JOT
JOURNAL OF
OBJECT TECHNOLOGY

On the Precise Semantics of the Software Layering Design Pattern

Bran Selić
Malina Software Corp., Canada

ABSTRACT Layering and layered system architectures are among the most common architectural styles in software engineering. Despite its near ubiquity, it turns out that there has been very little theoretical work on a more precise definition of what constitutes and what characterizes layering. This has had the unfortunate consequence that the layering style has been left open to frequent misinterpretation and misuse. To gain a clearer understanding of the semantics of layering, this article first delves into the origins of the concept, including a review of several key publications that popularized its application in the software domain. Through focused analysis of these foundations, the characteristics that make layering unique are derived and then used to provide a more precise definition of the semantics of layers and layer relationships. Along the way, some of the most common misconceptions associated with layered software architectures are identified and reviewed. Finally, some high-level guidance is provided to software architects and designers on when it is appropriate to use layering and how it should be applied.

KEYWORDS Layering patter, layering relationships, software architecture, virtual machines.

1. Introduction

There is a long history behind the notion of a vertical stack of layers as a metaphor for capturing some kind of directed progression of entities, starting from a foundational element at the bottom and then progressing towards more goal-oriented elements through a series of discrete levels (Figure 1). The vertical orientation of the metaphor is intended to suggest both an ordering of the individual levels as well as mutual dependence, such that each higher layer is supported by the set of layers below.

This metaphor is encountered quite frequently in software, where it has been long recognized as a significant structural design pattern (e.g., (Avgeriou & Zdun 2005; Buschmann et al. 2008; Clements et al. 2003; Hofmeister et al. 2000; Sarkar et al. 2009; Selic et al. 1994; Shaw & Garlan 1996)). Phrases such as "layered software systems", "layered architecture", or "layers of abstraction" appear frequently in numerous textbooks, technical publications, and design discussions. While there is a shared general intuition about the intent and meaning of this architectural style, to the best of the author's knowledge, there is no agreed-on definition of its precise meaning[1].

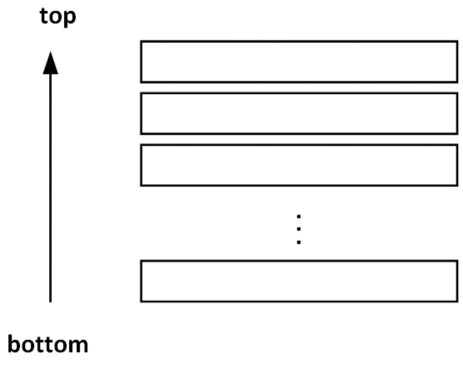

Figure 1 A generic layer stack

JOT reference format:
Bran Selić. *On the Precise Semantics of: the Software Layering Design Pattern*. Journal of Object Technology. Vol. 20, No. 2, 2021. Licensed under Attribution - NonCommercial - No Derivatives 4.0 International (CC BY-NC-ND 4.0) http://dx.doi.org/10.5381/jot.2021.20.2.a6

[1] This problem was first pointed out, to a certain extent, by David Parnas as far back as 1974, in the article "On a 'Buzzword': Hierarchical Structure" (Parnas 1978).

This has had the unfortunate consequence that there is much confusion about its semantics and how and when it should be applied in practice. A primary objective of this article is to identify and critique the various fallacies and misunderstandings that have arisen around this important architectural style. Another is to address this by providing a more precise definition of its key concepts along with an explanation of the core semantics that uniquely distinguish it from some other architectural styles that are often confused with it. We start with a review of the origins of the concepts of layers and layering in the following section. It describes several historically significant layered systems, which have had major influence in popularizing the approach. These systems are almost mythological, not only because of their huge impact on software architecture in general, but also because, over time, they have acquired numerous diverse and not necessarily accurate interpretations. It is, therefore, useful to explain what they really are, what is true about them, and what is "folklore". Starting from this foundation, a more detailed analysis of the concepts of layers and layering in software is provided in Section 3, along with a more precise definition of these concepts and a description of their semantics. Based on this, Section 4 contains a few high-level guidelines on when it is appropriate to use the layering architectural style and how it can or should be used. Section 5 provides a critical review of notable related work, although, given the sheer volume of publications that discuss layering, it is likely that some important works have been omitted. If so, any such omission is unintentional. Finally, this article concludes with a short summary of the content and a brief glimpse of what potential future work related to layering and layered system architectures.

2. Origins

A wise man once remarked: "To truly understand something, we need to know how it came to be." In that spirit, this section contains a review of several historically significant layered software architectures. However, even before that, it is very helpful to first look at where the inspiration for the layering architectural style originated.

2.1. The Basic Software-Hardware Layering Relationship

There is no doubt that the concept of software layering was inspired by the relationship that exists between computing hardware and the software that it runs. Strictly speaking, the software is not physically separable from the hardware, since it actually realized as a particular state of the storage elements of the hardware. Nevertheless, the two are separated in Figure 2. This is intended to convey the intuition that software is logically distinct from the hardware, since it is defined separately from the hardware and is not inherently a part of it. On the other hand, software requires hardware in order to achieve its purpose, so the "vertical stack" rendering is used to imply this dependency. The hardware acts as a platform for the software.
The following properties characterize the software-hardware relationship:

1. *Asymmetry*: the dependency between the layers is strictly directional; that is, the overall functioning of the upper software layer depends fundamentally (and, in this case, even existentially) on the lower hardware layer. In contrast, both the existence and implementation of the lower layer are independent of the content and structure of the upper layer.

2. *Client-server interaction*: Software realizes its functionality by selectively invoking a set of instructions provided by the hardware. The hardware then executes the invoked instructions and thereby realizes the desired behavior. This relationship can be viewed as an example of the well-known client-server paradigm, if we consider the instruction set of the hardware as a set of "services" that are invoked by the software "client".

3. *Interface-based interaction*: The instruction set "services" of the hardware are accessed by the software by referring to the instruction identifiers (i.e., a string of bits that uniquely identifies a CPU instruction) along with any necessary arguments. But the details of how the instructions are implemented by the hardware are inaccessible to the software ("information hiding").

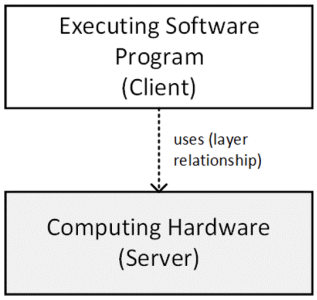

Figure 2 The Core Software-Hardware Layering Relationship

As noted later, these characteristics apply to all authentic layering relationships. To illustrate the critical role that hardware plays in the implementation of software, an alternative "onion peel" representation is often used, as shown in Figure 3. In this case, the hardware is shown as being enclosed within the software suggesting that, in effect, the hardware can be viewed as a hidden internal component of the software.

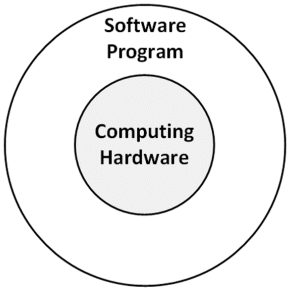

Figure 3 The "Onion-peel" Rendering of the Core Software-Hardware Layering Relationship

It is important to note that this "onion-peel" representation can be somewhat misleading. Namely, it can be easily misinterpreted to mean that the hardware is somehow "contained" within the software – which suggests that the hardware will be terminated when the program terminates. But this is clearly not the case here, particularly in situations commonly encountered in practice whereby the same hardware is shared concurrently by multiple software programs. For this reason, it is not quite appropriate to claim that the hardware constitutes a part *of the implementation of the software. Instead, it is perhaps best characterized as an independent extension of the implementation of the software*.

2.2. Dijkstra's "T.H.E" Multiprogramming System

The architecture of the T.H.E. multiprogramming system, designed and developed by Edsger Dijkstra and his team at the Technological University in Eindhoven in the 1960's (Dijkstra 1983), is probably one of the most influential works that led to the broader adoption of the layered approach to the design of complex software systems. Although the paper describing the architecture of this system does not include an explicit graphical representation[2] , it clearly suggests the 5-layer architecture shown in Figure 4.

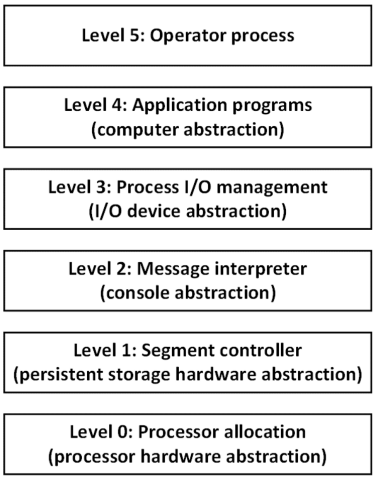

Figure 4 The layers of the "T.H.E." multiprogramming system (Dijkstra 1983)

The software in Level 0 provided core support for concurrent processes, each of which simulated a physical processor dedicated to an individual program. This included the mechanism for processor allocation (i.e., process scheduling). Level 1 was used to hide the specifics of the persistent storage hardware (a drum) by presenting them as a simple set of logical memory segments. Support for interactions with the system console (a teletype) was handled at Level 2. Level 3 handled interactions between software and any other input-output devices, allowing a unified interface model regardless of the actual type of device. The top two levels were independent of the operating system: User programs ran at Level 4, while Level 5 contained a special system program by which a system operator could manage the entire system. In essence, Levels 0 through 3 provided the desired "logical" execution environment for the programs located in Levels 4 and 5. The lower four levels jointly provide a software foundation that transforms the single underlying computer into a dynamic system that emulates a set of concurrently executing independent computers. According to (Dijkstra 1983), the primary motivation for this architecture was portability, that is, the desire to protect applications from changes and idiosyncrasies of the underlying hardware by hiding it behind a set of software layers. In addition, a graduated modular approach simplified the design and implementation and also enabled stepwise verification of the software. For example, once Level 0, was realized and independently verified, Level 1 could then be verified while taking for granted the correctness of the Level 0 – and so on. This gradual build-up of functionality not only simplified the task of verifying the correctness of the system, but it also enabled modularized concurrent development of individual levels. However, care must be taken not to interpret the hierarchy represented by Figure 4 in a strict sense. That is, *there is nothing in this design that mandates that a higher level can only access the services of the layer immediately below*. In fact, the opposite holds here, as shown in Figure 5. Thus, the concurrent process facility provided by Level 0, is used to write the software of all three higher levels of the operating system. Similarly, the logical storage service of Level 1 is used by both Levels 2 and 3. Therefore, contrary to what might be (incorrectly) concluded from Figure 4, this important precedent-setting system is not a "strict" layered architecture. Bypassing levels to access levels further down is sometimes referred to "layer jumping" and is a practice that is considered by many as incompatible with "true" layered architectures.

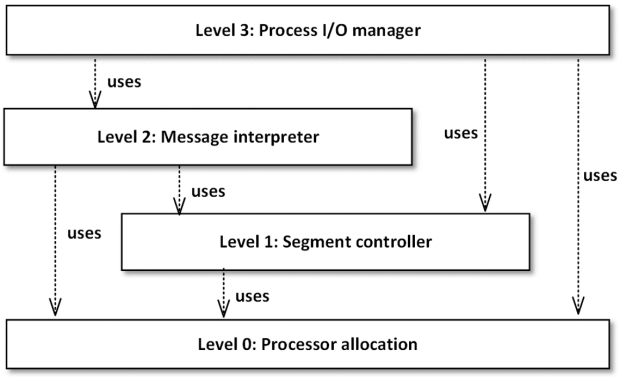

Figure 5 The uses relationships between layers of the T.H.E. system.

Despite the layer jumping, it is possible to represent this system by a system-wide vertical stack. This can be done by hiding the four operating system layers within a single *logical* layer as shown in Figure 6. It is important, however, to keep in mind that this representation is only a selective view of the

[2] Dijkstra was famous for his aversion to the use of diagrams, claiming that: "Whenever someone draws a picture to explain a program, it is a sign that something is not understood", to which Wolfram Bartussek responded: "Yes, a picture is what you draw when you are trying to understand something or trying to help someone understand" (as quoted in (Parnas & Lawton 1998)).

more complex actual layering structure.

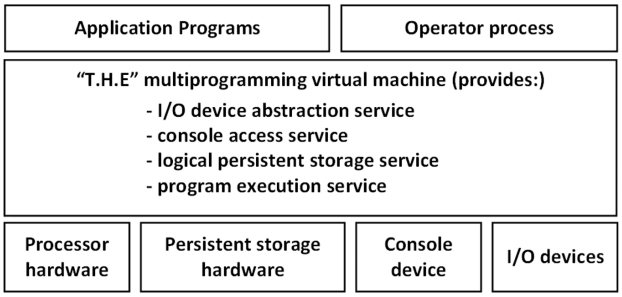

Figure 6 A more abstract representation of the architecture of the T.H.E. system.

The merged intermediate layer in Figure 6 encapsulates a complex multi-layered software system. In addition to enabling portability, it provides a customized execution environment (or platform), whose services are semantically better suited to support application-level semantics when compared to the services of the "raw" hardware. For this reason, such a layer is often referred to as a virtual machine. As explained later, this concept is central to the layering architectural style.

2.3. The Open System Interconnection (OSI) Reference Model

Another highly influential layered system is the *Open System Interconnection (OSI) reference model* (Laguë et al. 1998). This architecture was standardized by the International Standards Organization (ISO) in the early 1980s, and contains one of the most extensive descriptions of layering and layered systems. It is represented by the 7-layer stack depicted in Figure 7. The intent of this reference model was to provide a well-structured conceptual framework for organizing and realizing communications services over a physically distributed network.

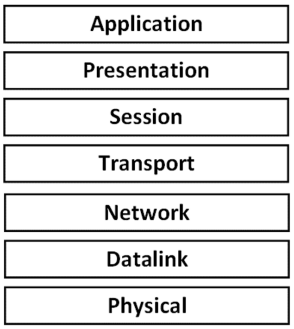

Figure 7 The ISO standard model for Open System Interconnection (OSI)

2.3.1. The Basic Model of Communications Services

In contrast to Dijkstra's T.H.E. system, this was designed to be a *strict* hierarchy of layers. That is, each layer is allowed to access only to the services of the layer directly below and none other[3]. Each layer in this model provides a different and complete set of communication-oriented services appropriate for its scope. "Complete" here means that software in one layer provides the full set of communication services needed by the layer immediately above. Consequently, there is no need for layer jumping.

At the bottom is the Physical layer, which covers the network and all the hardware needed to realize communications. Directly above it is the Datalink layer, which uses the services of the Physical layer to provide point-to-point communications between two directly linked network nodes. At this level there is still no notion of a network. That first appears at the Network layer, whose scope encompasses the full network of nodes and physical connections between them, and which provides services such as routing. The Transport layer includes mechanisms for creating and removing end-to-end connections between nodes in the network. The Session layer transforms these end-to-end chains of links into logical point-to-point connections between two or more logical connectable entities, known as communication end points (CEPs). The Presentation layer provides an extended set of services that are useful in a distributed environment, such as options for secure communications, protocol adaptation, and data compression. Finally, the Application layer may also include a "sub-layer" of application-specific services in addition to the actual communicating application elements.

The OSI standard was introduced at a time when computer-to-computer communication over networks was becoming a practical reality. Because of its conceptually clean design, because it was well documented, and because it was adopted as an international standard, it received a great deal of attention. In particular, it inspired many software system designers, who saw in it a general architectural style to be emulated for structuring complex software systems.

2.3.2. The Extended OSI Model

What is often overlooked, however, is that the OSI model deals exclusively with *communications services*. In other words, *it is not an overall system architecture,* but only an architecture for structuring distributed communication services. It says nothing about how other types of virtual machine services should be structured. For example, as explained in (Zimmermann 1980), there is another function in this model having to do with network management. This is represented by an element that is placed orthogonally relative to the 7-layer communications services stack, suggesting a separate *layering dimension*[4] (Figure 8).

Moreover, neither Figure 7 nor Figure 8 cover all the functionality needed in such a system. For example, the entire structure in Figure 8, would likely need the services of an underlying operating system. This requires yet another layering dimension, one that is independent of the layering structure of the communications services. This type of structure can be represented by the 3-dimensional diagram in Figure 9.

[3] However, it should be noted that one of the primary authors of the OSI model, Hubert Zimmermann, suggested that it should be possible to "allow bypassing of sublayers" (Zimmermann 1980). By "sublayers" he meant internal layers within layers.

[4] This notation is referred to as a "sidecar" notation (see Section 3.2.2)

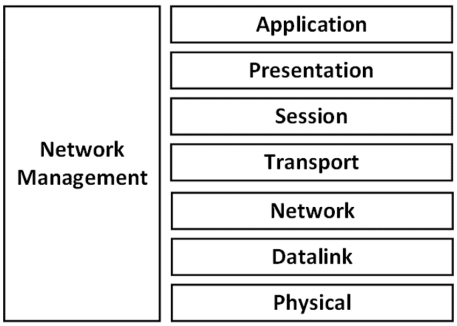

Figure 8 The OSI model, with the network management function represented by a "sidecar" notation

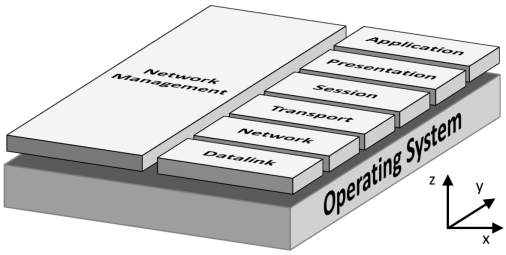

Figure 9 The multidimensional nature behind the OSI reference model

But, even this 3-D rendering is a simplification of the full layering structure of such a system. As demonstrated by Dijkstra's T.H.E. system, an operating system is likely to have its own internal layering structure, adding yet more hidden layering dimensions. This challenging issue of how to represent such systems is discussed in Section 3.2.2.

2.3.3. The OSI Service Access Point (SAP) Concept

One very important feature of the OSI model is its explicit formalization of inter-layer relationships, via the concept of a *Service Access Point* (SAP). This is an abstract representation of the client-server relationship between an entity in the upper layer and a service provisioning entity in the layer immediately below (Figure 10). SAPs were introduced in the OSI model to deal with the fact that, in communications applications, it may be necessary to distinguish between individual connection end points corresponding to different communicating entities within a given location. However, the concept is generally useful in the context of layering, since it provides an abstract model of the mechanism by which any inter-layer interactions occur[5]. In concrete cases, SAPs can be realized by a variety of technology-specific mechanisms. For example, a SAP can represent a straightforward call to an application programming interface (API) of a service in the supporting layer.

A SAP represents a directed relationship that crosses the boundary between layers. Its source end is located in the service-invoking upper layer entity, while its target end terminates on the interface of the entity that provides the service. Crucially, a SAP relationship is created and used by the *implementation* of the upper layer entity. As noted in Section 2.1, the lower layer entity can be considered as an *extension of the implementation* of the upper layer entity.

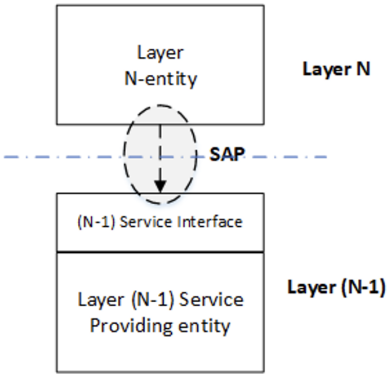

Figure 10 OSI Service Access Points

SAPs are one convenient way of describing the virtual machine nature of inter-layer relationships: the implementation code of upper layer entities initiate the interactions by invoking the appropriate services (i.e., "instruction set") of the lower layer, analogous to the way that a software program "invokes" the instructions of the CPU.

Note that although SAP-based interactions can flow in both directions, they are invariably initiated by entities in the upper layer. Thus, a response to a synchronous service request may flow from the lower layer back up to the upper layer, but that was initiated by the service request made in the upper layer. Similarly, an upper layer entity may subscribe to a service of the lower layer, which could result in a series of asynchronous upward bound communications from the lower layer. But, yet again, the initiative for this originates in the upper layer.

2.4. The UNIX Operating System

Another highly influential layered architecture is that of the UNIX operating system. It is often cited as a distinctive example of a well-structured system. The architecture of one UNIX variant, System V, is shown in Figure 11.

Note that, similar to the way that Dijkstra's T.H.E. operating system was represented as single vertical stack in Figure 4, this complex structure can also be represented in *abstract form* as a simple three-layer stack, consisting of an Application Layer, a Kernel Layer, and a Hardware layer (not shown in the diagram). However, its actual internal layering structure is more complex. It can only be folded into a simple vertical layer with loss of accuracy.

3. The Semantics of Layering

The examples discussed in the preceding section demonstrate that layering is a more complex design pattern than a simple vertical layer stack suggests. To understand the true meaning

[5] This aspect is often overlooked, particularly in numerous theoretical treatments. Namely, they often abstract out the potentially disruptive effects, such as resource sharing conflicts, that a supporting layer may have on its upper layer entities (Selic 2020). This is discussed further in Section 3.2.5.

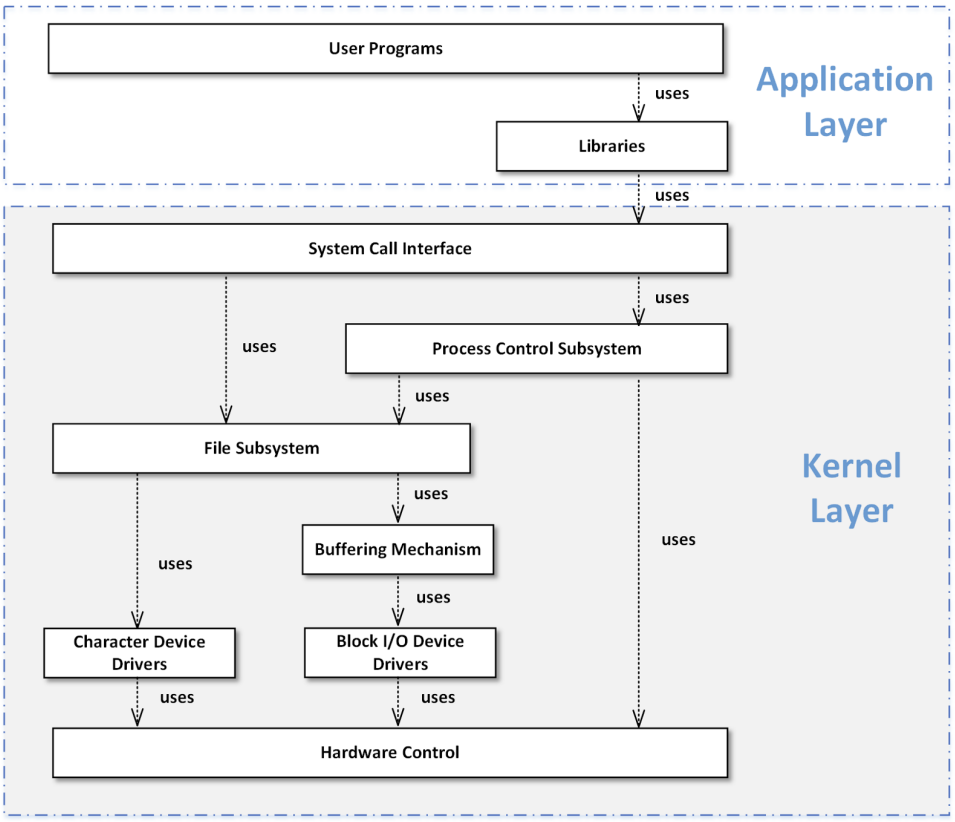

Figure 11 The layering architecture of the UNIX System V operating system (adapted from (Avgeriou & Zdun 2005)).

of layering and how to apply it, it is first necessary to provide clear and precise definitions of its key concepts. There are, in essence, just two core concepts involved: *layer* and *layer (usage) relationship*. Before defining them more precisely, it is first necessary to clarify the difference between two distinct interpretations of what layering represents.

3.1. Run-time versus Compile-time Interpretations of Layering

In the literature, layering has been discussed and interpreted in two very different contexts. The *compile-time interpretation* deals with file-system artifacts that specify the source code of the system. This covers source code modules, their associated directory structures, and their compilation dependencies. As illustrated by the simple 2-layer example in Figure 12, it is possible to get a general sense of the layering relationships by tracing through the compilation dependencies between source code files. However, not all compilation dependencies necessarily represent inter-layer relationships. For example, the uses relationships that both the Application Layer package and Services Layer package have with the Shared Source Code Definitions package (e.g., containing header file declarations), do not represent inter-layer relationships, but merely static compilation dependencies. Moreover, the compile-time interpretation is missing important dynamic information that is only known at run time, such as the number of run-time entities and the dynamics of their relationships at a given point in the execution of a program. Consequently, the compile-time view is usually an inadequate and ambiguous method of describing layered system architectures — despite the fact that it is often used and, in some cases, even recommended (Eeles 2002).

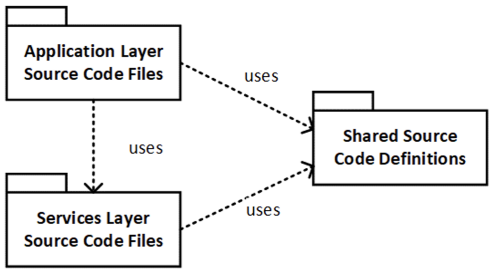

Figure 12 A compile-time representation of a simple layered system

In contrast, the run-time interpretation focuses on layering relationships between *executing software units*. This interpretation is certainly more consequential, since it represents the actual system in operation. It covers the dynamic state of a running system, the currently active objects and data structures as well as the links between them - information that can only be partially inferred from compile time specifications. The run time is what ultimately defines the actual (versus potential) behavior of a system and, therefore, it is what really matters to its users and designers. For that reason, the terms “layer” and

"layering" throughout this text refer exclusively to their run-time manifestations.

3.2. The Essential Semantics of Layer Uses Relationships

Clements et al. (Clements et al. 2003) point to the "allowed-to-use" relationship as the defining feature of layering. This is an asymmetric relation between a pair of run-time entities such that one entity can use another but not the other way around. The term "use" is defined here as a relationship whereby the correct operation of the using module depends on the correct operation of the used module. While "allowed-to-use" captures the essential asymmetric nature of layering, it is still not sufficient to fully characterize the semantics of the layering relationship. For example, consider the two systems shown in Figure 13. Both are cases of asymmetric usage, with Comp A depending on Comp B, whereas Comp B does *not* depend on Comp A. Nevertheless, in most graphical renderings, the system on the right is shown using a vertical arrangement suggesting a layering relationship between the two components, in contrast to the system on the left, where the two are drawn as "peers" situated "in the same layer". So, is there something more to layering "uses" relationship than merely a discretionary matter of graphical rendering?

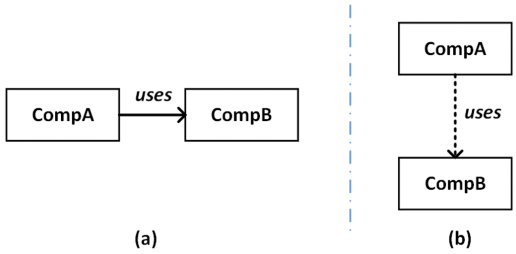

Figure 13 The two types of "uses" relationships between software modules: (a) between "peer" elements in the same layer, and (b) between elements in a layering relationship.

3.2.1. The Distinguishing Semantics of Layering

There are two fundamental semantic differences between the two systems depicted in Figure 13:

- *Existential dependence*. In the "peer" case (Figure 13(a)), Comp A is not existentially dependent on Comp B. That is, if Comp B fails or becomes unavailable for whatever reason, Comp A may not be able to function correctly, although it might still continue to execute. For instance, it could initiate a failure recovery action of some type. However, in a layering relationship (Figure 13(b)) the upper layer is existentially dependent on the lower layer, as explained in Section 2.1.
- *Implementation dependence*. In case of the "peer" relationship, the implementations of the two components are mutually independent. That is, as long as the interfaces between the two remain unchanged, each component can change its implementation without affecting the implementation of the other. In the case of layering, however, the lower layer is an extension of the implementation of the upper layer.

One important consequence of implementation dependence is that an application program can usually safely assume the existence of the lower layer. Unlike the "peer" case, in most practical situations it does not need the run-time address of the lower layer to bind to it and access its services[6]. This is because the binding of an application with its underlying virtual machine layer is typically done at compile time, with the possible exception of some distributed systems.

These two distinguishing characteristics of layer relationships, existential dependence and implementation dependence, are both encompassed by the virtual machine essence of supporting layers. Many software frameworks, such as .NET[7] or Eclipse[8] as well as the interpreters of interpreted languages such as Basic and Java, are examples of virtual machines of this kind.

3.2.2. Layering as a Multi-Dimensional Structure

The practical examples described in Section 2 all suggest that, in real-world systems, layered system architectures are rarely (if ever) accurately represented in graphic form by a simple system-wide vertical layer stack, such that each layer spans across the entire system. To deal with this unpleasant reality, various notational tricks have been attempted, such as the examples shown in Figure 14. Unfortunately, these are usually futile, because they are attempts to present what is, in the general case, a multidimensional structure by means of a limiting two- or three-dimensional representation.

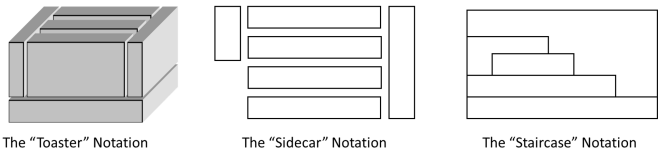

Figure 14 Different multidimensional representations of layered systems.

This is not to say that such simplified representations should be avoided, since they can be useful for didactic purposes. Such as, for example, the simplified representation of the T.H.E. system shown in Figure 6 or the OSI model in Figure 7. However, it should always be made clear that these only represent selective views of a system.

3.2.3. On the Layer Jumping "Problem"

"Layer jumping", "layer bridging" or "skip calls" all refer to the practice whereby an application in one layer bypasses the layer immediately below in order to access a service in a layer that further down. This is often considered as a serious violation of architectural integrity and something to be avoided if at all possible. However, this is based on the idealistic assumption that the entire architecture of a practical system can be realized as a single

[6] The exception might be communication services where it may be required to access a specific SAP representing a specific communication end point. But, even in those cases, what is addressed is an interface and not the layer itself.
[7] https://dotnet.microsoft.com/
[8] https://www.eclipse.org/org/

strictly vertical layer stack. But, when the multidimensional nature of layering is recognized, then this so-called "problem" often disappears. It is not a matter of bypassing layers but simply accessing a service that belongs to a different layering hierarchy, or, what is referred to here as a different layering dimension. To fully understand why, consider the example in Figure 15, which depicts a fragment of the internal structure of the Kernel Layer in the UNIX System V architecture (Figure 11).

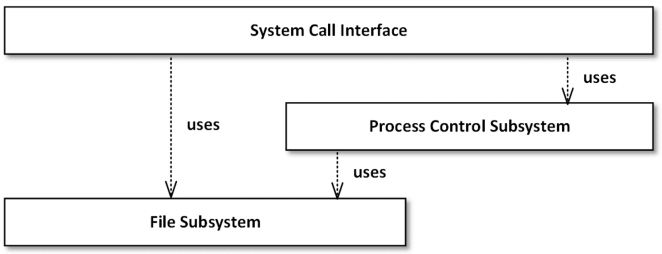

Figure 15 Fragment of the internal structure of the UNIX System V kernel layer.

If this structure was viewed as a single system-wide vertical stack, then the structure would consist of a 3-layer stack consisting of the System Call Interface on top, followed by the Process Control Subsystem, and, finally, the File Subsystem. The latter is at the bottom because it is used by the Process Control Subsystem. However, it is also used directly by the System Call Interface. This results in what appears as a case of undesirable layer jumping since it violates the strict layering rule, which dictates that a layer should only access the services of the layer below. On the other hand, we cannot put the File Subsystem and the Process Control Subsystem in a common layer, since the implementation of the Process Control Subsystem depends on the File Subsystem.

It would not make sense for the top-layer System Call Interface to access the File Subsystem services by accessing them through "pass through" interfaces so that they appear as if they are provided by the underlying Process Control Subsystem. This is illustrated by the example in Figure 16, where — to avoid layer jumping — the interface of FS Service X (a service that is actually realized and provided by the File Subsystem), is propagated upwards "through" the Process Control Subsystem. This would not only result in needless performance overhead, but, even worse, it would force the Process Control Subsystem layer to incorporate the interface of a "phantom" service that has nothing to do with its own function. This would be exacerbated if a supporting layer had to support multiple different applications with different service needs.

This is why the direct usage link from the System Call Interface to the File Subsystem should not be perceived as an architecture-corrupting "layer jump". Instead, it is a usage relationship based on a *different layering dimension* from the one to the Process Control Subsystem. The layering structure of services-providing layers should be based on the functional needs of the services themselves, as opposed to the needs of applications that use them.

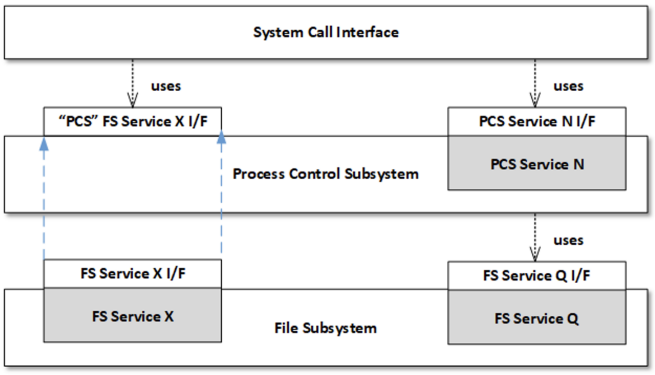

Figure 16 The problem of "pass-through" services.

3.2.4. Practical Layered Architectures The ultimate layering structure of most practical systems is a non-trivial directed acyclic graph (DAG) rather than a simple vertical stack (Figure 17). These layers may be acyclically interdependent, as was the case of the Process Control Subsystem, which uses the File Subsystem. Of course, all of these different software layering dimensions ultimately converge on the single hardware layer. A vertical service-specific virtual stack can always be extracted by tracing one top-to-bottom path through the graph. For instance, a vertical stack of this type for Service A is indicated by the shaded contour in Figure 17. As can be seen here, such "clean" representations are almost invariably just selective partial views of the full system architecture.

To be clear: system-wide layers that underlie all software do actually occur in practice. One obvious example is the hardware layer, since a major semantic and technological discontinuity occurs at the layer boundary here. Another common example is an operating system, whose "system space" interface is designed to protect it from application programming errors, such as the System Call Interface of the UNIX system.

However, both of these are exceptions rather than the rule. In practice, *an application should have the freedom to define a virtual machine that is most suitable for its purpose*. It does this by choosing a combination of different domain-specific virtual machines, each providing a subset of the services that the application needs. In the example in Figure 17, we see that the Application Layer creates its own virtual machines by means of a combination of three different virtual machines: Service A.1, Service B. 1, and Service C.1. The private internal layering structures and dependencies of these three virtual machines are not of concern to the application.

This is particularly applicable nowadays, when many software applications take advantage of various specialized open-source or proprietary software libraries that are available, many of which are designed independently. In most cases, library software specializes in providing a specific set of related utilities, such as communications service stacks or user-interface capabilities.

Finally, inappropriate layer jumping can and does occur, but this is a much less common issue than is widely recognized. Such a layer jump can can take place in case of strict service-

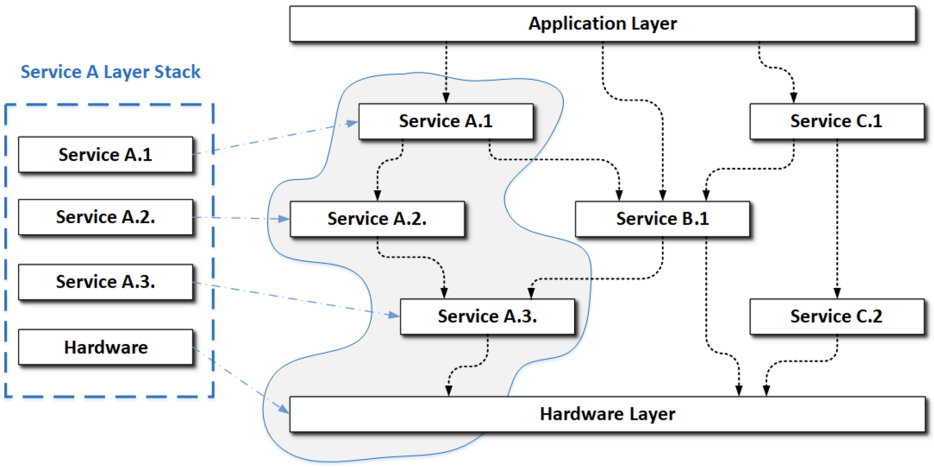

Figure 17 A realistic layered system architecture with a selective vertical stack view.

specific vertical stacks, such as the one in the OSI model. For example, if software implementing the Network layer were to directly access a Datalink layer service.

3.2.5. The Problem of Shared Virtual Machines One often overlooked problem of layers that support multiple concurrent applications is that unintended or undetected coupling can occur between the applications. For example, in Figure 17, Service A.3, Service B.1, and Service C.2 all share the same hardware layer. It may be the case that all three application take advantage of the same physical resources, such as the CPU, memory storage, internal busses, etc. In these situations unanticipated resource contention conflicts can occur, which can have a perceptible and undesirable impact on the behavior of the applications (e.g., slow response, deadlocks, buffer capacity overflow). This type of interference is often difficult to detect, not only because it depends on implementation details hidden behind layer interfaces, but also because the applications involved may have been conceived and designed independently of each other.

One potential approach for mitigating or avoiding the problem of shared implementation resources is described in (Selic 2020). It advocates clearly identifying inter-layer interaction points (i.e., SAPs) as explicit interfaces, so that the impact of shared resources can be properly accounted for when verifying a design.

3.2.6. Sidebar: On the Widely Used Term "Layers of Abstraction" Note that a software layer not only hides the hardware, but it also transforms it into a different machine. If we define abstraction as the process of generalizing or omitting detail, then, strictly speaking, a virtual machine layer on top of hardware does *not* provide a more abstract representation of the underlying hardware. Instead, it uses the hardware to realize a new and *different* machine, which hides the actual hardware.

The same reasoning applies in cases where software multiple layers are stacked one on top of the other. Each layer transforms its supporting layers into a new and different virtual machine. Consequently, it hardly makes sense to talk about "layers of abstraction", since virtual machines are not abstractions of anything[9]. Moreover, as pointed out by Parnas (Parnas 1978) and others (Clements et al. 2003), there is no ranking of abstraction here[10]. Instead, *layers of virtual machines* would be a more appropriate term.

3.3. Layering Concepts Defined

Taking into consideration the analyses of the semantics of layer relationships, we can now provide more precise (albeit informal) definitions of the core concepts associated with layering.

> *A **layer** is a logical or physical grouping of related run-time services realized either in software or hardware that jointly provide an execution environment that can be utilized for implementing one or more software applications.*

This encompasses both real physical computer layers as well as software-based virtual machines, and it emphasizes the core semantics of layers — which is why the word "implementing" is critical. The above definition is supplemented by the following definition:

> *A **layer service** is a run-time mechanism that realizes a specific utility function and which can be activated through its interface by a software application executing on top of the layer that owns the service.*

The term "activated" is used here is generic and covers different kinds of technology-specific mechanisms. Depending on the context, activation can be realized by means of synchronous API calls, asynchronous message-based communications, or implicit CPU instruction activations (for hardware layers). From a conceptual viewpoint, these services represent a logical or physical instruction set.

[9] This may be due to the fact that virtual machines are sometimes also called "abstract machines. However, in that context, "abstract" simply denotes that a virtual machine is realized by software rather than hardware.

[10] An example of true "layers of abstraction" is a progression such as: John — Male — Human — Mammal — Animal. Each level here is an abstraction of the same unique entity (John).

Note that the definition of a layer does not necessarily cover the topmost "application layer", since not all applications necessarily realize virtual machines for other applications[11].

Finally, layering relationships are defined by:

> *A **layering relationship** represents a mechanism by which an executing software application can activate one or more services of its supporting layers. Activation of a supporting layer service is exclusively controlled by the application.*

The second part of this definition captures the key asymmetrical and acyclical properties of true layering relationships.

Crucially, since a virtual machine is just a special kind of software application, layering can be applied recursively, resulting in a hierarchy of software layers. However, it is worth emphasizing here that, contrary to "common wisdom", this *potential for creating multi-layer stacks is not an essential feature of the layering architectural style*.

4. When and How to Define Layers

The primary dilemma facing software architects and designers is summarized by the two different options depicted in Figure 13. Which of the two alternatives should be used in given circumstances? When is it appropriate to define layers?

Hubert Zimmermann, one of the original authors of the OSI reference model, provided a list of 13 principles for designing layers and layered architectures (Zimmermann 1980). These include principles that specify when to provide layers and principles that explain how layers should be defined and realized. However, as noted earlier, the OSI model was dedicated exclusively to distributed communications services.

In light of the virtual machine essence of layering noted in this text, the "when to define layering" question can be reformulated as: *under what circumstances would it be effective to insert a new virtual machine between an application and the currently available platform (or platforms)?* This leads to the following *mutually inter-dependent* criteria:

- *Semantic inadequacy*: This occurs when the services provided by the available execution environment (platform) are semantically too far removed from the semantics of the core concepts of the application domain. In such case, the role of the new layer is to construct new service primitives using the services of the available platform, such that the new services directly provide the appropriate domain-specific semantics. For many simple applications, either the hardware itself or the operating system provide all the run-time services they need. However, in more complex cases, implementation can be greatly simplified and made more reliable if the corresponding program can be written using domain-specific concepts as first-class primitives.
- *Technology independence (i.e., need for portability)*: If there is a possibility that the interface of the services provided by the available platform may change or vary (e.g., by different hardware platforms), then it is appropriate to add a virtual machine layer that provides services whose interfaces are independent of such effects.
- *Reuse*: The benefits of the effort of constructing a new virtual machine are multiplied if it can be reused for multiple different applications that encompass[12] the domain supported by the virtual machine.

The matter of *how* layers should be realized, is determined by the client-server nature of the relationship between an application and its platform/virtual machine: This includes some characteristics of layers discussed previously:

- *Asymmetry*: Interactions between a virtual machine and its application layer should be initiated exclusively by action of the application. (Note, however, that this applies only to steady-state operation, since there may be cases during system start up, shut down, or failure recovery, where the virtual machine may need to initiate the action.)
- *Interface-based interactions*: All interactions between applications and a layer should occur via the interface of the layer. This interface should provide services that are well-suited for implementing applications that encompass its domain. Furthermore, the interface should be stable even if the implementation of the virtual machine might be subject to change.
- *Domain-specific services grouping*: The set of services realized by a virtual machine, should be selected based on the domain or domains of the applications it is intended to support.

From a pragmatic programming point of view, there are different ways in which virtual machines can be implemented. The most obvious and most direct is to provide interpreter-like mechanisms, which, in combination with the hardware, directly execute the upper-layer application programs. Another is to realize them through domain-specific libraries, which are bound to the application at run time. For more complex application domains, complete application frameworks, such as Eclipse or .NET, may be more appropriate.

So, when is it appropriate to create layer stacks? The answer is quite simple: when the implementation of a particular virtual machine itself needs its own domain-specific platform or platforms. After all, a virtual machine is just another application. But it is worth recalling here yet again that an application, including a virtual machine, may span multiple domains so that it may be supported by multiple virtual machines. Hence, the result will not necessarily be a single-dimensional stack, especially if some of those lower virtual machines may have layering relationships among themselves.

5. Related Work

The topics of layers and layering in software-based systems are covered in numerous textbooks as well as journal or conference publications. In particular, they are discussed in publications

[11] Although it could be fairly argued that even the top-level software layer provides a virtual machine for its users.

[12] As noted in Section 3.2.1, an application may cover more than one domain, each with its own set of services.

that deal with general software architecture (e.g., (Avgeriou & Zdun 2005; Bass et al. 2003; Buschmann et al. 2008; Clements et al. 2003; Hofmeister et al. 2000; Shaw & Garlan 1996; Vogel et al. 2011). However, in practically all cases, the topic is discussed in the context of hypothetical or highly abstracted architectures consisting of a system-wide vertical layer stack. As explained in Section 3.2.4, these rarely – if ever – accurately represent the system architectures of real-world systems. Only a few of these works delve into the core semantics of the layering relationship to distill its virtual machine essence. An early example is found in the work of Parnas on hierarchical structure (Parnas 1978), which equates layer-like structures with the concept of "abstract machine". Subsequently, Clements et al. (Clements et al. 2003) explicitly note the semantic equivalence between layers and virtual machines and focus on the "allowed-to-use" relationship. But, as argued in Section 3.2, although necessary, it is not sufficient to uniquely capture the semantics of layering.

One of the most extensive treatments of layering is provided in the work of Zimmermann (Zimmermann 1980). This was intended primarily to provide a justification of the OSI reference model and, therefore, focused almost exclusively on the communications domain. Herzberg and Broy do provide a precise and formal model of layering relationships, but, yet again, only for the specific case of distributed communications services, based on OSI-like layering (Herzberg & Broy 2005). Another effort that deals with the semantics of layering is included in the aforementioned work of Parnas (Parnas 1978). The focus here is more on layer hierarchies (i.e., layer stacks) rather than inter-layer semantics that are at the core of the virtual machine paradigm.

To the best of the author's knowledge, no prior published work has explicitly identified the two fundamental semantic differentiators of layering, which are described here in Section 3.2.1: *existence dependence* and *implementation dependence*. In addition, the multidimensional nature of layering has not been fully recognized in published literature. This is evidenced by numerous publications that treat "layer jumping" as a troublesome violation of core architectural principles (e.g., (Bourquin & Keller 2007; "ISO/IEC 7498-1:1994:1: Information Technology – Open Systems Interconnection – Basic Reference Model: The Basic Model" 1994; Sarkar et al. 2009; Zdun & Avgeriou 2005)).

The layering pattern is long-established as a significant architectural style. Despite that, no widely-used programming language includes either layers or layering as first-class language constructs. Instead, layer relationships are usually captured indirectly by means of compilation dependency directives. In Section 3.1, it was explained that these only capture a static view of what is fundamentally a run-time relationship. To overcome this, a number of research efforts have focused on attempting to derive the dynamic view of layering by static analysis of program code (e.g., (Belle et al. 2016; Bourquin & Keller 2007; Laguë et al. 1998; Sarkar et al. 2009)). This is greatly complicated by the fact that not all compilation dependencies necessarily represent layer relationships.

The author is aware of only one relatively widely used computer language that does provide direct support for layering: the Unified Modeling Language (UML) (OMG 2017). In its second major release, UML includes explicit support for a port-like layer interface inspired by the SAP concept of the OSI model. This is realized through a special Boolean attribute of the UML port concept called "isService". If the value of this attribute is set to "false"[13], the corresponding port represents a private interface that can only be accessed by the internal behavior of an object. It is used as a gateway for accessing a corresponding service of the virtual machine below. Unfortunately, the text of the standard does not sufficiently explain the purpose of this concept, so that it has mostly escaped the attention of users. The origins of this unique UML feature can be traced to the Real-time Object-Oriented Modeling (ROOM) language (Selic et al. 1994). This was one of the rare (only?) computer languages that captures the concept of layering as a first-class language construct.

6. Summary and Conclusions

The concepts of layers and layered system architectures appear frequently in numerous software engineering projects and contexts. While there is a shared general intuition about the meaning of these concepts, they are not defined with sufficient precision. This has led to some serious misunderstanding on what it actually represents, and when and how it can and should be used. For example, it is argued here that the much-discussed problem of "layer jumping" is not a real issue, but actually a misinterpretation of the essential nature of layering.

The layering architectural style was inspired by the basic relationship between hardware and software, in which the former serves as a platform supporting the latter. This relationship was generalized by early work of software pioneers such as Dijkstra and Parnas, leading to the concept of a virtual machine. A virtual machine is a software layer that stands between an application and an underlying hardware or software platform. Its purpose is to provide both portability and a customized execution environment. Such an environment can greatly simplify the implementation of an application and, consequently, increase its reliability. This is because the "instruction set" of a virtual machine comprises a set of services that directly realize key concepts from the targeted application domain.

The essential semantics of the layering relationship, which distinguish it from other similar relationships, can be reduced to just two related characteristics: existential dependence and implementation dependence of an application on its virtual machine(s). Layering is fundamentally a binary relationship between two adjacent layers, although it can be applied recursively to create layer hierarchies. Such hierarchies are typically organized around the services provided and the lower-level services needed to do that. This means that the virtual machine of a complex application will typically itself be based on multiple, possibly interdependent stacks. Consequently, contrary

[13] Unfortunately, there is a potential for confusion here; the word "service" here should not be confused with the term "service" in "service access point". A "true" value of the "isService" attribute means that the port represents a port through which an object provides services to its peer clients. In other words it is exactly the opposite of a SAP.

to widespread "folk wisdom", the architecture of a real-world software system will hardly ever be represented accurately by a single vertical stack of layers such that each one spans the full system. Instead, the actual architectural relationships in such systems are normally represented by possibly non-trivial directed acyclic graphs of virtual machines.

It is hoped that the analysis and definitions provided in this work could serve as a basis for a more systematic treatment and application of the layering concept in the future. This could replace the current highly unsystematic, semantically loose and ambiguous, and often confused application of the architectural style. One possible research direction would be to develop a more comprehensive theory of layers (i.e., when and how to use them). Another is to provide additional technological support, such as providing automated support for creating optimized configurations of virtual machine platforms. In addition, it would be useful to provide a theory on how to construct domain- and application-specific virtual machine layers — similar to what has been done with the design of language interpreters.

Acknowledgments

The author would like to thank Prof. Henry Muccini and the anonymous reviewers of this document for their most helpful comments and suggestions. In addition, the author would particularly like to express his gratitude to Prof. Alfonso Pierantonio for his substantianal and most unselfish efforts in preparing this manuscript for publication.

References

Avgeriou, P., & Zdun, U. (2005). Architectural patterns revisited-a pattern language.

Bach, M. J., et al. (1986). *The design of the unix operating system* (Vol. 5). Prentice-Hall Englewood Cliffs.

Bass, L., Clements, P., & Kazman, R. (2003). *Software architecture in practice*. Addison-Wesley Professional.

Belle, A. B., El Boussaidi, G., & Kpodjedo, S. (2016). Combining lexical and structural information to reconstruct software layers. *Information and Software Technology*, *74*, 1–16.

Bourquin, F., & Keller, R. K. (2007). High-impact refactoring based on architecture violations. In *11th european conference on software maintenance and reengineering (csmr'07)* (pp. 149–158).

Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., & Stal, M. (2008). *Pattern-oriented software architecture: A system of patterns, volume 1* (Vol. 1). John wiley & sons.

Clements, P., Garlan, D., Little, R., Nord, R., & Stafford, J. (2003). Documenting software architectures: views and beyond. In *25th international conference on software engineering, 2003. proceedings.* (pp. 740–741).

Dijkstra, E. W. (1983). The structure of "THE"-multiprogramming system. *Communications of the ACM*, *26*(1), 49–52.

Eeles, P. (2002). Layering strategies. *Rational Software White Paper, TP*, *199*(08), 01.

Herzberg, D., & Broy, M. (2005). Modeling layered distributed communication systems. *Formal Aspects of Computing*, *17*(1), 1–18.

Hofmeister, C., Nord, R., & Soni, D. (2000). *Applied software architecture*. Addison-Wesley Professional.

ISO/IEC 7498-1:1994:1: Information technology – open systems interconnection – basic reference model: The basic model. (1994).

Laguë, B., Leduc, C., Le Bon, A., Merlo, E., & Dagenais, M. (1998). An analysis framework for understanding layered software architectures. In *Proceedings. 6th international workshop on program comprehension. iwpc'98 (cat. no. 98tb100242)* (pp. 37–44).

OMG. (2017). Unified Modeling Language™ - (OMG UML), version 2.5.1, omg doc. no. formal/2017-12-05.

Parnas, D. L. (1978). On a "buzzword": hierarchical structure. In *Programming methodology* (pp. 335–342). Springer.

Parnas, D. L., & Lawton, A. (1998). *Precisely annotated hierarchical pictures of programs*. Communications Research Laboratory, McMaster University.

Parnas, D. L., & Madey, J. (1995). Functional documents for computer systems. *Science of Computer programming*, *25*(1), 41–61.

Putman, J. (2001). *Architecting with rm-odp*. Prentice Hall Professional.

Sarkar, S., Maskeri, G., & Ramachandran, S. (2009). Discovery of architectural layers and measurement of layering violations in source code. *Journal of Systems and Software*, *82*(11), 1891–1905.

Selic, B. (2020). The forgotten interfaces: A critique of component-based models of computing. *J. Object Technol.*, *19*(3), 3–1.

Selic, B., Gullekson, G., Ward, P. T., et al. (1994). *Real-time object-oriented modeling* (Vol. 2). John Wiley & Sons New York.

Shaw, M., & Garlan, D. (1996). Software architecture: perspectives on an emerging discipline. *Prentice-Hall*.

Vogel, O., Arnold, I., Chughtai, A., & Kehrer, T. (2011). *Software architecture: a comprehensive framework and guide for practitioners*. Springer Science & Business Media.

Zdun, U., & Avgeriou, P. (2005). Modeling architectural patterns using architectural primitives. *ACM SIGPLAN Notices*, *40*(10), 133–146.

Zimmermann, H. (1980). Osi reference model-the iso model of architecture for open systems interconnection. *IEEE Transactions on communications*, *28*(4), 425–432.

About the author

Bran Selić Bran Selić is President of Malina Software Corp., a Canadian company that provides consulting services to corporate clients and government institutions worldwide. He is also Director of Advanced Technology at Zeligsoft Limited in Canada, and a Visiting Scientist at Simula Research Laboratories in Norway. In 2007, Bran retired from IBM Canada, where he was an IBM Distinguished Engineer responsible for setting the strategic direction for software development tools. In addition, he is an adjunct professor at the University of Sydney (Australia) and regular lecturer at INSA (Lyon, France). With over 40 years of practical experience in designing and implementing

large-scale industrial software systems, Bran has pioneered the application of model-based engineering methods and has led the definition of several international standards in that domain, including the widely used Unified Modeling Language (UML). In 2016, he was presented with a lifetime Career Award by the steering committee of the IEEE/ACM MoDELS conference in recognition of his contributions to model-driven technologies and practice. You can contact him at selic@acm.org.