# Software Engineering - I

An Introduction to Software Construction Techniques for Industrial Strength Software

Chapter 8 – Software and System Architecture

# Introduction

When building a house, the architect, the general contractor, the electrician, the plumber, the interior designer, and the landscaper all have different views of the structure. Although these views are pictured differently, all are inherently related: together, they describe the building's architecture. The same is true with software architecture. Architectural design basically establishes the overall structure of a software system.

The design process for identifying the sub-systems making up a system and the framework for sub-system control and communication is *architectural design*. The output of this design process is a description of the *software architecture*.

The study of software architecture is in large part a study of software structure that began in 1968 when Edsger Dijkstra pointed out that it pays to be concerned with how software is partitioned and structured, as opposed to simply programming so as to produce a correct result. Dijkstra was writing about an operating system, and first put forth the notion of a layered structure, in which programs were grouped into layers, and programs in one layer could only communicate with programs in adjoining layers. Dijkstra pointed out the elegant conceptual integrity exhibited by such an organization, with the resulting gains in development and maintenance ease.

David Parnas pressed this line of observation with his contributions concerning information-hiding modules, software structures, and program families.

A program family is a set of programs (not all of which necessarily have been or will ever be constructed) for which it is profitable or useful to consider as a group. This avoids ambiguous concepts such as "similar functionality" that sometimes arise when describing domains. For example, software engineering environments and video games are not usually considered to be in the same domain, although they might be considered members of the same program family in a discussion about tools that help build graphical user interfaces, which both happen to use.<sup>1</sup>

Parnas argued that early design decisions should be ones that will most likely remain constant across members of the program family that one may reasonably expect to produce. In the context of this discussion, an early design decision is the adoption of a particular architecture. Late design decisions should represent trivially-changeable decisions, such as the values of compile-time or even load-time constants.

All of the work in the field of software architecture may be seen as evolving towards a paradigm of software development based on principles of architecture, and for exactly the same reasons given by Dijkstra and Parnas: Structure is important, and getting the structure right carries benefits.

Before talking about the software architecture in detail, let us first look at a few of its definitions

#### What is Software Architecture?

What do we mean by software architecture? Unfortunately, there is yet no single universally accepted definition. Nor is there a shortage of proposed definition candidates. The term is interpreted and defined in many different ways. At the essence of all the discussion about software architecture, however, is a focus on reasoning about the *structural* issues of a system. And although architecture is sometimes used to mean a certain architectural style, such as client-server, and sometimes used to refer to a field of study, it is most often used to describe structural aspects of a particular system.

Before looking at the definitions for the software architecture, it is important to understand how a software system is defined. It is important because many definitions of software architecture make a reference to software systems.

According to UML 1.3, a system is a collection of connected units that are organized to accomplish a specific purpose. A system can be described by one or more models, possibly from different viewpoints. IEEE Std. 610.12-1990 defines a system as a collection of components organized to accomplish a specific function or set of functions. That is, a system is defined as an organized set of connected components to accomplish the specified tasks.

Let us now look at some of the software architecture definitions from some of the most influential writers and groups in the field.

#### **Software Architecture Definitions**

#### **UML 1.3:**

Architecture is the organizational structure of a system. An architecture can be recursively decomposed into parts that interact through interfaces, relationships that connect parts, and constraints for assembling parts. Parts that interact through interfaces include classes, components and subsystems.

## Bass, Clements, and Kazman. Software Architecture in Practice, Addison-Wesley 1997:

The software architecture of a program or computing system is the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships among them.

By "externally visible" properties, we are referring to those assumptions other components can make of a component, such as its provided services, performance characteristics, fault handling, shared resource usage, and so on. The intent of this definition is that a software architecture must abstract away some information from the system (otherwise there is no point looking at the architecture, we are

simply viewing the entire system) and yet provide enough information to be a basis for analysis, decision making, and hence risk reduction."

# Garlan and Perry, guest editorial to the *IEEE Transactions on Software Engineering*, April 1995:

Software architecture is "the structure of the components of a program/system, their interrelationships, and principles and guidelines governing their design and evolution over time."

#### **IEEE Glossary**

Architectural design: The process of defining a collection of hardware and software components and their interfaces to establish the framework for the development of a computer system.

#### **Shaw and Garlan**

The architecture of a system defines that system in terms of computational components and interactions among those components. Components are such things as clients and servers, databases, filters, and layers in a hierarchical system. Interactions among components at this level of design can be simple and familiar, such as procedure call and shared variable access. But they can also be complex and semantically rich, such as client-server protocols, database accessing protocols, asynchronous event multicast, and piped streams.

Each of these definitions of software architecture, though seemingly different, emphasizes certain structural issues and corresponding ways to describe them. It is important to understand that although they are apparently different, they do not conflict with one another.

One can thus conclude from these definitions that an architectural design is an early stage of the system design process. It represents the link between specification and design processes. It provides an overall abstract view of the solution of a problem by identifying the critical issues and explicitly documenting the design choices made under the specified constraints as its primary goal is to define the non-functional requirements of a system and define the environment. It is often carried out in parallel with some specification activities and It includes the high-level design of modular components, their relationships and organization, and provides a foundation that one can build on to solve a problem.

## Why is architecture important?

Barry Boehm says:

If a project has not achieved a system architecture, including its rationale, the project should not proceed to full-scale system development. Specifying the architecture as a deliverable enables its use throughout the development and

maintenance process.

Why is architecture important and why is it worthwhile to invest in the development of a architecture? Fundamentally, there are three reasons:

1. **Mutual communication.** Software architecture represents a common high-level abstraction of the system that most, if not all, of the system's stakeholders can use as a basis for creating mutual understanding, forming consensus, and communicating with each other.

Each stakeholder of a software system (customer, user, project manager, coder, tester, and so on) is concerned with different characteristics of the system that are affected by its architecture. Architecture provides a common language in which different concerns can be expressed, negotiated, and resolved at a level that is intellectually manageable, even for large, complex systems. Without such language, it is difficult to understand large systems sufficiently to make the early decisions that influence both quality and usefulness.

2. **Early design decisions.** Software architecture represents the embodiment of the earliest set of design decisions about a system, and these early bindings carry weight far out of proportion to their individual gravity with respect to the system's remaining development, its service in deployment, and its maintenance life. It is also the earliest point at which the system to be built can be analyzed.

An implementation exhibits an architecture if it conforms to the structural design decisions described by the architecture. This means that the implementation must be divided into the prescribed components, the components must interact with each other in the prescribed fashion, and each component must fulfill its responsibility to the other components dictated by the architecture.

Resource allocation decisions also constrain implementation. These decisions may be invisible to implementers working on individual components. The constraints permit a separation of concerns that allows management decisions that make best use of personnel and computational capacity. Component builders must be fluent in the specification of their individual components but not in architectural trade-offs. Conversely, the architects need not be experts in all aspects of algorithm design or the intricacies of the programming language, but they are the ones responsible for architectural trade-offs.

Not only does architecture prescribe the structure of the system being developed, but it also engraves that structure on the structure of the development project. The normal method of dividing up the labor in a large system is to assign different portions of the system to different groups to construct. This is called the work breakdown structure of a system. Because the system architecture includes the highest level decomposition of the system, it is typically used as the basis for the work breakdown structure. Specifically, the module structure is most often the

basis for work assignments. The work breakdown structure, in turn, dictates units of planning, scheduling, and budget, as well as inter-team communications channels, configuration control and file system organization, integration and test plans and procedures. Teams communicate with each other in terms of the interface specifications to the major components. The maintenance activity, when launched, will also reflect the software structure, with teams formed to maintain specific structural components.

A side effect of establishing the work breakdown structure is to effectively freeze the software architecture, at least at the level reflected in the work breakdown. A group that is responsible for one of the subsystems will resist having its responsibilities distributed across other groups. If these responsibilities have been formalized in a contractual relationship, changing responsibilities could become expensive. Tracking progress on a collection of tasks that is being distributed would also become much more difficult.

Thus, once the architecture's module structure has been agreed on, it becomes almost impossible for managerial and business reasons to modify it. This is one argument (among many) for carrying out extensive analysis before freezing the software architecture for a large system.

Is it possible to tell that the appropriate architectural decisions have been made (i.e., if the system will exhibit its required quality attributes) without waiting until the system is developed and deployed? If the answer were "no," choosing an architecture would be a hopeless task: random architecture selection would perform as well as any other method. Fortunately, it is possible to make quality predictions about a system based solely on an evaluation of its architecture.

3. **Reusable abstraction of a system.** Software architecture embodies a relatively small, intellectually graspable model for how the system is structured and how its components work together; this model is transferable across systems; in particular, it can be applied to other systems exhibiting similar requirements, and can promote large scale reuse.

In an architecture-based development of a product line, the architecture is in fact the sum of the early design decisions. System architects choose an architecture (or a family of closely related architectures) that will serve all envisioned members of the product line by making design decisions that apply across the family early and by making other decisions that apply only to individual members late. The architecture defines what is fixed for all members of the family and what is variable.

A family-wide design solution may not be optimal for all systems derived from it, but the quality gained and labor savings realized through architectural-level reuse may compensate for the loss of optimality in particular areas. On the other hand, reusing a family-wide design reduces the risk that a derived system might have an

inappropriate architecture. Using a standard design reduces both risk and development costs, at the risk of non-optimality.

#### **Architectural Attributes**

Software architecture must address the non-functional as well as the functional requirements of the software system. This includes performance, security, safety, availability, and maintainability. Following are some of the architectural design guidelines that can help in addressing these challenges.

#### Performance

 Performance can be enhanced by localising operations to minimise subsystem communication. That is, try to have self-contained modules as much as possible so that inter-module communication is minimized.

# Security

 Security can be improved by using a layered architecture with critical assets put in inner layers.

#### Safety

Safety-critical components should be isolated

# Availability

 Availability can be ensured by building redundancy in the system and having redundant components in the architecture.

#### Maintainability

 Maintainability is directly related with simplicity. Therefore, maintainability can be increased by using fine-grain, self-contained components.

#### Architectural design process

Just like any other design activity, design of software architecture is a creative and iterative process. This involves performing a number of activities, not necessarily in any particular order or sequence. These include system structuring, control modeling, and modular decomposition. These are elaborated in the following paragraphs.

## System structuring

System structuring is concerned with decomposing the system into interacting sub-systems. The system is decomposed into several principal sub-systems and communications between these sub-systems are identified. The architectural design is normally expressed as a block diagram presenting an overview of the system structure. More specific models showing how sub-systems share data, are distributed and interface with each other may also be developed. A sub-system is a system in its own right whose operation is independent of the services provided by other sub-systems. A module is a system component that provides services to other components but would not normally be considered as a separate system.

# Software Engineering-I (CS504)

# • Control modelling

Control modelling establishes a model of the control relationships between the different parts of the system.

# • Modular decomposition

During this activity, the identified sub-systems are decomposed into modules.

This design process is further elaborated in the following section where architectural views are discussed.

#### **Architectural Views**

Software architecture defines the high level structure of the software by putting together a number of architectural elements in an organized fashion. These elements are chosen to satisfy the functional as well as non-functional requirements of the system. Perry and Wolfe proposed the following formula for software architecture:

Software architecture = {Elements, Forms, Rationale}

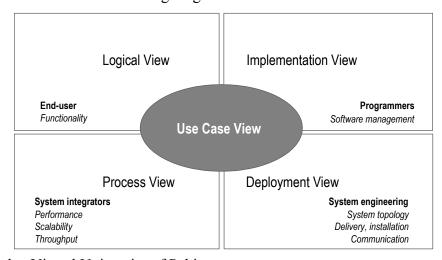
That is, a software architecture is a set of elements which have a certain form. These elements are further divided into three categories. These are: data elements, processing elements, and connecting elements. The data elements contain the information that is used and transformed in the system; the processing elements process the data elements and specify the transformation functions, and connecting elements connect different pieces of architecture together. These can themselves be data or processing elements.

This formula was modified by Boehm as follows:

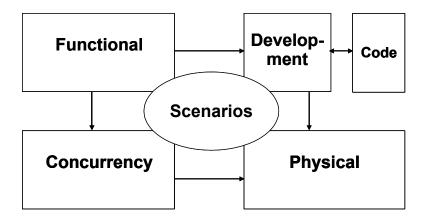
Software architecture = {Elements, Forms, Rationale/Constraints}

Krutchen proposed that the software architecture model should be composed of multiple views as a software architecture deals with abstraction, with decomposition and composition, with style and esthetics as well as requirements from different stake holders. His architectural model is known as Krutchen's 4+1 architectural view model. As evident, this model proposes the development of 5 main views namely the logical view, the process view, the physical view, the development view, and the use case view. The logical view is the object model of the design, the *process* view captures the concurrency and synchronization aspects of the design, the *physical* view documents the mapping(s) of the software onto the hardware and reflects its distributed aspect, the *development* view describes the static organization of the software in its development environment, and the use case view uses selected use cases or scenarios to validate that the architecture supports the required functionality.

This model is shown in the following diagram.



This model has been slightly modified by Clements et. al. and is shown in the following diagram.



Clement's modified version of Krutchen's 4+1 Architectural View Model

In this model, the architecture is again prepared and analyzed from 5 different perspectives. The 4 main views are Functional View, the Concurrency View, the Physical View, and the Development View. Code view in not present in the original Krutchen model and is basically an extension of the development view.

The Functional View comprises of various different functions provided by the system, key system abstraction, and the domain elements. It connects different dependencies and data flows into a single view. This view can be used by the domain engineers, product-line engineers, as well as the end users of the system. It can be used for understanding the functionality provided by the system, modifiability of the system, reusability, tool support, and allocation of work.

The Development View documents the different files and directories in the system and users of this view include the development and the configuration management staff as well as the project managers. The major uses of this view include maintenance, testing, configuration management, and version control.

The components of the Code View include classes, objects, procedures, functions, subsystems, layers, and modules. It documents the calling as well as the containing hierarchy of different components of the system. Its primary users are the programmers and designers of the system. The primary intent of developing this view is to use it for maintenance and portability.

Concurrency View intends to document different parallel processes and threads in the system. Its main focus is on event synchronization and parallel data flows. It is used

primarily by integrators, performance engineers, and testers. The main purpose of building this view is to identify ways and means to improve performance by highlighting possible opportunities for parallelism.

The Physical View depicts the physical organization of this system and how this system will be physically deployed. This includes different processors, sensors, and storage devices used by the system. This view connects various network elements and communication devices. The primary users of this system include hardware and system engineers. The view is developed with an intention to document and analyze system delivery and installation mechanism. The view is also used in understanding and analyzing issues pertaining to system performance, availability, scalability, and security.

Finally, there are Scenarios. Scenarios are basically use cases which describe the sequences of responsibilities and change cases which are changes to the system. As stated earlier, the first objective of developing a system is to fulfill the functional requirements setout for the system. These functional requirements are written in the form of use cases. Therefore, scenarios are used to understand and validate the system. Their secondary purpose is to communicate the design to the other users of the system. Scenarios tie all the view together into an integrated system. They are also used to understand the dynamic behavior of the system and understand the limits of design.

This is summarized in the following table:

View	Components	Users	Rationale
Functional View	functions, key system abstractions,	domain engineers, product-line	functionality, modifiability, product
	domain elements	designers, end users	lines/reusability, tool support, work allocation
Code View	classes, objects, procedures, functions, subsystems, layers, modules	programmers, designers, reusers	modifiability/maintainabi lity, portability, subsetability
Development View	files, directories	managers, programmers, configuration managers	managers, programmers, configuration managers
Physical View	CPUs, sensors, storage	hardware engineers, system engineers	system delivery and installation, performance, availability, scalability, security
Concurrency View	processes, threads	performance engineers, integrators, testers	performance, availability

# What Are Views Used For?

Views are an engineering tool to help achieve desired system qualities. Each view provides an engineering handle on certain quality attributes. In some systems, distinct views collapse into one (e.g., the concurrency and physical views may be the same for small systems.). Views are also used as documentation vehicle for current development and future development. Users of views include both managers and customers. For these reasons views must be *annotated* to support analysis. This annotation can be achieved with the help of scenarios and design *rationale*.

Structures can also be used to document how the current system was developed and how future development should occur. This information is needed by managers and customers.

Although one often thinks about a system's structure in terms of functionality, there are other system properties in addition to functionality (such as physical distribution, process communication, and synchronization) that must be reasoned about at an architectural level. Each structure provides a method for reasoning about some of the relevant quality attributes. The uses structure, for instance, must be <u>engineered</u> (not merely recorded) to build a system that can easily be extended or contracted. The calls structure is <u>engineered</u> to reduce bottlenecks. The module structure is <u>engineered</u> to produce modifiable systems, and so on. Each structure provides a different view into the system and a different leverage point for design to achieve desired qualities in the system.

# **Hierarchical Views**

Every view is potentially hierarchical, e.g. *functional view could contain* sub-functions. Similarly *development view* contains directories which contain files. Code view would have modules and systems that contain sub-modules and sub-systems respectively. Concurrency view contains processes that are further subdivided into threads. Finally, *physical view* clusters contain computers which contain processors.

Architectural views are related to each other in complicated ways. One has to choose the views that are useful to the system being built and to the achievement of qualities that are important for that particular application. The architectural views should be hierarchical (where needed) and should contain enough annotated information to support desired analyses.

## Architectural styles

The architectural model of a system may conform to a generic architectural model or style. An awareness of these styles can simplify the problem of defining system architectures. However, most large systems are heterogeneous and do not follow a single architectural style.

Each style describes a system category that encompasses:

- (1) a set of components (e.g., a database, computational modules) that perform a function required by a system,
- (2) a set of connectors that enable "communication, coordination and cooperation" among components,
- (3) constraints that define how components can be integrated to form the system, and
- (4) semantic models that enable a designer to understand the overall properties of a system by analyzing the known properties of its constituent parts.

#### **Architectural models**

Like analysis models, many different kinds of architectural models are developed during the architectural design process. Static structural model shows the major system components while a dynamic process model shows the process structure of the system. Interface models are developed to define sub-system interfaces.

#### **Architectural Styles**

Architectural design may be based upon a certain pattern of model. These different patterns are also called architectural styles. These styles have different characteristics and attributes and can be useful to solve problems related to a particular situation of requirement. Among the many styles, the most commonly practiced are the following:

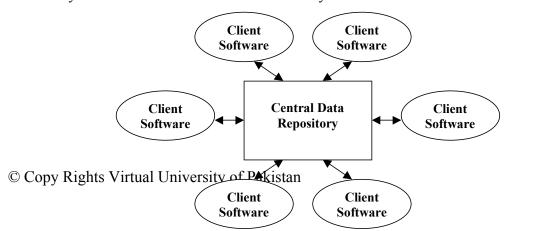
- Data-centered architectures
- Client Server Architecture and its variations
- Layered architectures
- Reference Architecture

#### **Data-Centered or the repository model**

In any system, sub-systems need to exchange information and data. This may be done in two ways:

- 1. Shared data is held in a central database or repository and may be accessed by all subsystems
- 2. Each sub-system maintains its own database and passes data explicitly to other subsystems

When large amounts of data are to be shared, the repository model of sharing is most commonly used. This model has been extensively used in the main-frame based



13

application. The model is depicted in the following diagram:

# Repository model characteristics

# Advantages

Repository model is an efficient way to share large amounts of data. In this case sub-systems need not be concerned with how data is produced Centralised management e.g. backup, security, etc. This model also provides a global view of the system and the sharing model is published as the repository schema.

# Disadvantages

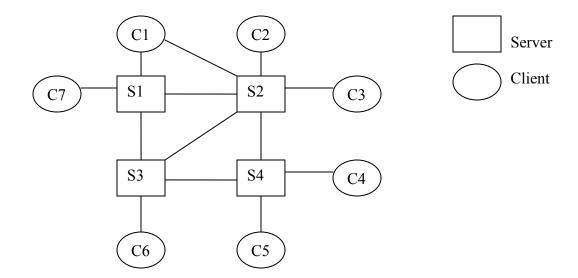
Repository model suffers from a number of disadvantages. First of all, subsystems must agree on a repository data model. This inevitably leads to a compromise. The major problem however is that data evolution is difficult and expensive. There is also little scope for implementing specific management policies. It is also difficult to distribute efficiently

#### **Client-server model**

Client server model tries to distribute data and processing. This is a shift from main-frame based applications where both the data management and the processing of data used to be typically carried out by the same main-frame computer. In those applications, the user interface used to be a provided through a "dumb" terminal which did not have much processing power. With the availability of the cheaper but power machines, it was possible to shift some load from the back-end computer to other smaller machines.

The client-server model is a distributed system model which shows how data and processing is distributed across a range of components. In this model, the application is modeled as a set of services that are provided by servers and a set of clients that use these services. The system is organized as a set of stand-alone servers which provide specific services such as printing, data management, etc. and a set of clients which call on these services. These clients and servers are connected through a network which allows clients to access servers. Clients and servers are logical processes (not always physical machines). Clients know the servers but the servers do not need to know all the clients and the mapping of processes to processors is not always 1:1.

The following diagram depicts a general client-server organization.



#### **Client/Server Software Components**

A typical client-server architecture based system is composed of a number of different components. These include user interaction/presentation subsystem, application subsystem, database management subsystem, and middleware. The application subsystem implements requirements defined by the application within the context of the operating environment. In this case the application components may reside on either client or server side. Middleware provides the mechanism and protocols to connect clients with the servers.

#### Representative Client/Server Systems

Following are some of the representative server types in a client-server systems.

• File servers

In this case, client requests selected records from a file and the server transmits records to client over the network.

Database servers

In this case, client sends requests, such as SQL queries, to the database server, the server processes the request and returns the results to the client over the network

Transaction servers

In this configuration, client sends requests that invokes remote procedures on the server side, server executes procedures invoked and returns the results to the client.

Groupware servers

Groupware servers provide set of applications that enable communication among clients using text, images, bulletin boards, video, etc.

#### Client-server characteristics

#### Advantages

The main advantage of the client-server architecture is that it makes effective use of networked systems. Instead of building the system with very expensive large computers and hardware devices, cheaper hardware may be used to gain performance and scalability. In addition, adding new servers or upgrading existing servers becomes easier. Finally, distribution of data is straightforward in this case.

#### Disadvantages

The main disadvantage of this model is that there is no standard way of sharing data, so sub-systems may use different data organisation. Hence data interchange may be inefficient. From a management point of view, each server needs attention and hence there is redundant management in each server. Finally there is no central register of names and services - it may be hard to find out what servers and services are available.

#### **Representative Client/Server Configurations**

The client-server model can have many different configurations. In the following sections, we look at some of these configurations.

#### **Thin Client Model**

This model was initially used to migrate legacy systems to client server architectures. In this case the legacy system may act as a server in its own right and the GUI may be implemented on a client. It chief disadvantage is that it places a heavy processing load on both the server and the network.

#### **Fat Client Model**

With advent of cheaper and more powerful hardware, people thought of using the processing power of client side machines. So the fat client model came into being. In this model, more processing is delegated to the client as the application processing is locally extended. It is suitable for new client/server systems when the client system capabilities are known in advance. It however is more complex than thin client model with respect to management issues. Since the client machine now also has a significant part of the application resident on it, new versions of each application need to be installed on every client.

#### Zero Install

As discussed earlier, fat-client architecture posed major challenges in terms of installation and maintenance of the client side of the application, especially when there are large number of client machines. So the idea behind zero install architecture is to develop a system where no installation on the client side is needed. This can only be done when there is no or little processing done at the client side. This is basically a trade-off between using the computing power available at the client machine versus maintenance overhead. This in essence takes us back to an architecture which is similar to thin-client architecture. There is little difference though. In the classical thin-client architecture, the entire processing is carried-out by a single server, in the case of zero-install, the network environment is used to distribute server side processing by adding a number of servers which share processing load. Web-based application where the web-pages are put on a web-server is an example of this type of architecture. In this case, whenever there is a change, the web page is updated accordingly. Now when the user logs in, he gets to use the modified version of the application without any changes or updates at the client side.

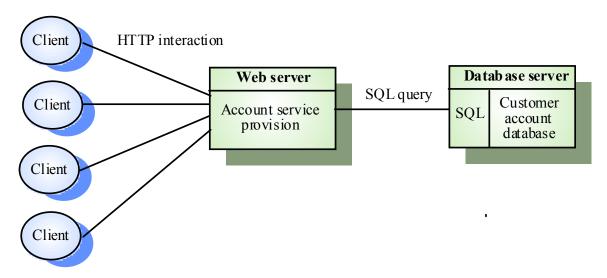
#### N-Tier architecture

N-tier architecture stems from the struggle to find a middle ground between the fatclient architecture and the thin-client architecture. In this case the idea is to enhance scalability and performance by distributing both the data and the application using multiple server machines. This could involve different types of servers such as application server, web server, and DB server. Three-tier architecture which is explained below is a specialized form of this architecture.

#### **Three-tier Architecture**

In this architecture, each application architecture layers (presentation, application, database) may run on separate processors. It therefore allows for better performance than a thin-client approach. It is simpler to manage than fat client approach and highly scalable (as demands increase add more servers).

A typical 3-tier architecture is depicted in the following diagram.

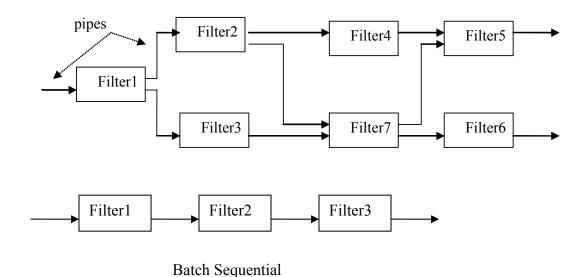


N-tier architecture generalizes the concepts of 3-tier architecture. In this case the system architecture may have more than 3 layers. That is, in n-tier architecture, in order to increase performance, we may distribute the application over different servers by putting different subsystems on different servers.

## **Data Flow or Pipes and Filters Architecture**

This architecture is very similar to data flow diagrams. This is used when the input data is processed through a series of transformations to yield the desired output. It is also known as pipes and filters architecture where each processing step is called a filter and the connecting link from one process to the other is called the pipe through which the information flows from one process to the other. An important aspect of this model is that each filter works independently of others and does not require knowledge of the working of any of the other filters including its neighbours.

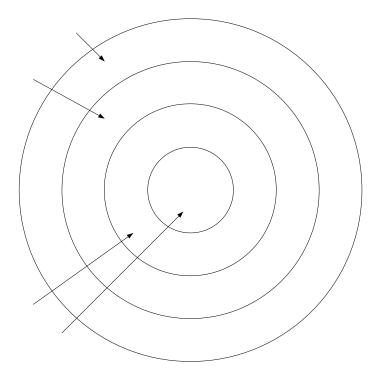
If the dataflow has only a single sequence of processes with no alternative or parallel paths, then it is called batch sequential. These models are depicted in the following diagrams.



© Copy Rights Virtual University of Pakistan

## **Layered Architecture**

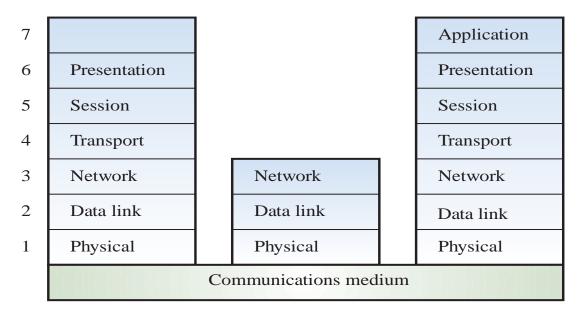
As the name suggests, a layered architecture has many different layers. One typical example of a layered architecture is an operating system where different layers are used to provide services and functionality and the inner layers are closer to the machine hardware than the outer layers. In this way, each layer isolates the outer layer from inner complexities. In order to work properly, the outer layer only needs to know the interface provided by the inner layer. If there are any changes in the inner layer, as long as the interface does not change, the outer layer is not affected. This scheme tremendously portability of the system. The basic layered architecture is depicted in the following diagram.



Ilear Interface

#### Reference architectures

Reference architecture is not a physical architecture. It is only a reference for defining protocols and designing and implementing systems developed by different parties. Reference models are derived from a study of the application domain rather than from existing systems. It may be used as a basis for system implementation or to compare different systems. It acts as a standard against which systems can be evaluated. One very common example of such a reference model is the OSI model which is a layered model for communication systems. The success of this kind model is evident from the success of the Internet where all kinds of heterogeneous systems can talk to each other only because all of them use the same reference architecture.



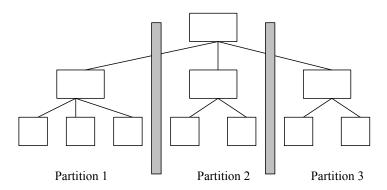
OSI reference model

# **Partitioning the Architecture**

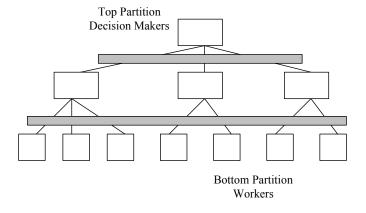
Partitioning of architecture is an important concept. What we basically want to do is distribute the responsibilities to different subsystems so that we get a software system which is easy to maintain. Partitioning results in a system that suffers from fewer side effects. This ultimately means that we get a system that is easier to test and extend and hence is easier to maintain.

Partitioning of an architecture may be "horizontal" and/or "vertical".

In the horizontal partitioning we define separate branches of the module hierarchy for each major function and control modules are used to coordinate communication between functions. This concept is depicted in the following diagram.



Vertical partitioning divides the application from a decision making perspective. The architecture is partitioned in horizontal layers so that decision making and work are stratified with the decision making modules residing at the top of the hierarchy and worker coming at the bottom. This partitioning is also known as factoring and the general model is depicted in the following diagram.



## Analyzing Architecture design

In a given system, the required characteristics may conflict. Trade-offs seek optimal combinations of properties based on cost/benefit analysis. So the analysis requires an understanding of what is required and the relative priority of the attributes has to be established. The following sequence of steps provides a guideline for performing architectural analysis.

- 1. Collect scenarios.
- 2. Elicit requirements, constraints, and environment description.
- 3. Describe the architectural styles/patterns that have been chosen to address the scenarios and requirements. These views include module view, process view, and data flow view.
- 4. Evaluate quality attributes by considered each attribute in isolation.
- 5. Identify the sensitivity of quality attributes to various architectural attributes for a specific architectural style.
- 6. Critique candidate architectures (developed in step 3) using the sensitivity analysis conducted in step 5.