such unintentional dependencies introduced by inheritance are also known as the *fragile base class problem*.

**Known Uses**

**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, also called *bytecodes*, 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[1], while maintaining platform-independency.

**APIs.** An Application Programming Interface 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—you can only know from the documentation to which group `open()` or `sbrk()` belong. Above system calls we find other layers, such as the C standard library [KR88] 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 often carry the liability of lower efficiency[2], and perhaps more tightly-prescribed behavior, whereas conventional system calls would give more flexibility—and more opportunities for errors and conceptual mismatches, mostly due to the wide gap between high-level application abstractions and low-level system calls.

**Information Systems (IS)** 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. Mainframe interactive systems and the much-extolled Client-Server systems often employ this architecture. 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. These are, from highest to lowest:

- Presentation
- Application logic
- Domain layer
- Database

See [Fow96] for more information on business modeling.

---

1. The Java bytecodes can be transformed into an ASCII representation that is a kind of object-oriented assembler code. This code can be read, but only with some pain!

2. Input/output buffering in higher layers is often intended to have the inverse effect—better performance than undisciplined direct use of lower-level system calls.

**Windows NT** [Cus93]**.** This operating system is structured according to the Microkernel pattern (171). The NT Executive component corresponds to the microkernel component of the Microkernel pattern. The NT Executive is a Relaxed Layered System, as described in the Variants section. It has the following layers:

- System services: the interface layer between the subsystems and the NT Executive.
- Resource management layer: this contains the modules Object Manager, Security Reference Monitor, Process Manager, I/O Manager, Virtual Memory Manager and Local Procedure Calls.
- Kernel: this takes care of basic functions such as interrupt and exception handling, multiprocessor synchronization, thread scheduling and thread dispatching.
- HAL (Hardware Abstraction Layer): this hides hardware differences between machines of different processor families.
- Hardware

Windows NT relaxes the principles of the Layers pattern because the Kernel and the I/O manager access the underlying hardware directly for reasons of efficiency.

**Consequences** The Layers pattern has several **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, layering would cause high performance penalties—and they would do a better job anyway. An empirical study hints that black-box reuse of existing layers can dramatically reduce development effort and decrease the number of defects [ZEWH95].

*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. A well-known example of a standardized interface is the POSIX programming interface [IEEE88].

*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.

*Exchangeability*. Individual layer implementations can be replaced by semantically-equivalent implementations without too great an effort. If the connections between layers are hard-wired in the code, these are updated with the names of the new layer's implementation. You can even replace an old implementation with an implementation with a different interface by using the Adapter pattern for interface adaptation [GHJV95]. The other extreme is dynamic exchange, which you can achieve by using the Bridge pattern [GHJV95], for example, and manipulating the pointer to the implementation at run-time.

Hardware exchanges or additions are prime examples for illustrating exchangeability. A new hardware I/O 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. Higher layers will not be affected by the exchange. 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.

The Layers pattern also imposes **liabilities**:

*Cascades of changing behavior.* A severe problem can occur when the behavior of a layer changes. Assume for example that we replace a 10 Megabit/sec Ethernet layer at the bottom of our networked application and instead put IP on top of 155 Megabit/sec ATM[3]. Due to limitations with I/O and memory performance, our local-end system cannot process incoming packets fast enough to keep up with ATM's high data rates. However, bandwidth-intensive applications such as medical imaging or video conferencing could benefit from the full

speed of ATM. Sending multiple data streams in parallel is a high-level solution to avoid the above limitations of lower levels. Similarly, IP routers, which forward packets within the Internet, can be layered to run on top of high-speed ATM networks via multi-CPU systems that perform IP packet processing in parallel [PST96].

In summary, higher layers can often be shielded from changes in lower layers. This allows systems to be tuned transparently by collapsing lower layers and/or replacing them with faster solutions such as hardware. The layering becomes a disadvantage if you have to do a substantial amount of rework on many layers to incorporate an apparently local change.

*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.

*Unnecessary work*. If some services performed by lower layers perform excessive or duplicate work not actually required by the higher layer, this has a negative impact on performance. Demultiplexing in a communication protocol stack is an example of this phenomenon. Several high-level requests 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 file 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. See [SRC84] for details of these trade-offs and further considerations about where to place functionality in a layered system.

---

3. ATM (Asynchronous Transfer Mode) provides much higher data rates (ranging from 155Mbps to 2.4Gbps) and functionality (such as quality of service guarantees) than conventional low-speed networks such as Ethernet and Token Ring. In addition, ATM can emulate the behavior of Ethernet in a LAN, which allows it to be integrated seamlessly into existing networks. See [HHS94] for more information on ATM.

*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 overheads 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.

**See Also** *Composite Message.* Aamod Sane and Roy Campbell [SC95b] describe an object-oriented encapsulation of messages traveling through layers. A composite message is a packet that consists of headers, payloads, and embedded packets. The Composite Message pattern is therefore a variation of the Composite pattern [GHJV95].

A *Microkernel* architecture (171) can be considered as a specialized layered architecture. See the discussion of Windows NT in the Known Uses section.

The *PAC* architectural pattern (145) also emphasizes levels of increasing abstraction. However, the overall PAC structure is a tree of PAC nodes rather than a vertical line of nodes layered on top of each other. PAC emphasizes that every node consists of three components, *presentation*, *abstraction*, and *control*, while the Layers pattern does not prescribe any subdivisions of an individual layer.

**Credits** This pattern was carefully reviewed by Paulo Villela, who highlighted many dark corners in earlier drafts. Douglas Schmidt gave valuable support in the ATM discussion.