Table 13.1. Layered Pattern Solution

Overview	The layered pattern defines layers (groupings of modules that offer a cohesive set of services) and a unidirectional *allowed-to-use* relation among the layers. The pattern is usually shown graphically by stacking boxes representing layers on top of each other.
Elements	*Layer*, a kind of module. The description of a layer should define what modules the layer contains and a characterization of the cohesive set of services that the layer provides.
Relations	*Allowed to use*, which is a specialization of a more generic *depends-on* relation. The design should define what the layer usage rules are (e.g., "a layer is allowed to use any lower layer" or "a layer is allowed to use only the layer immediately below it") and any allowable exceptions.
Constraints	▪ Every piece of software is allocated to exactly one layer. ▪ There are at least two layers (but usually there are three or more). ▪ The *allowed-to-use* relations should not be circular (i.e., a lower layer cannot use a layer above).
Weaknesses	▪ The addition of layers adds up-front cost and complexity to a system. ▪ Layers contribute a performance penalty.

Table 13.4. Pipe-and-Filter Pattern Solution

Overview	Data is transformed from a system's external inputs to its external outputs through a series of transformations performed by its filters connected by pipes.
Elements	*Filter*, which is a component that transforms data read on its input port(s) to data written on its output port(s). Filters can execute concurrently with each other. Filters can incrementally transform data; that is, they can start producing output as soon as they start processing input. Important characteristics include processing rates, input/output data formats, and the transformation executed by the filter. *Pipe*, which is a connector that conveys data from a filter's output port(s) to another filter's input port(s). A pipe has a single source for its input and a single target for its output. A pipe preserves the sequence of data items, and it does not alter the data passing through. Important characteristics include buffer size, protocol of interaction, transmission speed, and format of the data that passes through a pipe.
Relations	The *attachment* relation associates the output of filters with the input of pipes and vice versa.
Constraints	Pipes connect filter output ports to filter input ports. Connected filters must agree on the type of data being passed along the connecting pipe. Specializations of the pattern may restrict the association of components to an acyclic graph or a linear sequence, sometimes called a pipeline. Other specializations may prescribe that components have certain named ports, such as the *stdin*, *stdout*, and *stderr* ports of UNIX filters.
Weaknesses	The pipe-and-filter pattern is typically not a good choice for an interactive system. Having large numbers of independent filters can add substantial amounts of computational overhead. Pipe-and-filter systems may not be appropriate for long-running computations.

CATEGORIES OF QUALITY ATTRIBUTES

- System Qualities: availability, modifiability, performance, security, testability, usability, others.
- Business Qualities: time to market, cost and benefit, product lifetime, target market, roll-out schedule, integration, others.
- Architectural Qualities: conceptual integrity, correctness and completeness.

System Quality Attributes:

- **Availability:**

The availability attribute is concerned with system failures. Faults are problems that are corrected or masked by the system. Failures are uncorrected errors that are user-visible. availability = [mean time to failure] / ([mean time to failure] + [mean time to repair])

- **Modifiability:**

The modifiability quality is concerned with what can change, when are changes made, and who makes the changes.

- **Performance:**

The performance quality is concerned with response times and similar measures for various events.

- **Security:**

  - Non-repudiation
  - Confidentiality
  - Integrity
  - Assurance or authenticity
  - Availability (no denial of service)
  - Auditing

- **Testability:**

The testability attribute is concerned with detecting failure modes. Typically, 40% of the cost of a large project is spent on testing. This means architectural support for testing that reduces test cost is time well spent. We need to control the internal state of and inputs to each unit, then observe the corresponding output of that unit.

- **Usability:**

  - How easy it is to learn the features of the system
  - How efficiently the user can use the system
  - How well the system handles user errors
  - How well the system adapts to user needs
  - To what degree the system gives the user confidence in the correctness of its actions.

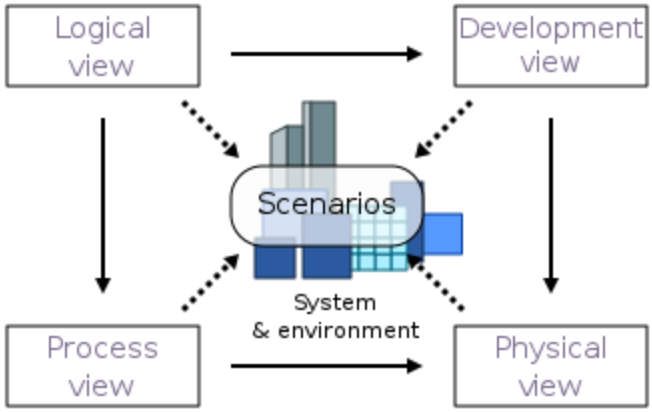
Business Quality Attributes:

- Time to Market: architectural reuse affects development time.
- Cost and Benefit: in-house architectural expertise is cheaper than outside expertise.
- Projected Lifetime of the System: long-lived systems require architectures that are modifiable and scalable.
- Targeted Market: architecture affects what platforms will be compatible and incompatible with the system.
- Roll-out Schedule: if functionality is planned to increase over time, the architecture needs to be customizable and flexible.
- Integration with Legacy Systems: the architecture of the legacy system being integrated will influence the overall system's architecture.

Architectural Quality Attributes:

- Conceptual Integrity is the underlying vision or theme unifying the components and their interactions. The architecture should do similar things in similar ways.
- Correctness and Completeness is concerned with checking the architecture for errors and omissions.
- Buildability is concerned with the organization's capabilities to actually construct the architecture in question.

4+1 ARCHITECTURAL VIEW MODEL



4+1 is a view model designed by Philippe Kruchten for "describing the architecture of software-intensive systems, based on the use of multiple, concurrent views". The views are used to describe the system from the viewpoint of different stakeholders, such as end-users, developers and project managers. The four views of the model are logical, development, process and physical view. In addition selected use cases or scenarios are used to illustrate the architecture serving as the 'plus one' view. Hence the model contains 4+1 views:

Logical view: The logical view is concerned with the functionality that the system provides to end-users. UML diagrams used to represent the logical view include, class diagrams, and state diagrams, or which shows the key abstractions in the system as objects or object classes.

Process view: The process view deals with the dynamic aspects of the system, explains the system processes and how they communicate, and focuses on the runtime behavior of the system. The process view addresses concurrency, distribution, integrators, performance, and scalability, etc. UML diagrams to represent process view include the activity diagram.

Development view: The development view illustrates a system from a programmer's perspective and is concerned with software management. This view is also known as the implementation view. It uses the UML Component diagram to describe system components. UML Diagrams used to represent the development view include the Package diagram.

Physical view: The physical view depicts the system from a system engineer's point of view. It is concerned with the topology of software components on the physical layer as well as the physical connections between these components. This view is also known as the deployment view. UML diagrams used to represent the physical view include the deployment diagram.

Scenarios: The description of an architecture is illustrated using a small set of use cases, or scenarios, which become a fifth view. The scenarios describe sequences of interactions between objects and between processes. They are used to identify architectural elements and to illustrate and validate the architecture design. They also serve as a starting point for tests of an architecture prototype. This view is also known as the use case view.

1.1.1 Software Architecture as a Design Plan

It has been said that all implemented software systems have a software architecture, regardless of whether they have explicit documentation describing it, or had a separately recognized task of designing the software architecture. Although we agree with this perspective as a philosophical statement about software architecture, we take a more pragmatic view here. In this book when we use the term *software architecture*, we are speaking of the purposeful design plan of a system.

The architect documents the architecture and makes sure it's understood by the stakeholders (at the appropriate level of detail) and by the developers.

This design plan isn't a project plan that describes activities and staffing for designing the architecture or developing the product. Instead, it is a structural plan that describes the elements of the system, how they fit together, and how they work together to fulfill the system's requirements. It is used as a blueprint during the development process, and it is also used to negotiate system requirements, and to set expectations with customers, and marketing and management personnel. The project manager uses the design plan as input to the project plan.

Having a distinct software architecture phase, with a documented architecture as the result, forces the architect or architecture team to consider the key design aspects early and across the whole system. Because the software architecture is the bridge between the system requirements and implementation, this design phase comes after the domain analysis, requirements analysis, and risk analysis, and before detailed design, coding, integration, and testing. We're not saying that the analysis tasks must end before the architecture design begins, or that later tasks have no impact on the architecture. This is just an approximate order of events, based on the primary dependencies between the tasks. There is some overlap and iteration between tasks.

The diagram in Figure 1.1 shows how the software architecture fits in with other product development tasks. In this book we don't advocate a particular software process. Instead we identify tasks based primarily on the results they produce, then describe the input needed for a task and how its results affect other tasks. We use this approach throughout the book, describing tasks and activities in terms of their dependencies.

The architect reviews requirements and negotiates them.

The analysis tasks produce the requirements for the system, and may produce things like a domain model, a requirements model, and an organizational model. Although none of these are part of the software architecture, the requirements are a key input to the software architecture design. As the archi-

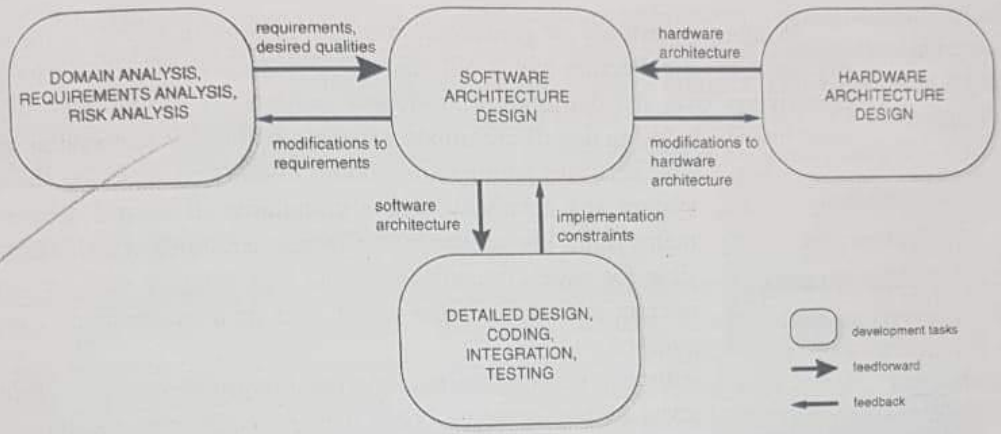


Figure 1.1. Relation of software architecture to other development tasks

The architect provides requirements to the system architect, who configures the hardware architecture.

tect or architecture team reviews the requirements and proceeds with the architecture design, they may need to return to the stakeholders to negotiate changes to the requirements.

The team also works closely with the system architect, who configures the hardware architecture. The hardware architecture is another key input to the software architecture design. The software architect or team in turn gives requirements to the system architect, and may suggest modifications as the software architecture design progresses.

The architect makes sure the architecture is followed.

The software architecture then guides the implementation tasks, including detailed design, coding, integration, and testing. Although some people consider things like coding patterns and implementation templates to be part of the architecture, we think of these code-level artifacts as implementation mechanisms, not as the software architecture. Although ideally all of the technical constraints that must be accommodated during implementation would have been anticipated and addressed in the software architecture, this isn't realistic. Inevitably there will be some unforeseen implementation constraints that cause changes to the architecture.

1.1.2 Software Architecture as an Abstraction

The other main aspect of software architecture is that it is an abstraction that helps manage complexity. The software architecture is not a comprehensive decomposition or refinement of the system: Many of the details needed to implement the system are abstracted and encapsulated within an element of the architecture.