Sommaire

Bibliographie

- **Pattern-Oriented Software Architecture**, Volume 1: A System of Patterns, de F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal. Wiley
- **Microsoft® Application Architecture Guide**, de Microsoft patterns & practices. 2009.
- **Real-Time Design Patterns: Robust Scalable Architecture for Real-Time Systems**, de B. Powel Douglass
- **Software architecture and Design Illuminated**, de K. Qian, X. Fu, L. Tao, C.-W. Xu, J. L. Dias Herrera, John and Barret Publishers. 2010

Data-flow Architecture

Dataflow architectures view an entire software system as a series of transformations on successive sets of data. Each of these sets is independend of the others. The software is decomposed in data processing elements where data directs the order of computation and processing. – Kai Quian et. al,

In such architectures, data can either flow in linear fashion, or in cycles or any other topology of processing elements. Regardless of the structure, data always moves from one element to another. There is usually no other interaction or dependency between the processing elements than these data connectors. These connections can be implemented in various ways (I/O streams, sockets, files, queues or other means).

Data-Flow Architecture : *Pipe and Filter*

Extract from : https://docs.microsoft.com/en-us/azure/architecture/patterns/pipes-and-filters

Decompose a task that performs complex processing into a series of separate elements (also filters) that can be reused. These filters can be combined together into a pipeline. This helps to avoid duplicating code, and makes it easy to remove, replace, or integrate additional components if the processing requirements change.

The time it takes to process a single request depends on the speed of the slowest filter in the pipeline. One or more filters could be a bottleneck, especially if a large number of requests appear in a stream from a particular data source. A key advantage of the pipeline structure is that it provides opportunities for running parallel instances of slow filters, enabling the system to spread the load and improve throughput.

The filters that make up a pipeline can run on different machines, enabling them to be scaled independently and take advantage of the elasticity that many cloud environments provide. A filter that is computationally intensive can run on high-performance hardware, while other less demanding filters can be hosted on less expensive commodity hardware. The filters don't even have to be in the same data center or geographical location, which allows each element in a pipeline to run in an environment that is close to the resources it requires.

If the input and output of a filter are structured as a stream, it's possible to perform the processing for each filter in parallel. The first filter in the pipeline can start its work and output its results, which are passed directly on to the next filter in the sequence before the first filter has completed its work.

Issues and considerations

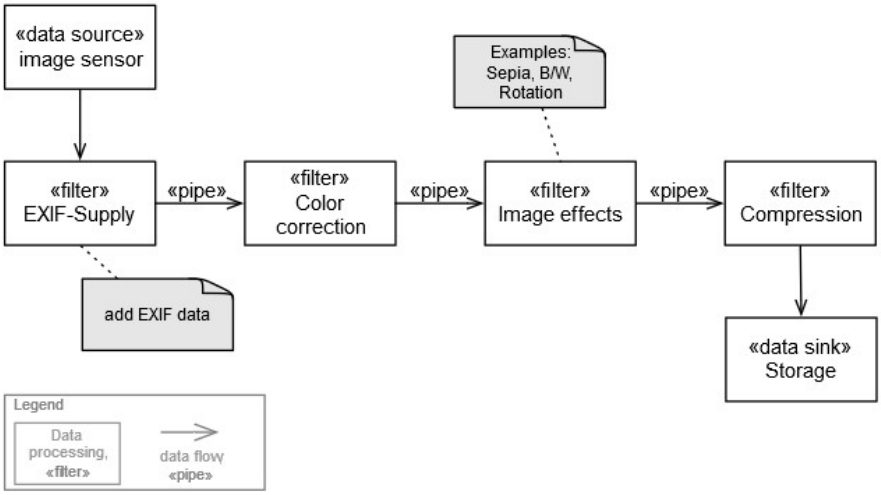
- **Complexity**. The increased flexibility that this pattern provides can also introduce complexity, especially if the filters in a pipeline are distributed across different servers.
- **Reliability**. Use an infrastructure that ensures that data flowing between filters in a pipeline won't be lost.
- **Repeated messages**. If a filter in a pipeline fails after posting a message to the next stage of the pipeline, another instance of the filter might be run, and it'll post a copy of the same message to the pipeline. This could cause two instances of the same message to be passed to the next filter. To avoid this, the pipeline should detect and eliminate duplicate messages.
- **Context and state**. In a pipeline, each filter essentially runs in isolation and shouldn't make any assumptions about how it was invoked. This means that each filter should be provided with sufficient context to perform its work. This context could include a large amount of state information.

Use this pattern when:

- The processing required by an application can easily be broken down into a set of independent steps.
- The processing steps performed by an application have different scalability requirements.
- Flexibility is required to allow reordering of the processing steps performed by an application, or the capability to add and remove steps.
- The system can benefit from distributing the processing for steps across different servers.
- A reliable solution is required that minimizes the effects of failure in a step while data is being processed.

This pattern might not be useful when:

- The processing steps performed by an application aren't independent, or they have to be performed together as part of the same transaction.
- The amount of context or state information required by a step makes this approach inefficient. It might be possible to persist state information to a database instead, but don't use this strategy if the additional load on the database causes excessive contention.



Example of pipe and filter architecture from http://patterns.arc42.org/patterns/pipes-filter/

Data-Flow Architecture : *Process Control Architecture*

https://www.tutorialspoint.com/software_architecture_design/data_flow_architecture.htm

The flow of data comes from a set of variables, which controls the execution of process.

Types of Subsystems

A process control architecture would have a **processing unit** for changing the process control variables and a **controller unit** for calculating the amount of changes.

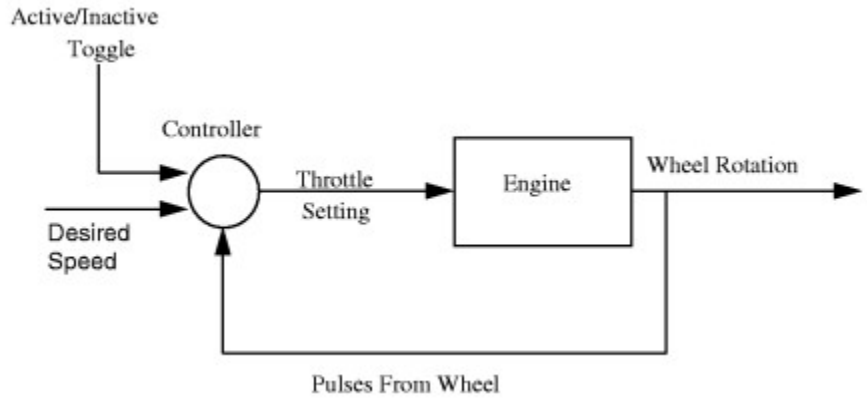
A controller unit must have the following elements −

- **Controlled Variable** − Controlled Variable provides values for the underlying system and should be measured by sensors. For example, speed in cruise control system.
- **Input Variable** − Measures an input to the process. For example, temperature of return air in temperature control system
- **Manipulated Variable** − Manipulated Variable value is adjusted or changed by the controller.
- **Process Definition** − It includes mechanisms for manipulating some process variables.
- **Sensor** − Obtains values of process variables pertinent to control and can be used as a feedback reference to recalculate manipulated variables.
- **Set Point** − It is the desired value for a controlled variable.

- **Control Algorithm** − It is used for deciding how to manipulate process variables.

Application Areas

- Embedded system software design, where the system is manipulated by process control variable data.
- Applications, which aim is to maintain specified properties of the outputs of the process at given reference values.
- Applicable for car-cruise control and building temperature control systems.
- Real-time system software to control automobile anti-lock brakes, nuclear power plants, etc.



Example for a cruise control system, from https://www.cs.cmu.edu/afs/cs/project/tinker-arch/www/html/Tutorial_Slides/Soft_Arch//base.092.html
Benefit of close-loop feedback process control architecture:

- it offers a bette solution to the control system where no precise formula can be used to decide the manipulated variable.
- The software can be completely embedded in the devices.

Data-Flow Architecture : *Shared Memory*

Shared Memory is a pattern which can be classified both as data-flow or real-time architectures

---

The Shared Memory Pattern uses a common memory area addressable by multiple processors as a means to send messages and share data. This is normally accomplished with the addition of special hardware—specifically, multiported RAM chips.

Problem
Many systems have to share data between multiple processors—this is the essence of distribution, after all. In some cases, the access to the data may persist for a long period of time, and the amount of data shared may be large. In such cases, sending messages may be an inefficient method for sharing such information. Multiple computers may need to update this "global" data, such as in a shared database, or they may need to only read it, as is the case

with executable code that may run on many processors or configuration tables. A means by which such data may be effectively shared is needed.

Solution
The Shared Memory Pattern is a simple solution when data must be shared among more than one processor, but timely responses to messages and events between the processors are not required. The pattern almost always involves a combined hardware/software solution. Hardware support for single CPU-cycle semaphore and memory access can avoid memory conflicts and data corruption, but usually some software support to assist the low-level hardware features is required for robust access. If the data to be shared is read-only, as for code that is to be executed on multiple processors, then such concurrency protection mechanisms may not be required.

Hierarchical Architecture

https://www.tutorialspoint.com/software_architecture_design/hierarchical_architecture.htm

Hierarchical architecture views the whole system as a hierarchy structure, in which the software system is decomposed into logical modules or subsystems at different levels in the hierarchy. This approach is typically used in designing system software such as network protocols and operating systems.

In system software hierarchy design, a low-level subsystem gives services to its adjacent upper level subsystems, which invoke the methods in the lower level. The lower layer provides more specific functionality such as I/O services, transaction, scheduling, security services, etc. The middle layer provides more domain dependent functions such as business logic and core processing services. And, the upper layer provides more abstract functionality in the form of user interface such as GUIs, shell programming facilities, etc.

Hierarchical Architecture : Main-subroutine

The aim of this style is to reuse the modules and freely develop individual modules or subroutine. In this style, a software system is divided into subroutines by using top-down refinement according to desired functionality of the system.

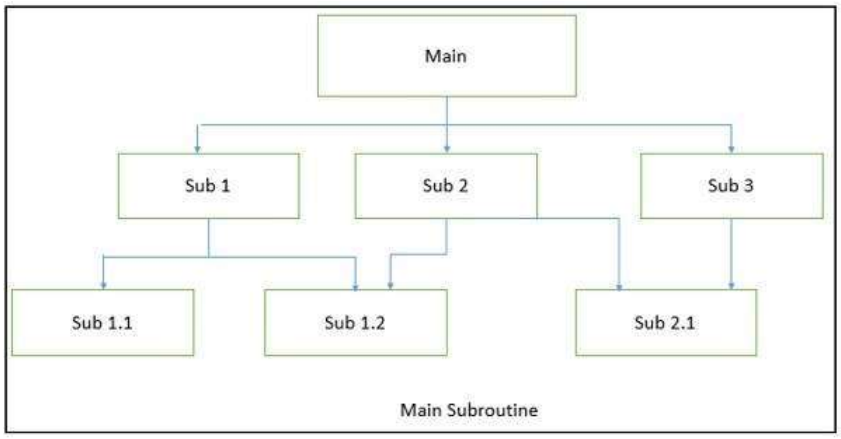
These refinements lead vertically until the decomposed modules is simple enough to have its exclusive independent responsibility. Functionality may be reused and shared by multiple callers in the upper layers.

Advantages

- it is easy to decompose the system based on hierarchy refinement.
- it can be also used in a subsystem of object oriented design.

Disadvantages

- tight coupling may cause more ripple effects of changes,
- , it contains globally shared data, so it is also vulnerable.



Hierarchical Architecture : Master-Slave

This approach applies the 'divide and conquer' principle and supports fault computation and computational accuracy. It is a modification of the main-subroutine architecture that provides reliability of system and fault tolerance.

In this architecture, slaves provide duplicat services to the master, and the master chooses a particular result among slaves by a certain selection strategy. The slaves may perform the same functional task by different algorithms and methods or totally different functionality. It includes parallel computing in which all the slaves can be executed in parallel.

The implementation of the Master-Slave pattern follows five steps −

- **Step 1** − Specify how the computation of the task can be divided into a set of equal sub-tasks and identify the sub-services that are needed to process a sub-task.
- **Step 2** − Specify how the final result of the whole service can be computed with the help of the results obtained from processing individual sub-tasks.
- **Step 3** − Define an interface for the sub-service identified in step 1. It will be implemented by the slave and used by the master to delegate the processing of individual sub-tasks.
- **Step 4** − Implement the slave components according to the specifications developed in the previous step.
- **Step 5** − Implement the master according to the spec. developed in step 1 to 3.

Applications

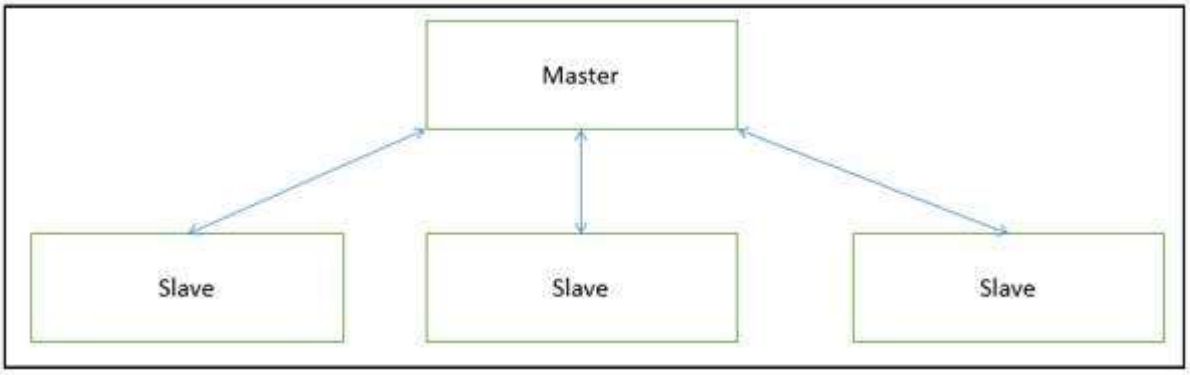
- Suitable for applications where reliability of software is critical issue.
- Widely applied in the areas of parallel and distributed computing.

Advantages

- Faster computation and easy scalability.
- Provides robustness as slaves can be duplicated.
- Slave can be implemented differently to minimize semantic errors.

Disadvantages

- Communication overhead.
- Not all problems can be divided.
- Hard to implement and portability issue.



Hierarchy : Layered Style

In this approach, the system is decomposed into a number of higher and lower layers in a hierarchy, and each layer has its own sole responsibility in the system. Each layer consists of a group of related classes that are encapsulated in a package, in a deployed component, or as a group of subroutines in the format of method library or header file. Each layer provides service to the layer above it and serves as a client to the layer below i.e. request to layer i +1 invokes the services provided by the layer i via the interface of layer i. The response may go back to the layer i +1 if the task is completed; otherwise layer i continually invokes services from layer i -1 below.

Applications

- Applications that involve distinct classes of services that can be organized hierarchically.
- Any application that can be decomposed into application-specific and platform-specific portions.
- Applications that have clear divisions between core services, critical services, and user interface services, etc.

Advantages

- Design based on incremental levels of abstraction.
- Provides enhancement independence as changes to the function of one layer affects at most two other layers.
- Separation of the standard interface and its implementation.
- Implemented by using component-based technology which makes the system much easier to allow for plug-and-play of new components.
- Each layer can be an abstract machine deployed independently which support portability.
- Easy to decompose the system based on the definition of the tasks in a top-down refinement manner
- Different implementations (with identical interfaces) of the same layer can be used interchangeably

Disadvantages

- Lower runtime performance since a client's request or a response to client must go through potentially several layers.
- There are also performance concerns on overhead on the data marshaling and buffering by each layer.

- Opening of interlayer communication may cause deadlocks and “bridging” may cause tight coupling.
- Exceptions and error handling is an issue in the layered architecture, since faults in one layer must spread upwards to all calling layers

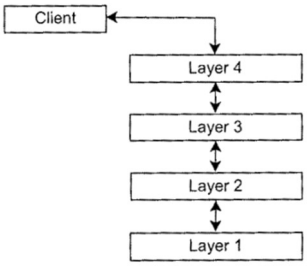


figure from https://coronet.iicm.tugraz.at/sa/s5/sa_styles.html

Hierarchy : Virtual Machine Architecture

Virtual Machine architecture is a special layared-style architecture. It pretends some functionality, which is not native to the hardware and/or software on which it is implemented. A virtual machine is built upon an existing system and provides a virtual abstraction, a set of attributes, and operations.

In virtual machine architecture, the master uses the ‘same’ subservice’ from the slave and performs functions such as split work, call slaves, and combine results. It allows developers to simulate and test platforms, which have not yet been built, and simulate "disaster" modes that would be too complex, costly, or dangerous to test with the real system.

In most cases, a virtual machine splits a programming language or application environment from an execution platform. The main objective is to provide **portability**. Interpretation of a particular module via a Virtual Machine may be perceived as −

- The interpretation engine chooses an instruction from the module being interpreted.
- Based on the instruction, the engine updates the virtual machine’s internal state and the above process is repeated.

Applications

- Suitable for solving a problem by simulation or translation if there is no direct solution.
- Sample applications include interpreters of microprogramming, XML processing, script command language execution, rule-based system execution, Smalltalk and Java interpreter typed programming language.
- Common examples of virtual machines are interpreters, rule-based systems, syntactic shells, and command language processors.

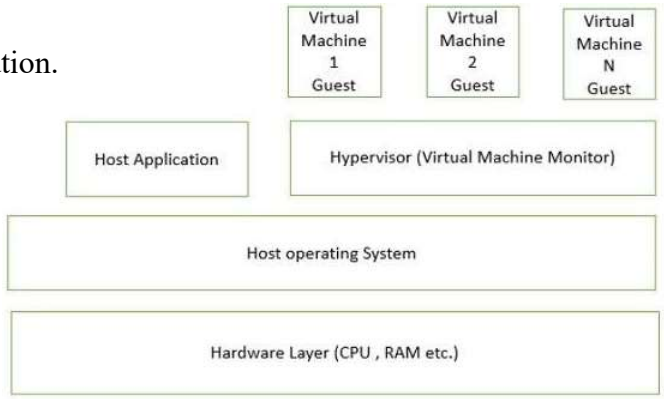
Advantages

Because of having the features of portability and machine platform independency, it has following advantages −

- Simplicity of software development.
- Provides flexibility through the ability to interrupt and query the program.
- Simulation for disaster working model.
- Introduce modifications at runtime.

Disadvantages

- Slow execution of the interpreter due to the interpreter nature.
- There is a performance cost because of the additional computation involved in execution.



https://www.tutorialspoint.com/software_architecture_design/hierarchical_architecture.htm

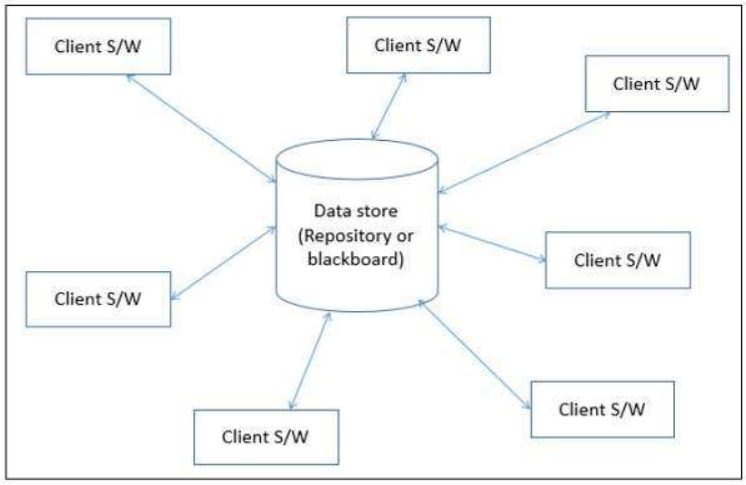
Data-Centered Architecture

https://www.tutorialspoint.com/software_architecture_design/data_centered_architecture.htm

In data-centered architecture, the data is centralized and accessed frequently by other components, which modify data. The main purpose of this style is to achieve integrality of data. Data-centered architecture consists of different components that communicate through shared data repositories. The components access a shared data structure and are relatively independent, in that, they interact only through the data store.

The most well-known examples of the data-centered architecture is a database architecture, in which the common database schema is created with data definition protocol – for example, a set of related tables with fields and data types in an RDBMS.

Another example of data-centered architectures is the web architecture which has a common data schema (i.e. meta-structure of the Web) and follows hypermedia data model and processes communicate through the use of shared web-based data services.



There are two types of components −

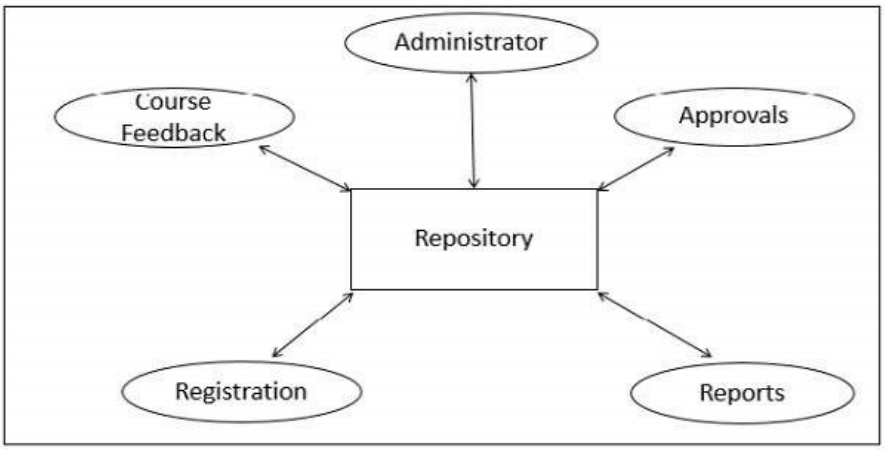
- A **central data** structure or data store or data repository, which is responsible for providing permanent data storage. It represents the current state.
- A **data accessor** or a collection of independent components that operate on the central data store, perform computations, and might put back the results.

Interactions or communication between the data accessors is only through the data store. The data is the only means of communication among clients. The flow of control differentiates the architecture into two categories as **Repository Architecture Style** and **Blackboard Architecture Style**.

Data-centered architecture : Repository Architecture Style

In Repository Architecture Style, the data store is passive and the clients (software components or agents) of the data store are active, which control the logic flow. The participating components check the data-store for changes.

A client sends a request to the system to perform actions (e.g. insert data). The computational processes are independent and triggered by incoming requests. If the types of transactions in an input stream of transactions trigger selection of processes to execute, then it is traditional database or repository architecture, or passive repository. This approach is widely used in DBMS, library information system, the interface repository in CORBA, compilers, and CASE (computer aided software engineering) environments.



Advantages

- Provides data integrity, backup and restore features.
- Provides scalability and reusability of agents as they do not have direct communication with each other.
- Reduces overhead of transient data between software components.

Disadvantages

Because of being more vulnerable to failure and data replication or duplication, Repository Architecture Style has following disadvantages −

- High dependency between data structure of data store and its agents.
- Changes in data structure highly affect the clients.
- Evolution of data is difficult and expensive.
- Cost of moving data on network for distributed data.

Data-centered architecture : Blackboard Architecture Style

In Blackboard Architecture Style, the data store is active and its clients are passive. Therefore the logical flow is determined by the current data status in data store. It has a blackboard component, acting as a central data repository, and an internal representation is built and acted upon by different computational elements.

Further, a number of components that act independently on the common data structure are stored in the blackboard. The components interact only through the blackboard. The data-store alerts the clients whenever there is a data-store changes. The current state of the solution is stored in the blackboard and processing is triggered by the state of the blackboard.

When changes occur in the data, the system sends the notifications known as **trigger** and data to the clients. This approach is found in certain AI applications and complex applications, such as speech recognition, image recognition, security system, and business resource management systems etc.

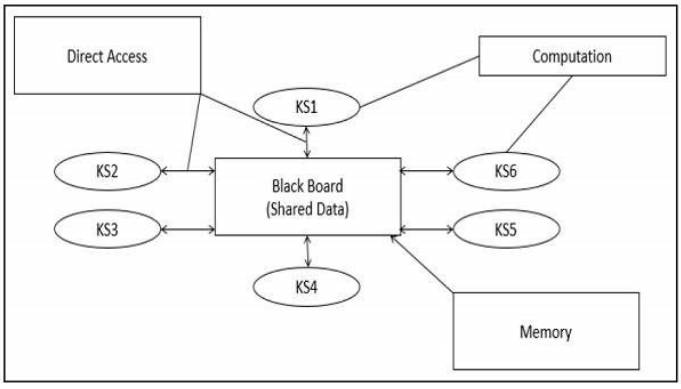
If the current state of the central data structure is the main trigger of selecting processes to execute, the repository can be a blackboard and this shared data source is an active agent.

A major difference with traditional database systems is that the invocation of computational elements in a blackboard architecture is triggered by the current state of the blackboard, and not by external inputs.

Parts of Blackboard Model

The blackboard model is usually presented with three major parts −

- Knowledge Sources (KS), also known as **Listeners** or **Subscribers** are distinct and independent units. They solve parts of a problem and aggregate partial results. Interaction among knowledge sources takes place uniquely through the blackboard.
- Blackboard Data Structure The problem-solving state data is organized into an application-dependent hierarchy. Knowledge sources make changes to the blackboard that lead incrementally to a solution to the problem.
- Control manages tasks and checks the work state.



Advantages

- Provides concurrency that allows all knowledge sources to work in parallel as they independent of each other.
- IIts scalability feature facilitates easy steps to add or update knowledge source.

- It supports experimentation for hypotheses and reusability of knowledge source agents.

Disadvantages

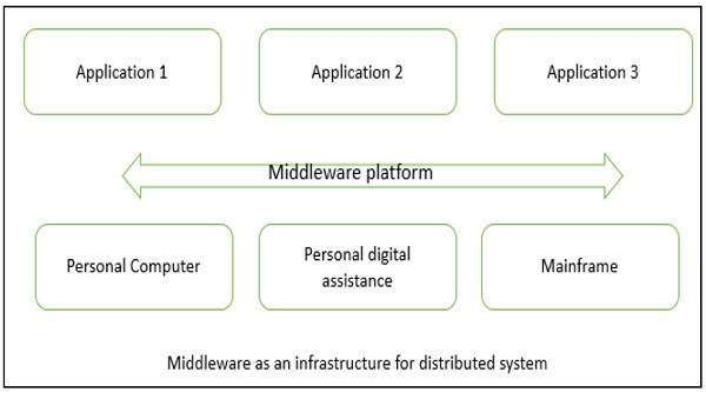
- The structural change of blackboard may have a significant impact on all of its agents, as close dependency exists between blackboard and knowledge source.
- Blackboard model is expected to produce approximate solution; however, sometimes, it becomes difficult to decide when to terminate the reasoning.
- Further, this model suffers some problems in synchronization of multiple agents, therefore, it faces challenge in designing and testing of the system.

Distributed Architecture

A distributed system can be demonstrated by the client-server architecture, which forms the base for multi-tier architectures; alternatives are the broker architecture such as CORBA, and the Service-Oriented Architecture (SOA). In this architecture, information processing is not confined to a single machine rather it is distributed over several independent computers.

There are several technology frameworks to support distributed architectures, including .NET, J2EE, CORBA, .NET Web services, AXIS Java Web services, and Globus Grid services. Middleware is an infrastructure that appropriately supports the development and execution of distributed applications. It provides a buffer between the applications and the network.

It sits in the middle of system and manages or supports the different components of a distributed system. Examples are transaction processing monitors, data convertors and communication controllers, etc.



https://www.tutorialspoint.com/software_architecture_design/distributed_architecture.htm

Distributed Architecture ***Client-Server***

The client/server architectural style describes distributed systems that involve a separate client and server system, and a connecting network. The simplest form of client/server system involves a server application that is accessed directly by multiple clients, referred to as a 2-Tier architectural style.

Historically, client/server architecture indicated a graphical desktop UI application that communicated with a database server containing much of the business logic in the form of stored procedures, or with a dedicated file server. More generally, however, the client/server

architectural style describes the relationship between a client and one or more servers, where the client initiates one or more requests (perhaps using a graphical UI), waits for replies, and processes the replies on receipt. The server typically authorizes the user and then carries out the processing required to generate the result. The server may send responses using a range of protocols and data formats to communicate information to the client.

Today, some examples of the client/server architectural style include Web browser—based programs running on the Internet or an intranet; applications that access remote data stores (such as e-mail readers, FTP clients, and database query tools); and tools and utilities that manipulate remote systems (such as system management tools and network monitoring tools).

The main benefits of the client/server architectural style are:

- **Higher security**. All data is stored on the server, which generally offers a greater control of security than client machines.
- **Centralized data access**. Because data is stored only on the server, access and updates to the data are far easier to administer than in other architectural styles.
- **Ease of maintenance**. Roles and responsibilities of a computing system are distributed among several servers that are known to each other through a network. This ensures that a client remains unaware and unaffected by a server repair, upgrade, or relocation.

In **thin-client** model, all the application processing and data management is carried by the server. The client is simply responsible for running the GUI software. It is used when legacy systems are migrated to client server architectures in which legacy system acts as a server in its own right with a graphical interface implemented on a client. However, A major disadvantage is that it places a heavy processing load on both the server and the network.

In **thick-client** model, the server is in-charge of the data management. The software on the client implements the application logic and the interactions with the system user. It is most appropriate for new client-server systems where the capabilities of the client system are known in advance. However, it is more complex than a thin client model especially for management, as all clients should have same copy/version of software application.

Consider the client/server architectural style if your application is server based and will support many clients, you are creating Web-based applications exposed through a Web browser, you are implementing business processes that will be used by people throughout the organization, or you are creating services for other applications to consume. The client/server architectural style is also suitable, like many networked styles, when you want to centralize data storage, backup, and management functions, or when your application must support different client types and different devices.

Advantages

- Separation of responsibilities such as user interface presentation and business logic processing.
- Reusability of server components and potential for concurrency
- Simplifies the design and the development of distributed applications
- It makes it easy to migrate or integrate existing applications into a distributed environment.

- It also makes effective use of resources when a large number of clients are accessing a high-performance server.

Disadvantages

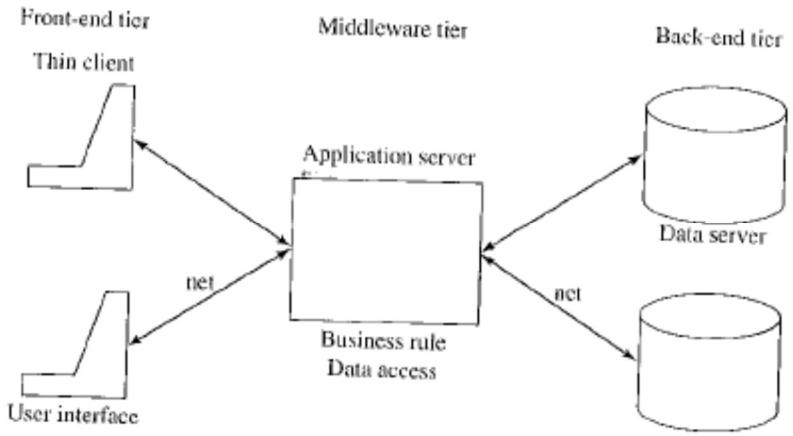
- Lack of heterogeneous infrastructure to deal with the requirement changes.
- Security complications.
- Limited server availability and reliability.
- Limited testability and scalability.
- Fat clients with presentation and business logic together.

Distributed Architecture 3-tiers & *Multi-tiers*

N-tier and 3-tier are architectural deployment styles that describe the separation of functionality into segments in much the same way as the layered style, but with each segment being a tier that can be located on a physically separate computer. They evolved through the component-oriented approach, generally using platform specific methods for communication instead of a message-based approach.

N-tier application architecture is characterized by the functional decomposition of applications, service components, and their distributed deployment, providing improved scalability, availability, manageability, and resource utilization. Each tier is completely independent from all other tiers, except for those immediately above and below it. The nth tier only has to know how to handle a request from the n+1th tier, how to forward that request on to the n-1th tier (if there is one), and how to handle the results of the request. Communication between tiers is typically asynchronous in order to support better scalability.

N-tier architectures usually have at least three separate logical parts, each located on a separate physical server. Each part is responsible for specific functionality. When using a layered design approach, a layer is deployed on a tier if more than one service or application is dependent on the functionality exposed by the layer.



Three-tier architecture

An example of the N-tier/3-tier architectural style is a typical financial Web application where security is important. The business layer must be deployed behind a firewall, which forces the deployment of the presentation layer on a separate tier in the perimeter network. Another example is a typical rich client connected application, where the presentation layer is deployed on client machines and the business layer and data access layer are deployed on one or more server tiers.

The main benefits of the N-tier/3-tier architectural style are:

- **Maintainability**. Because each tier is independent of the other tiers, updates or changes can be carried out without affecting the application as a whole.
- **Scalability**. Because tiers are based on the deployment of layers, scaling out an application is reasonably straightforward.
- **Flexibility**. Because each tier can be managed or scaled independently, flexibility is increased.
- **Availability**. Applications can exploit the modular architecture of enabling systems using easily scalable components, which increases availability.

Disadvantages

- Unsatisfactory Testability due to lack of testing tools.
- More critical server reliability and availability.

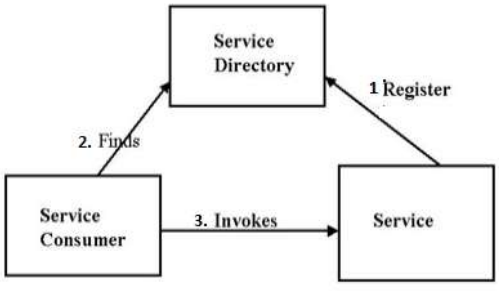
Consider either the N-tier or the 3-tier architectural style if the processing requirements of the layers in the application differ such that processing in one layer could absorb sufficient resources to slow the processing in other layers, or if the security requirements of the layers in the application differ. For example, the presentation layer should not store sensitive data, while this may be stored in the business and data layers. The N-tier or the 3-tier architectural style is also appropriate if you want to be able to share business logic between applications, and you have sufficient hardware to allocate the required number of servers to each tier.

Consider using just three tiers if you are developing an intranet application where all servers are located within the private network; or an Internet application where security requirements do not restrict the deployment of business logic on the public facing Web or application server. Consider using more than three tiers if security requirements dictate that business logic cannot be deployed to the perimeter network, or the application makes heavy use of resources and you want to offload that functionality to another server

Distributed Architecture — *Service Oriented Architecture*

A service is a component of business functionality that is well-defined, self-contained, independent, published, and available to be used via a standard programming interface. The connections between services are conducted by common and universal message-oriented protocols such as the SOAP Web service protocol, which can deliver requests and responses between services loosely.

Service-oriented architecture is a client/server design which support business-driven IT approach in which an application consists of software services and software service consumers (also known as clients or service requesters).



Advantages

- Loose coupling of service–orientation provides great flexibility to make use of all available service recourses irrespective of platform and technology restrictions.
- Each service component is independent from other services due to the stateless service feature.

- The implementation of a service will not affect the application of the service as long as the exposed interface is not changed.
- A client or any service can access other services regardless of their platform, technology, vendors, or language implementations.
- Reusability of assets and services since clients of a service only need to know its public interfaces, service composition.
- SOA based business application development are much more efficient in terms of time and cost.
- Enhances the scalability and provide standard connection between systems.
- Efficient and effective usage of 'Business Services'.
- Integration becomes much easier and improved intrinsic interoperability.

Distributed Architecture *Broker*

Broker Architectural Style is a middleware architecture used in distributed computing to coordinate and enable the communication between registered servers and clients. Here, object communication takes place through a middleware system called an object request broker (software bus).

However, client and the server do not interact with each other directly. Client and server have a direct connection to its proxy, which communicates with the mediator-broker. A server provides services by registering and publishing their interfaces with the broker and clients can request the services from the broker statically or dynamically by look-up.

In this architecture, **Broker** is responsible for coordinating communication, such as forwarding and dispatching the results and exceptions. It can be either an invocation-oriented service, a document or message - oriented broker to which clients send a message. Broker is responsible for brokering the service requests, locating a proper server, transmitting requests, and sending responses back to clients. It also retains the servers' registration information including their functionality and services as well as location information. Further, it provides APIs for clients to request, servers to respond, registering or unregistering server components, transferring messages, and locating servers.

Stubs are generated at the static compilation time and then deployed to the client side which is used as a proxy for the client. Client-side proxy acts as a mediator between the client and the broker and provides additional transparency between them and the client; a remote object appears like a local one. The proxy hides the IPC (inter-process communication) at protocol level and performs marshaling of parameter values and un-marshaling of results from the server.

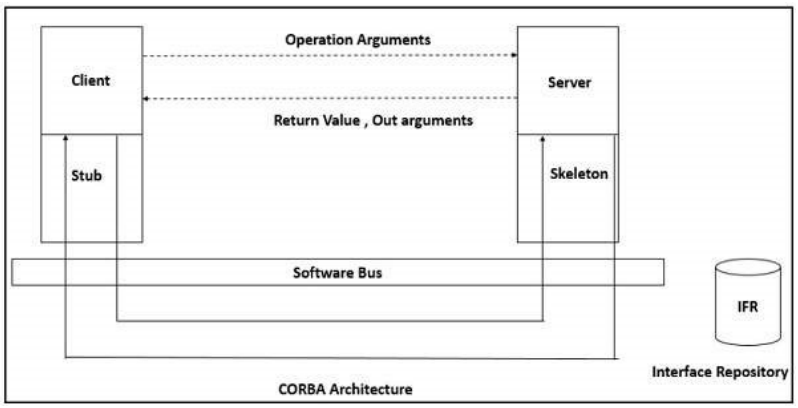
Skeleton is generated by the service interface compilation and then deployed to the server side, which is used as a proxy for the server. Server-side proxy encapsulates low-level system-specific networking functions and provides high-level APIs to mediate between the server and the broker. Further, it receives the requests, unpacks the requests, unmarshals the method arguments, calls the suitable service, and also marshals the result before sending it back to the client.

A **bridge** can connect two different networks based on different communication protocols. It mediates different brokers including DCOM, .NET remote, and Java CORBA brokers. Bridges are optional component, which hides the implementation details when two brokers

interoperate and take requests and parameters in one format and translate them to another format.

Broker implementation in CORBA

CORBA is an international standard for an Object Request Broker – a middleware to manage communications among distributed objects defined by OMG (object management group).



Avantages:

- server component implementation and location transparency
- changeability and extensibility
- simplicity for clients to access server and server portability
- interoperability via broker bridges
- reusability
- feasibility of runtime changes of server components

Disavantages:

- Inefficiency due to the overhead of proxies
- Low fault-tolerance
- Difficulty in testing due to the amount proxies

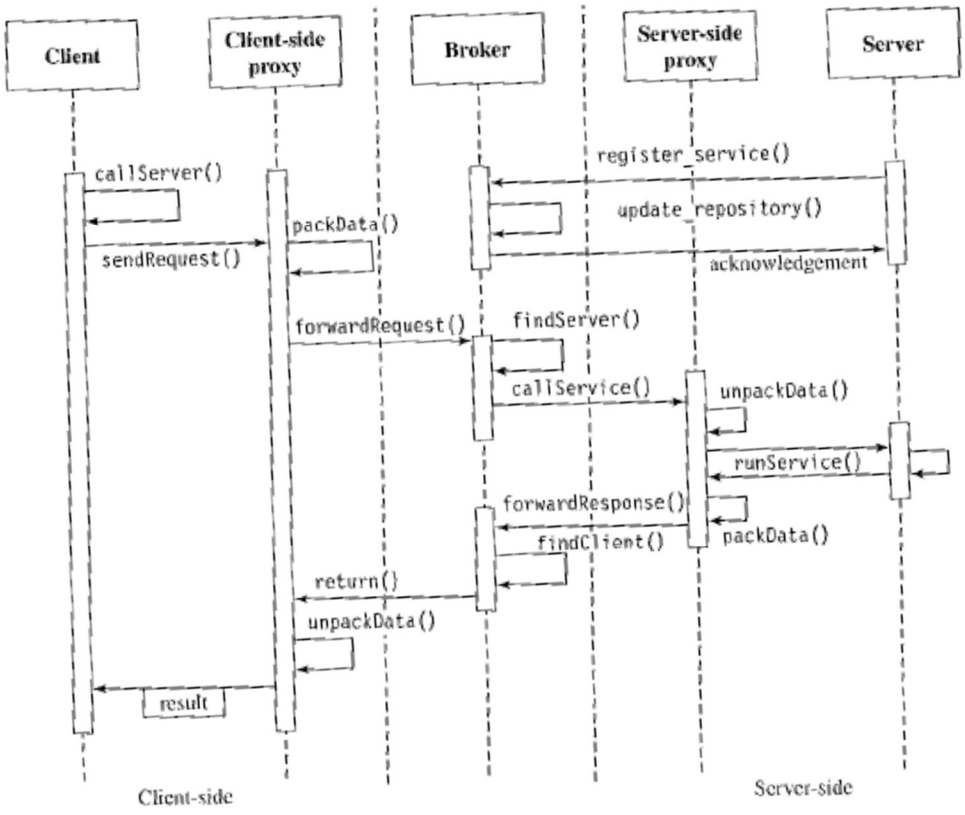


Figure 10.6

Interaction-oriented Architecture

Interaction-oriented architecture has two major styles − **Model-View-Controller** (MVC) and **Presentation-Abstraction-Control** (PAC). Both MVC and PAC propose three components decomposition and are used for interactive applications such as web applications with multiple talks and user interactions. They are different in their flow of control and organization. PAC is an agent-based hierarchical architecture but MVC does not have a clear hierarchical structure.

Interaction-Oriented Software Architecture *MVC*

MVC decomposes a given software application into three interconnected parts that help in separating the internal representations of information from the information presented to or accepted from the user.

In this architecture, **Model** is a central component of MVC that directly manages the data, logic, and constraints of an application. It consists of data components, which maintain the raw application data and application logic for interface. It is an independent user interface and captures the behavior of application problem domain. Model is the domain-specific software simulation or implementation of the application's central structure. If there is change in its state, it gives notification to its associated view to produce updated output, and the controller to change the available set of commands.

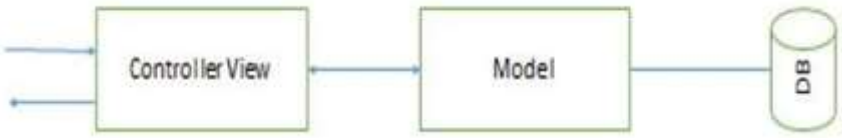
A **controller** accepts an input and converts it into commands for the model or view. It acts as an interface between the associated models and views and the input devices. Controller can send commands to the model to update the model's state and to its associated view to change the view's presentation of the model. Likewise, as shown in the picture given below, it consists of input processing components, which handle input from the user by modifying the model.

View can be used to represent any output of information in graphical form such as diagram or chart. It consists of presentation components which provide the visual representations of data. Views request information from their model and generate an output representation to the user. Multiple views of the same information are possible, such as a bar chart for management and a tabular view for accountants.

MVC – I is a simple version of MVC architecture where the system is divided into two sub-systems −

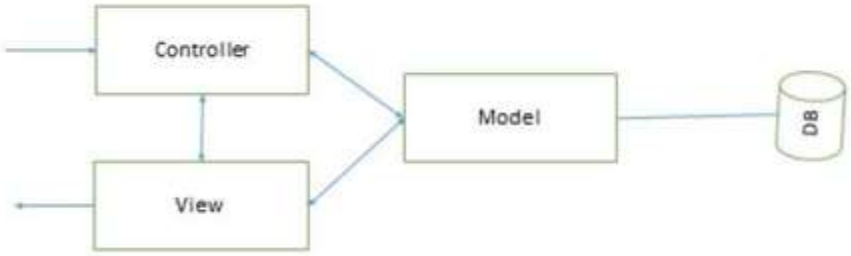
- **The Controller-View** acts as input /output interface and processing is done.
- **The Model** − The model provides all the data and domain services.

The model module notifies controller-view module of any data changes so that any graphics data display will be changed accordingly. The controller also takes appropriate action upon the changes.



The connection between controller-view and model can be designed in a pattern (as shown in the above picture) of subscribe-notify whereby the controller-view subscribes to model and model notifies controller-view of any changes.

MVC–II is an enhancement of MVC-I architecture in which the view module and the controller module are separate. The model module plays an active role as in MVC-I by providing all the core functionality and data supported by database. The view module presents data while controller module accepts input request, validates input data, initiates the model, the view, their connection, and also dispatches the task.



MVC applications are effective for interactive applications where multiple views are needed for a single data model and easy to plug-in a new or change interface view. They are suitable for applications where there are clear divisions between the modules so that different professionals can be assigned to work on different aspects of such applications concurrently.

Advantages

- Multiple views synchronized with same data model.
- Easy to plug-in new or replace interface views.
- Used for application development where graphics expertise professionals, programming professionals, and data base development professionals are working in a designed project team.

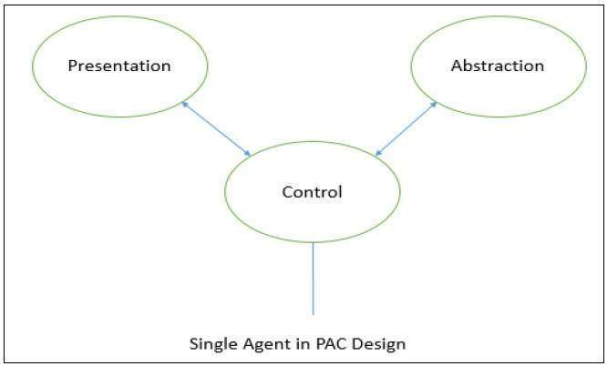
Disadvantages

- Not suitable for agent-oriented applications such as interactive mobile and robotics applications.
- Multiple pairs of controllers and views based on the same data model make any data model change expensive.
- The division between the View and the Controller is not clear in some cases.

Interaction-Oriented Software Architecture — *PAC*

In PAC, the system is arranged into a hierarchy of many cooperating agents (triads). It was developed from MVC to support the application requirement of multiple agents in addition to interactive requirements. Each agent has three components −

- **The presentation component** − Formats the presentation of data.
- **The abstraction component** − Retrieves and processes the data.
- **The control component** − Handles the task such as the flow of control and communication between the other two components.



Single Agent in PAC Design

The PAC architecture is similar to MVC, in the sense that presentation module is like view module of MVC. The abstraction module looks like model module of MVC and the control module is like the controller module of MVC, but they differ in their flow of control and organization.

There are no direct connections between abstraction and presentation components in each agent. The control component in each agent is in charge of communications with other agents.

In PACs consisting of multiple agents, the top-level agent provides core data and business logics. The bottom level agents define detailed specific data and presentations. The intermediate level or middle level agent acts as coordinator of low-level agents. Each agent has its own specific assigned job. For some middle level agents the interactive presentations are not required, so they do not have a presentation component. The control component is required for all agents through which all the agents communicate with each other.

Applications

- Effective for an interactive system where the system can be decomposed into many cooperating agents in a hierarchical manner.
- Effective when the coupling among the agents is expected to be loose so that changes on an agent does not affect others.
- Effective for distributed system where all the agents are distantly distributed and each of them has its own functionalities with data and interactive interface.

Advantages

- Support for multi-tasking and multi-viewing
- Support for agent reusability and extensibility
- Easy to plug-in new agent or change an existing one
- Support for concurrency where multiple agents are running in parallel in different threads or different devices or computers

Disadvantages

- Overhead due to the control bridge between presentation and abstraction and the communication of controls among agents.
- Difficult to determine the right number of agents because of loose coupling and high independence among agents.