Introduction to Software Architecture

Table of Contents

1 Introduction

What is software architecture?

There is no standard, universally-accepted definition of the term, for software architecture is a field in its infancy, although its roots run deep in software engineering.

1.1 What is a Software Architecture

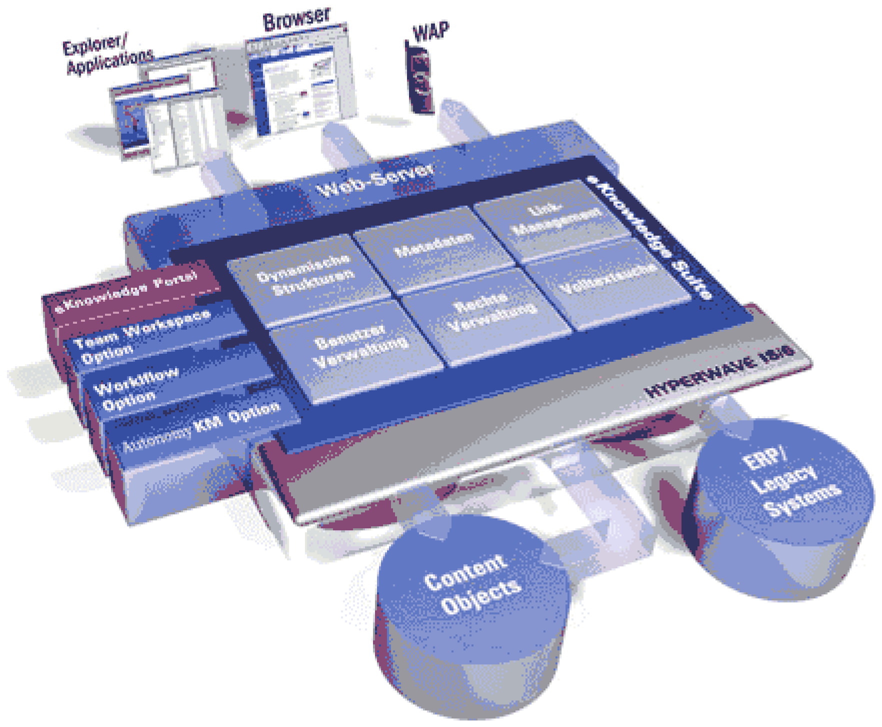
An ***architecture*** is

- set of significant decisions about the organization of a software system,
- selection of the structural elements and their interfaces by which the system is composed,
- behavior of the structural elements as specified in the collaborations among those elements,
- composition of these structural and behavioral elements into progressively larger subsystems,
- architectural style that guides this organization (i.e. these elements and their interfaces, their collaborations, and their composition).

In the definition above, we assume that components can make of a component.

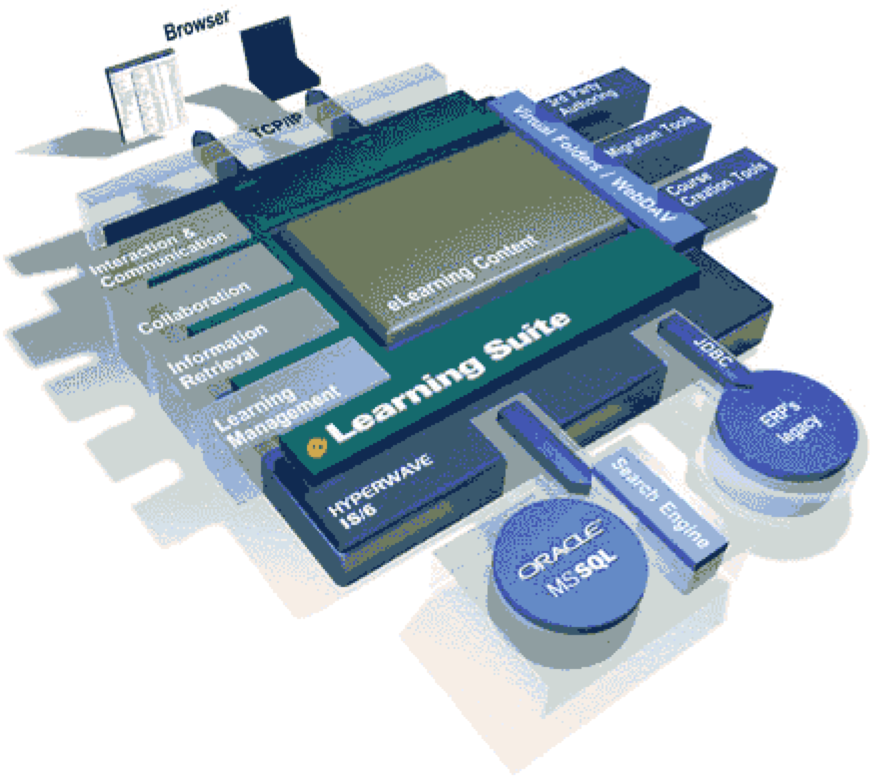
The intent of this definition is that a software architecture must abstract away some information from the system (otherwise there is no point looking at the architecture, we are simply viewing the entire system) and yet provide enough information to be a basis for analysis, decision making, and hence risk reduction.

First, architecture defines ***components***. The architecture embodies information about how the components interact with each other. This means that architecture specifically omits content information about components that does not pertain to their interaction.



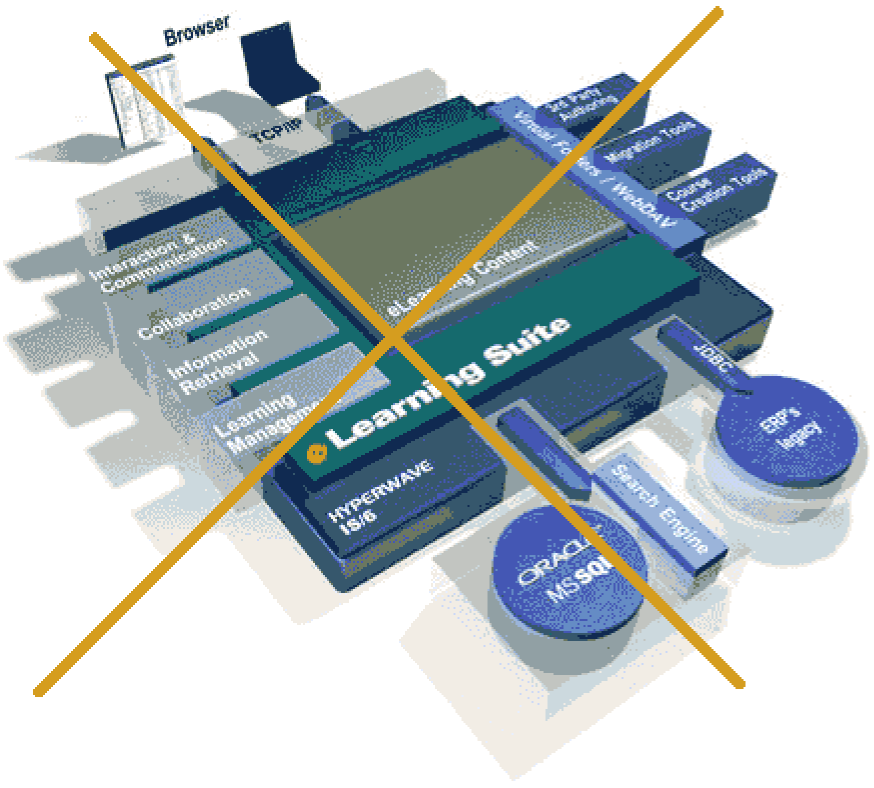
Second, the definition makes clear that systems can comprise more than one structure, and that no one structure holds the irrefutable claim to being the architecture.
By intention, the definition does not specify what architectural components and relationships are. Is a software component an object? A process? A library? A database? A commercial product? It can be any of these things and more.

Third, the definition implies that every software system has an architecture, because every system can be shown to be composed of components and relations among them.



Fourth, the ***behavior*** of each component is part of the architecture, insofar as that behavior can be observed or discerned from the point of view of another component. This behavior is what allows components to interact with each other, which is clearly part of the architecture.

Hence, most of the box-and-line drawings that are passed off as architectures are in fact not architectures at all. They are simply box-and-line drawings.


Browser
TCP/IP
Interaction & Communication
Collaboration
Information Retrieval
Learning Management
eLearning Content
Learning Suite
HYPERWAVE IS/6
Virtual Folders / WebDAV
3rd Party Authoring
Migration Tools
Course Creation Tools
JDBC
ERP's legacy
ORACLE MSSQL
Search Engine

2 System Quality Attributes

Fifth, the ***architecture*** essentially defines ***"externally visible" properties*** also known as ***system quality attributes*** for the whole software project, we are referring to such properties as its provided services, performance characteristics, fault handling, shared resource usage, and so on.

Evaluation of an architecture's properties is critical to successful system development. However, reasoning about a system's intended architecture must be recognized as distinct from reasoning about its realized architecture. As design and eventually implementation of an architecture proceed, faithfulness to the principles of the intended architecture is not always easy to achieve. This is particularly true in cases where the intended architecture is not completely specified, documented or disseminated to all of the project members.

2.1 Run-Time Quality Attributes

There are attributes of a software-intensive system that define the system functionality and are visible at runtime. They are discussed in the following subsections.

Performance refers to the responsiveness of the system − the 'time required to respond to stimuli (events) or the number of events processed in some interval of time.

Performance qualities are often expressed by the number of transactions per unit time or by the amount of time it takes a transaction with the system to complete.

Since communication usually takes longer than computation, performance is often a function of how much communication and interaction there is between the components of the system-clearly an architectural issue.

Security is a measure of the system's ability to resist unauthorized attempts at usage and denial of service while still providing its services to legitimate users.

It is categorized in terms of the ***types of threats*** that might be made to system;

Availability measures the proportion of time the system is up and running.

It is measured by the length of time between failures as well as by how quickly the system is able to resume operation in the event of failure. The steady state availability of a system is the proportion of time that the system is functioning correctly and is typically seen as follows:

time to failure/(time to failure + time to repair)

Availability comes from both "time to failure" and "time to repair"; both are addressed through architectural means.

Reliability is closely related to availability, the ability of the system to keep operating over time. Reliability is usually measured with "time to failure". This is a quality attribute that is tied to the architecture:

> careful attention to error reporting and handling (which involves constraining the interaction patterns among the components), and special kinds of components such as time-out monitors.

Mean time to failure is lengthened primarily by making an architecture ***fault tolerant***.

Fault tolerance, in turn, is achieved by the replication of critical processing elements and connections within the architecture. Mean time to failure can also be lengthened by gelding a less error-prone system, which is addressed architecturally by careful separation of concerns, which leads to better inerrability and testability.

Functionality is the ability of the system to do the work for which it was intended.

Performing a task requires that many or most of the system's components work in a coordinated manner to complete the job, just as framers, electricians, plumbers, drywall hangers, painters, and finish carpenters all come together to cooperatively perform the task of getting a house built.

Therefore, if the components have not been assigned the correct responsibilities or have not been endowed with the correct facilities for coordinating with other components (so that, for instance, they know when it is time for them to begin their portion of the task), the system will be unable to perform the required functionality.

Usability can be broken down into the following areas:

- ***Learnability***: How quick and easy is it for a user to learn to use the system's interface?
- ***Efficiency***: Does the system respond with appropriate speed to a user's requests?
- ***Memorability***: Can the user remember how to do system operations between uses of the system?
- ***Error avoidance***: Does the system anticipate and prevent common user errors?
- ***Error handling***: Does the system help the user recover from errors?
- ***Satisfaction***: Does the system make the user's job easy?

2.2 Engineering Quality Attributes

There are other attributes of a software-intensive system that cannot be discerned at runtime. They are discussed in the following subsections.

Modifiability, in all its forms, may be the quality attribute most closely aligned to the architecture of a system.

The ability to make changes quickly and cost effectively follows directly from the architecture: Modifiability is largely a function of the locality of any change.

Making a widespread change to the system is more costly than making a change to a single component, all other things being equal.

There are exceptions, of course.

A single component, if excessively large and complex, may be more costly to change than five simple ones.

It's also easy to imagine a global change that in each place is simple and systematic: changing the value of a constant that appears everywhere, for instance.

However, in large systems, making a change is much more costly than just, well, making the change. Development process costs start to dominate, such as maintaining version control, approving the change across many change control boards, coordinating the change time across many large teams, retesting all the units, perhaps assuring backward compatibility, and so forth. We take as a general principle that local is better.

Since the architecture defines the components and the responsibilities of each, it also defines the circumstances under which each component will have to change. An architecture effectively classifies all possible changes into four categories

- *Extending or changing capabilities*. Adding new functionality, enhancing existing functionality, or repairing bugs. The ability to acquire new features is called extensibility. Adding new capabilities is important to remain competitive against other products in the same market.
- *Deleting unwanted capabilities*. To streamline or simplify the functionality of an existing application, perhaps to deliver a less-capable (and therefore less expensive) version of a product to a wider customer base.
- *Adapting to new operating environments*. For example, processor hardware, input/output devices, and logical devices. This kind of modification occurs so often that the quality of being amenable to it has a special name, portability, which we will discuss separately. Portability makes a product more flexible in how it can be fielded, appealing to a broader customer base.
- *Restructuring*. For example, rationalizing system services, modularising, . optimising, or creating reusable components that may serve to give the organization a head start on future systems.

Portability is the ability of the system to run under different computing environments.

These environments can be hardware, software, or a combination of the two. A system is portable to the extent that all of the assumptions about any particular computing environment are confined to one component (or at worst, a small number of easily changed components).

The encapsulation of platform-specific considerations in an architecture typically takes the form of a portability layer, a set of software services that gives application software an abstract interface to its environment. This interface remains constant (thus insulating the application software from change) even though the implementation of that layer changes as the system is ported from environment to environment.

Reusability is usually taken to mean designing a system so that the system's structure or some of its components can be reused again in future applications.

Designing for reusability means that the system has been structured so that its components can be chosen from previously built products, in which case it is a synonym for integrability . In either case, reusability can be conceived of as another *special case of modifiability*.

Integrability is the ability to make the separately developed components of the system work correctly together. This in turn depends on the external complexity of the components, their interaction mechanisms and protocols, and the degree to which responsibilities have been cleanly partitioned, all architecture-level issues.

Inerrability also depends upon how well and completely the interfaces to the components have been specified. Integrating a component depends not only on the interaction mechanisms used (e.g., procedure call versus process spawning) but also on the functionality assigned to the component to be integrated and how that functionality is related to the functionality of this new component's environment.

Interoperability is a special kind of integrability.

lntegrability measures the ability of parts of a system to work together; interoperability measures the ability of a group of parts (constituting a system) to work with another system.

Software testability refers to the ease with which software can be made to demonstrate its faults through (typically execution-based) testing.

In particular, testability refers to the probability that, assuming that the software does have at least one fault, the software will fail on its next test execution.

Testability is related to the concepts of absorbability and coagulability. For a system to be properly testable, it must be possible to control each component's internal state and inputs and then to observe its outputs.

A system's testability relates to several structural or architectural issues: its level of architectural documentation, its separation of concerns, and the degree to which the system uses information hiding. Incremental development also benefits testability in the same way it enhances interoperability.

2.3 Business Quality Attributes

In addition to the preceding qualities that apply directly to a system, there are a number of ***business quality goals*** that frequently shape a system's architecture.

We (briefly) distinguish two kinds of business goals.

- The first concerns cost and schedule considerations;
- The other business goal deals with market and marketing considerations;

Time to market. If there is competitive pressure or if there is a short window of opportunity for a system or product, development time becomes important.

This in turn leads to pressure to buy or otherwise ***reuse existing components***. Time to market is often reduced by using prebuilt components such as commercial off-the-shelf (COTS) products or components reused from previous projects. The ability to insert a component into a system depends on the decomposition of the system into components, one or more of which are prebuilt.

Cost. The development effort will naturally have a budget that must not be exceeded.

Different architectures will yield different development costs; for instance, an architecture that relies on technology (or expertise with a technology) that is not resident within the developing organization will be more expensive to realize than one that takes advantage of assets already in-house.

Projected lifetime of the system. lf the system is intended to have a long lifetime, modifiability and portability across different platforms become important. But building in the additional infrastructure (such as a portability layer) to support modifiability and portability will usually compromise time to market.

On the other hand, a modifiable, extensible product is more likely to survive longer in the marketplace, extending its lifetime.

Targeted market. For general-purpose (mass-market) software, the platforms on which a system runs as well as its feature set will determine the size of the potential market. Thus, portability and functionality are key to market share. Other qualities such as performance, reliability, and usability also play a role.

For a large but specific market, a product-line approach should be considered, in which a core of the system is common (frequently including provisions for portability) and around which layers of software of increasing specificity are constructed.

Rollout schedule. lf a product is to be introduced as base functionality with many options, flexibility and customizability are important. Particularly, the system must be constructed with ease of expansion and contraction in mind.

Extensive use of legacy systems. If the new system must integrate with existing systems, care must be taken to define appropriate integration mechanisms.

This is a property that is clearly of marketing importance but which has substantial architectural implications.

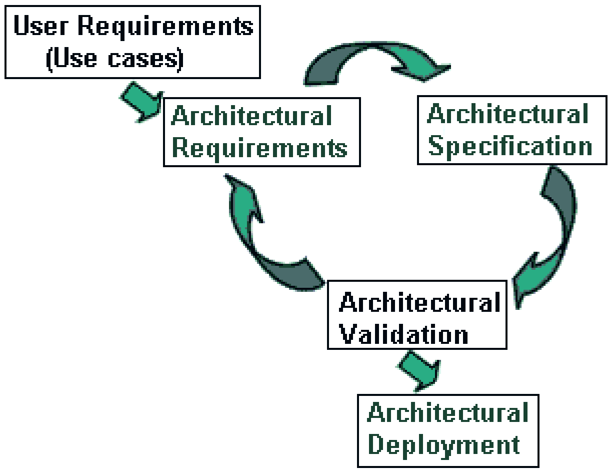
For example, the ability to integrate a legacy system with an HTTP server to make it accessible from the World Wide Web is currently a marketing goal in many corporations. The architectural constraints implied by this integration must be analyzed.

3 The Technical Architecting Process

The architecting process incorporates a ***technical process*** and an ***organizational process***.

- The technical process includes steps and heuristics for creating a good architecture.

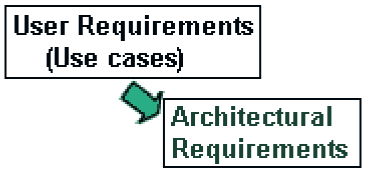
The focal deliverable of the architecting process is the ***architectural specification***, motivating and describing the structure of the system through various views. However, though system structuring is at the heart of the architecting process, it is just one of several activities critical to the creation of a good architecture.


User Requirements (Use cases)
Architectural Requirements
Architectural Specification
Architectural Validation
Architectural Deployment

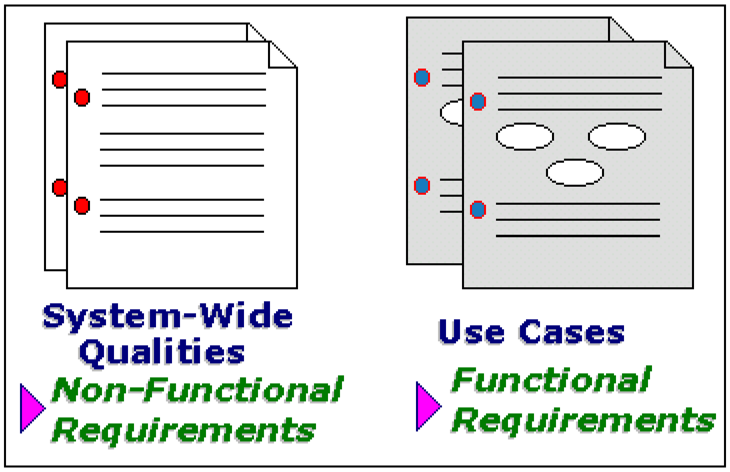
3.1 Architectural requirements

Architectural requirements are needed to focus the structuring activities. Different architectural approaches tend to yield differing degrees of fit to various system requirements, and evaluating alternatives or performing architectural tradeoff analyses are an important adjunct to the structuring phase.

Architectural requirements are a subset of the system requirements, determined by architectural relevance. The business objectives for the system, and the architecture in particular, are important to ensure that the architecture is aligned with the business agenda. The system context helps determine what is in scope and what is out of scope, what the system interface is, and what factors impinge on the architecture.



The system value proposition helps establish how the system will fit the users' agenda and top-level, high-priority goals. These goals are translated into a set of use cases, which are used to document functional requirements. The system structure fails if it does not support the services or functionality that users value, or if the qualities associated with this functionality inhibit user performance or are otherwise unsatisfactory.



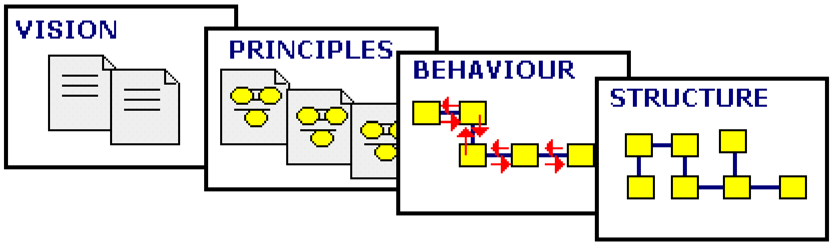
System qualities that have architectural significance (e.g., performance and security, but not usability at the user interface level) are therefore also important in directing architectural choices during structuring.
Of course, requirements may already have been collected by product teams. In that case, the architecture team needs to review those requirements for architectural relevance and completeness (especially with respect to non-functional requirements), and be concerned with requirements for future products that the architecture will need to support.

Lastly, for the architecture of a product line or family, architectural requirements that are unique to each product and those that are common across the product set need to be distinguished so that the structure can be designed to support both the commonality and the uniqueness in each product.

3.2 System Structuring

The architecture is created and documented in ***the system structuring phase***.

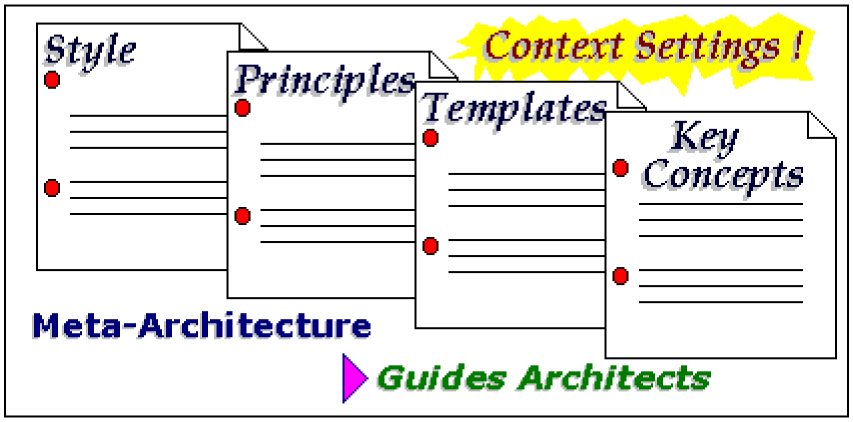
This is decomposed into sub-phases, along the lines of our model of software architecture:



First, the ***architectural vision*** is formulated, to act as a beacon guiding decisions during the rest of system structuring.

It is a good practice to explicitly allocate time for research in documented architectural styles, patterns, dominant designs and reference architectures, other architectures your organization, competitors, partners, or suppliers have created or you find documented in the literature, etc.

Based on this study, and your and the team's past experience, the meta-architecture is formulated. This includes the architectural style, concepts, mechanisms and principles that will guide the architecture team during the next steps of structuring.

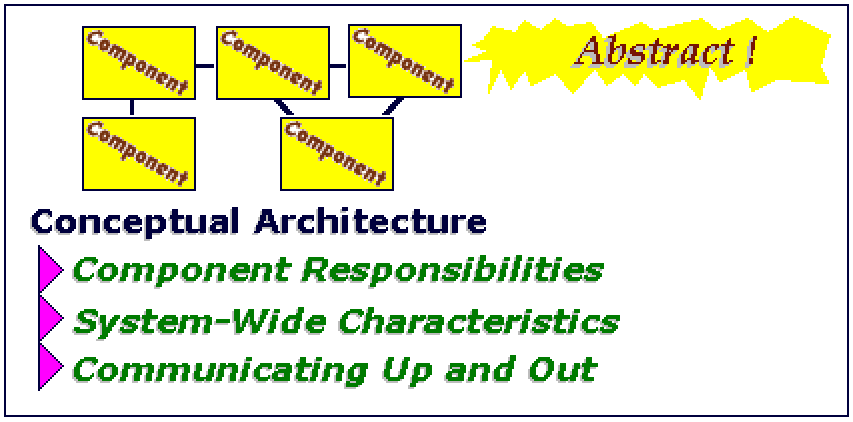


The system is

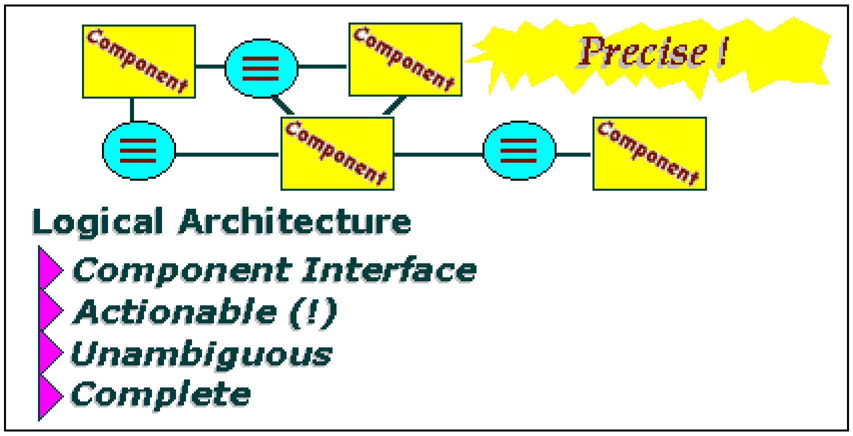
- decomposed into ***components*** and
- the ***responsibilities*** of each component, and ***interconnections*** between components are identified.

The intent of the conceptual architecture is to direct attention at an appropriate decomposition of the system without delving into the details of interface specification and type information.

Moreover, it provides a useful vehicle for communicating the architecture to non-technical audiences, such as management, marketing, and many users.



The conceptual architecture forms the starting point for the ***logical architecture***, and is likely to be modified as well as refined during the course of the creation of the logical architecture. Modeling the ***dynamic behavior*** of the system (at the architectural−or component−level) is a useful way to think through and refine the responsibilities and interfaces of the components.



Component specifications make the architecture concrete. These should include a summary description of services the component provides, the component owner's name, IID and version names, message signatures (IDL), a description of the operations, constraints or pre-post conditions for each operation (these may be represented in a state diagram), the concurrency model, constraints on component composition, a lifecycle model, how the component is instantiated, how it is named, a typical use scenario, a programming example, exceptions, and a test or performance suite.

An ***execution architecture*** is created for ***distributed*** or ***concurrent systems***.

It is formed by mapping the components onto the processes of the physical system.

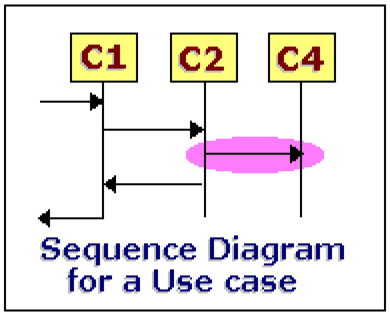
Different possible configurations are evaluated against requirements such as performance and scaling.

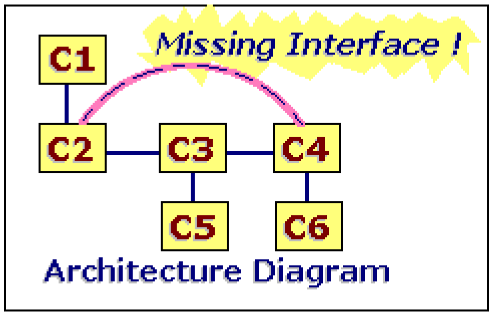
At each step in structuring, it is worthwhile challenging the team's creativity to expand the solution set under consideration, and then evaluating the ***different architecture alternatives*** against the prioritized architectural requirements.

This is known as ***architecture tradeoff analysis*** (Barbacci et. al., 1998), and it recognizes that different approaches yield differing degrees of fit to the requirements. Selection of the best solution generally involves some compromise, but it is best to make this explicit.

3.3 Architecture Validation

Lastly, a ***validation phase*** provides early indicators of, and hence an opportunity to resolve, problems with the architecture.





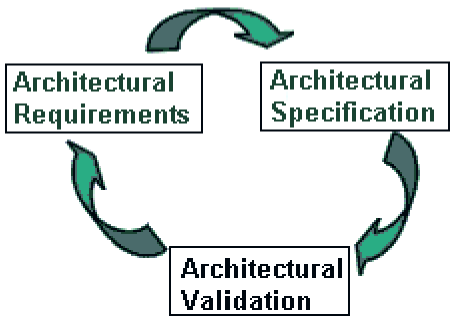
During structuring, the architects obviously make their best effort to meet the requirements on the architecture. The architecture validation phase involves ***additional people*** from outside the architecting team to help provide an objective assessment of the architecture. In addition to enhancing confidence that the architecture will meet the demands placed on it, including the right participants in this phase can help create buy-in to the architecture.

Architecture assessment involves:

- "thought experiments",
- modeling and walking-through scenarios that exemplify requirements,
- assessment by experts who look for gaps and weaknesses in the architecture based on their experience.

Another important part of validation is the development of prototypes or proofs-of-concept. Taking a skeletal version of the architecture all the way through to implementation, for example, is a really good way to prove out aspects of the architecture.

Though described sequentially above, the architecting process is best ***conducted iteratively***, with multiple cycles through requirements, structuring and validation.



One approach is to have at least one cycle devoted to each of Meta, Conceptual, Logical, and Execution architecture phases and cycles for developing the architectural guidelines and any other materials to help in deploying the architecture (such as tutorials). At each cycle, just enough requirements are collected to proceed with the next structuring step, and validation concentrates on the architecture in its current phase of maturity and depth.

Moreover, a number of architecture teams that we have worked with have stopped at different points, leaving more detailed architecting to the product and component teams.

At one end of the spectrum, a very small team of architects created the meta-architecture, and each of the product teams created their own architectures within the guidelines and constraints of the meta-architecture. Other architecture teams created the meta- and conceptual architectures, and a broader team of component owners developed the logical architecture.

At the other end of the spectrum, the architecture team developed the entire architecture, all the way to its detailed logical architecture specification. This approach yields the most control over the architecture specification, but is typically fraught with organizational issues (e.g., the "NIH syndrome") that slow or even completely inhibit the use of the architecture.

4 The Organizational Architecting Process

The architecting process incorporates a ***technical process*** and an ***organizational process***.

- However, a technically good architecture is not sufficient to ensure the successful use of the architecture, and the organizational process is oriented toward ensuring support for, and adoption of, the architecture.

Architecture projects are susceptible to three major organizational sources of failure:

- the project is under-resourced or cancelled prematurely by an uncommitted management;
- it is stalled with endless infighting or a lack of leadership;
- the architecture is ignored or resisted by product developers.

The organizational process helps address these pitfalls. Two phases − namely Init/Commit and Deployment − support the technical process.

However, the principal activities in these phases, namely championing the architecture and leading/teaming in Init/Commit, and consulting in Deployment, also overlap with the technical process activities.

The ***Init/Commit*** phase focuses on initiating the architecture project on a sound footing, and gaining strong commitment from upper management.

The creation of the architecture vision is central both to aligning the architecture team and gaining management sponsorship.

A communication plan is also helpful in sensitizing the team to the need for frequent communication with others in the organization.

A heads-down, hidden skunkworks architecture project may make quick progress − as long as it is well-led and its members act as a team. However, not listening to the needs of the management, developers, marketing, manufacturing and user communities and not paying attention to gaining and sustaining sponsorship in the management and technical leadership of the organization, or buy-in from the developer community, will lead to failure.

The communication plan places attention on balancing the need for communication and isolation, as well as planning what to communicate when, and to whom.

The ***Deployment phase*** follows the technical process, and addresses the needs of the developers who are meant to use the architecture to design and implement products. These range from understanding the architecture and its rationale, to responding to the need for changes to the architecture.

This entails consulting, and perhaps tutorials and demos, as well as the architects' involvement in design reviews.

It is important that at least the senior architect and the architecture project manager (if there is one) ***champion*** (fight for !) the architecture and gain the support of all levels of management affected by the architecture.

Championing the architecture starts early, and continues throughout the life of the architecture, though attention to championing tapers off as the architecture comes to be embraced by the management and developer communities.

For the architecture team to be successful, there must be a ***leader*** and the ***team members*** must collaborate to bring their creativity and experience to bear on creating an architecture that will best serve the organization.

This would seem so obvious as to not warrant being said, but unfortunately this is easier said than done. Explicit attention to developing the designated lead architect's leadership skills, in the same way one would attend to developing these skills in management, is a worthy investment.

Likewise, investing in activities aimed at developing the team as a team also has great payoff in the team's efficacy.

Consulting with and ***assisting*** the developer community in their use of the architecture is important in facilitating its successful adoption and appropriate use. These activities are most intense during deployment.

However, earlier communication and consulting helps create buy-in the developer community through participation and understanding. This allows the architecture team to understand the developers' needs and the developers to understand the architecture (and its rationale) as it evolves through the cycles of the technical process.

5 Architectural Styles

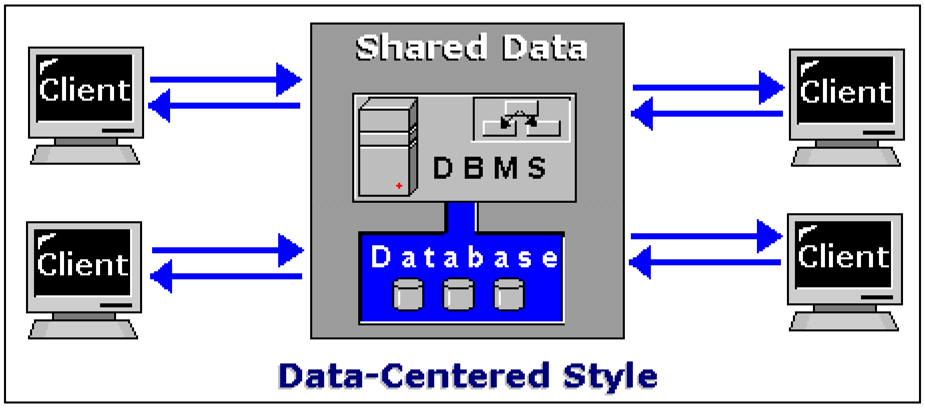
An ***architectural style*** in software consists of a few key features and rules for combining those features so that architectural integrity is preserved.
An architectural software style is determined by the following:

- set of component types (e.g., data repository, a process, a procedure) that perform some function at runtime
- topological layout of these components indicating their runtime interrelationships
- set of semantic constraints (for example, a data repository is not allowed to change the values stored in it)
- set of connectors (e.g., subroutine call, remote procedure call, data streams, sockets) that mediate communication, coordination, or cooperation among components.

5.1 Data Centered Architectures

Data-Centered architectures have the goal of achieving the quality of integrability of data.

The term ***Data-Centered Architectures*** refers to systems in which the access and update of a widely accessed data store is an apt description.



At its heart, it is nothing more than a centralized data store that communicates with a number of clients. The means of communication (sometimes called the coordination model) distinguishes the two subtypes: ***repository*** (the one shown) and ***blackboard***. A blackboard sends notification to subscribers when data of interest changes, and is thus active.

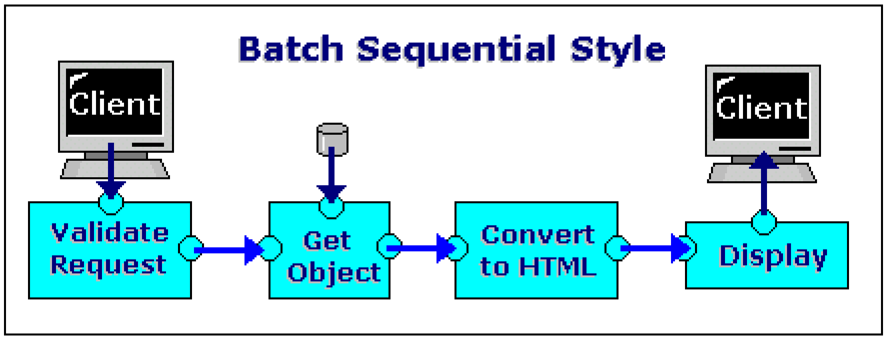
Data-centered styles are becoming increasingly important because they offer ***a structural solution to illegibility***. Many systems, especially those built from preexisting components, are achieving data integration through the use of blackboard mechanisms. They have the advantage that the clients are relatively independent of each other, and the data store is independent of the clients.

Thus, this ***style is scalable***: New clients can be easily added. It is also modifiable with respect to changing the functionality of any particular client because otherwise will not be affected. Coupling among clients will lessen this benefit but may occur to enhance performance.

5.2 Data-Flow Architectures

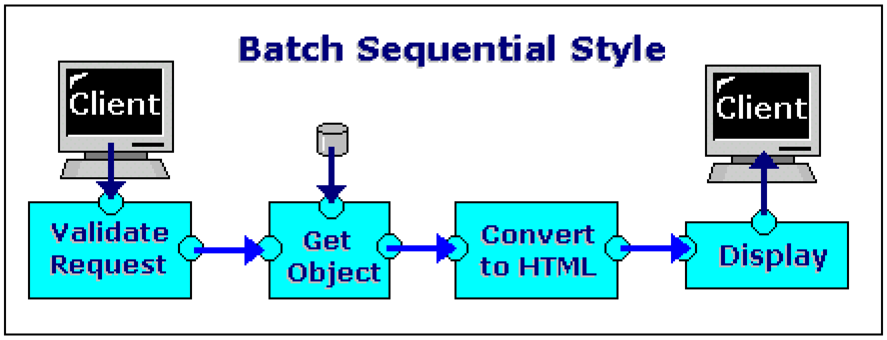
Data-Flow architectures have the goal of achieving the qualities of reuse and modifiability.

The data-bow style is characterized by viewing the system as a series of transformations on successive ***pieces of input data***. Data enter the system and then flows through the components one at a time until they are assigned to some final destination (output or a data store).

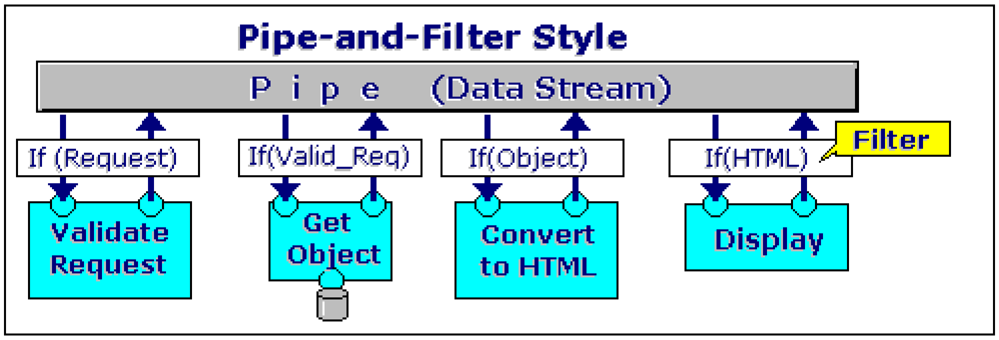


In the ***batch sequential style***, processing steps, or components, are independent programs, and the assumption is that each step runs to completion before the next step starts. Each batch of data is transmitted as a whole between the steps.

The typical application for this style is classical data processing.



The ***Pipe-and-Filter style*** emphasizes the incremental transformation of data by successive components. This is a typical style in the UNIX family of operating systems.



Filters are stream transducers. They incrementally transform data (stream to stream), use little contextual information, and retain no state information between instantiations. Pipes are stateless and simply exist to move data between filters.

Both pipes and alters are run non-deterministically until no more computations or transmissions are possible. Constraints on the pipe-and-alter style indicate the ways in which pipes and alters can be joined.

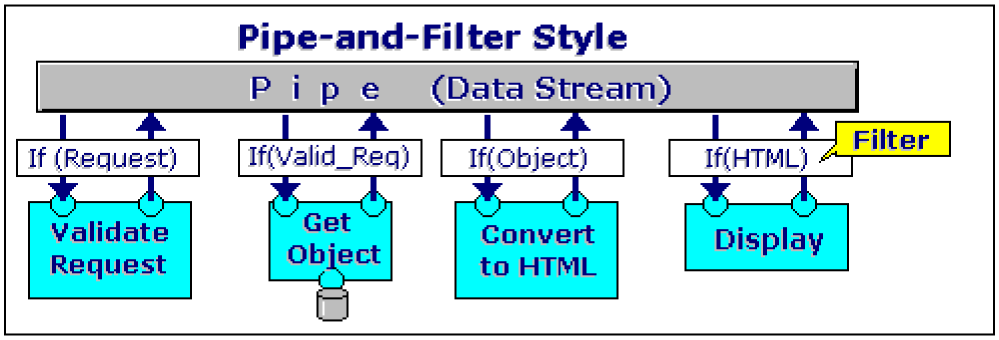
A pipe has a source end that can only be connected to a filter's output port and a sink end that can only be connected to a alter's input port.

Pipe-and-filter systems, like all other styles, have a number of advantages and disadvantages.

Their advantages principally flow from their simplicity-the limited ways in which they can interact with their environment.

This simplicity means that a pipe-and-filter system's function is no more and no less than the composition of the functions of its primitives.

There are no complex component interactions to manage.



Pipe-and-filter systems advantages:

- The pipe-and-filter style simplifies system maintenance and enhances reuse for the same reason-filters stand alone, and we can treat them as black boxes.
- Both pipes and filters can be hierarchically composed: Any combination of filters, connected by pipes, can be packaged and appear to the external world as a filter.
- Because a filter can process its input in isolation from the rest of the system, a pipe-and-filter system is easily made parallel or distributed, providing opportunities for enhancing a system's performance without modifying it.

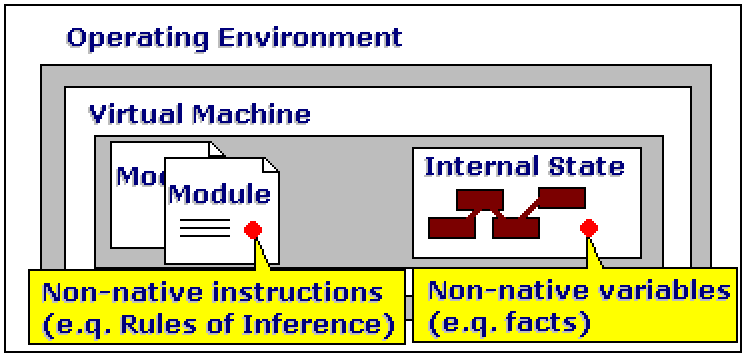
Pipe-and-filter systems also suffer from some ***disadvantages***.

- There is no way for filters to cooperatively interact to solve a problem.
- Performance in such a system is frequently poor for several reasons, all of which stem from the isolation of functionality that makes pipes and alters so modifiable; these reasons are listed below:
  - Filters typically force the lowest common denominator of data representation (such as an ASCII stream). lf the input stream needs to be transformed into tokens, every filter pays this parsing/unparsing overhead.
  - If a alter cannot produce its output until it has received all of its input, it will require an input buffer of unlimited size. A sort filter is an example of a filter that suffers from this problem. lf bounded buffers are used, the system could deadlock.

- Each filter operates as a separate process or procedure call, thus incurring some overhead each time it is invoked.

5.3 Virtual Machine Architecture

Virtual Machine architectures have the goal of achieving the quality of portability.
Virtual machines are software styles that simulate some functionality that is not native to the hardware and/or software on which it is implemented.


Operating Environment
Virtual Machine
Mo
Module
Internal State
Non-native instructions (e.g. Rules of Inference)
Non-native variables (e.g. facts)

This can be useful in a number of ways:

- It can allow one to simulate (and test) platforms that have not yet been built (such as new hardware), and it can simulate "disaster" modes (as is common in flight simulators and safety-critical systems) that would be too complex, costly, or dangerous to test with the real system.
- Common examples of virtual machines are interpreters, rule-based systems, syntactic shells, and command language processors.

Interpretation of a particular module via a ***Virtual Machine*** may be seen as follows:

- the interpretation engine selects an instruction from the module being interpreted;
- based on the instruction, the engine updates the virtual machine internal state;
- the process above is repeated;

Executing a module via a virtual machine adds flexibility through the ability to interrupt and query the program and introduce modifications at runtime, but there is a performance cost because of the additional computation involved in execution.

5.4 Call-and-Return Architectures

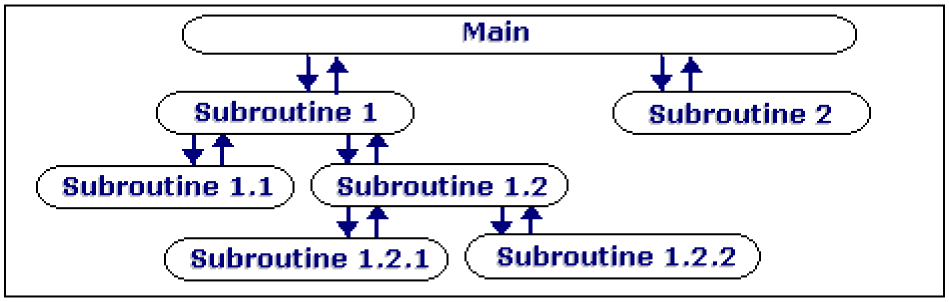
Call-and-Return architectures have the goal of achieving the qualities of modifiability and solvability.

Call-and-Return architectures have been the dominant architectural style in large software systems for the past 30 years.

However, within this style a number of substyles, each of which has interesting features, have emerged.

Main-Program-and-Subroutine architectures is the classical programming paradigm. The goal is to decompose a program into smaller pieces to help achieve modifiability.

A program is decomposed hierarchically. There is typically a single thread of control and each component in the hierarchy gets this control (optionally along with some data) from its parent and passes it along to its children.



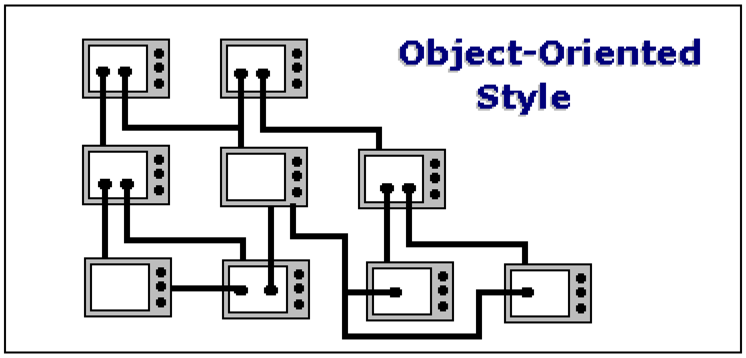
Remote procedure call systems are main-program-and-subroutine systems that are decomposed into parts that live on computers connected via a network.

The goal is to increase performance by distributing the computations and taking advantage of multiple processors. In remote procedure call systems, the actual assignment of parts to processors is deferred until runtime, meaning that the assignment is easily changed to accommodate performance tuning. In fact, except that subroutine calls may take longer to accomplish if it is invoking a function on a remote machine, a remote procedure call is indistinguishable from standard main program and subroutine systems.

Object-oriented or ***abstract data type*** systems are the modern version of call-and-return architectures.

The object-oriented paradigm, like the abstract data type paradigm from which it evolved, emphasizes the bundling of data and methods to manipulate and access that data (Public Interface).

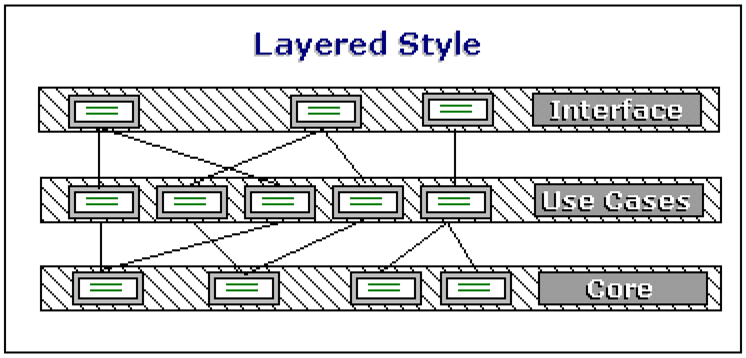
The object abstractions form components that provide black-box services and other components that request those services. The goal is to achieve the quality of modifiability.



This bundle is an encapsulation that hides its internal secrets from its environment. Access to the object is allowed only through provided operations, typically known as methods, which are constrained forms of procedure calls. This encapsulation promotes reuse and modifiability, principally because it promotes separation of concerns:

The user of a service need not know, and should not know, anything about how that service is implemented.

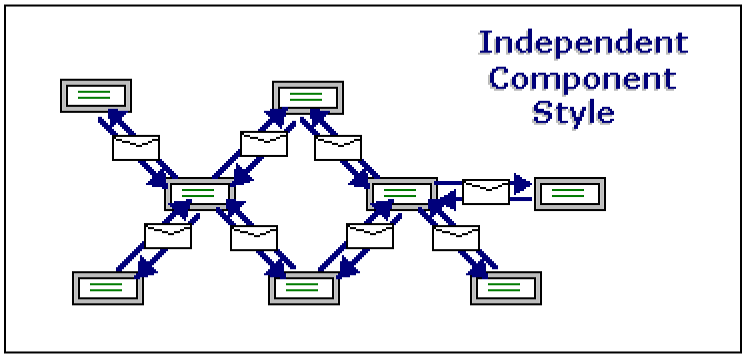
Layered systems are ones in which components are assigned to layers to control intercomponent interaction. In the pure version of this architecture, each level communicates only with its immediate neighbours.



The goal is to achieve the qualities of modifiability and, usually, portability. The lowest layer provides some core functionality, such as hardware, or an operating system kernel. Each successive layer is built on its predecessor, hiding the lower layer and providing some services that the upper layers make use of.

5.5 Independent Component Architectures

Independent component architectures consist of a number of independent processes or objects that communicate through messages.

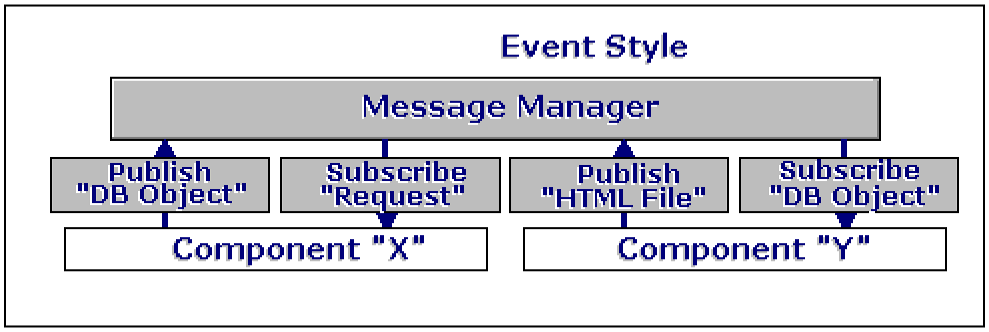


All of these architectures have the goal of achieving modifiability by decoupling various portions of the computations. They send data to each other but typically do not directly control each other. The messages may be passed to

- ***named participants*** (Communicating Processes style);
- ***unnamed participants*** using the publish/subscribe paradigm (Event Style) .

Event systems are a substyle in which control is part of the model. Individual components announce data that they wish to share (publish) with their ***environment − a set of unnamed components***.

These other components may register an interest in this class of data (subscribe). If they do so, when the data appears, they are invoked and receive the data.



Typically, event systems make use of a ***message manager*** that manages communication among the components, invoking a component (thus controlling it) when a message arrives for it. In this publish/subscribe paradigm, a message manager may or may not control the components to which it forwards messages.

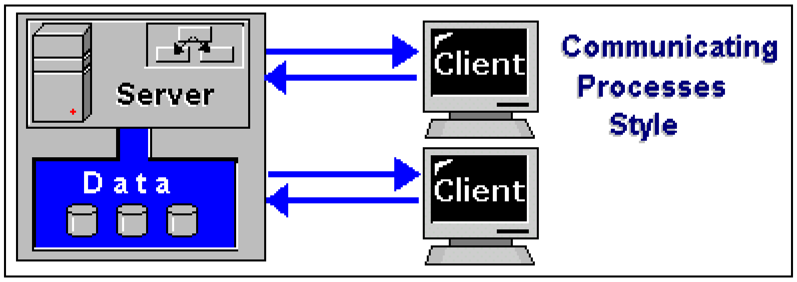
Components register types of information that they are willing to provide and that they wish to receive.

They then publish information by sending it to the message manager, which forwards the message, or in some cases an object reference, to all interested participants.

This paradigm is important because it decouples component implementations ***from knowing each others' names and locations***. As mentioned, it may involve decoupling control as well, which means that components can run in parallel, only interacting through an exchange of data when they so choose. This decoupling eases component integration

Besides event systems, the other substyle of independent components is the ***communicating processes style***. These are the classic multiprocess systems.

Of these, client-server is a well-known subtype. The goal is to achieve the quality of scalability.



A server exists to serve data to one or more clients, which are typically located across a network. The client originates a call to the server, which works, synchronously or asynchronously, to service the client's request.

If the server works synchronously, it returns control to the client at the same time that it returns data. If the server works asynchronously, it returns only data to the client (which has its own thread of control).

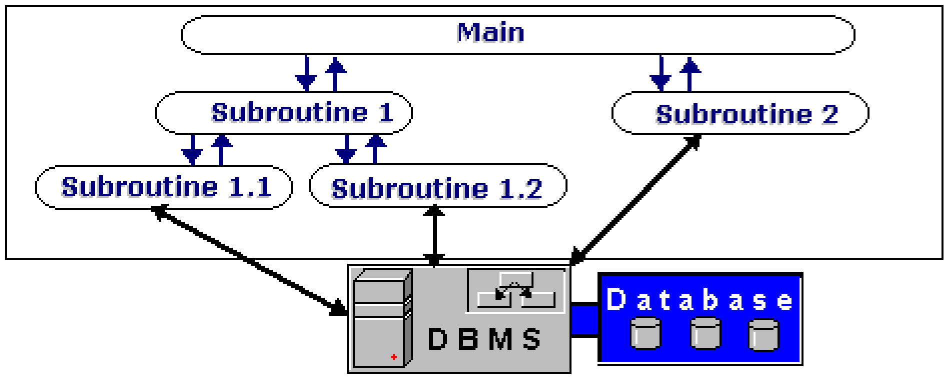
5.6 Heterogeneous Styles

Systems are seldom built from a single style, and we say that such systems are ***heterogeneous***.

There are three kinds of heterogeneity, they are as follows.

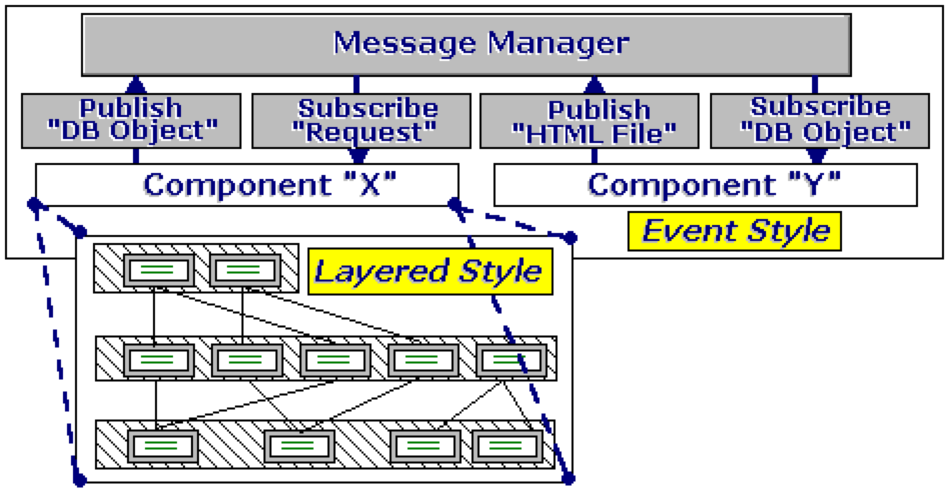
Locationally heterogeneous means that a drawing of its runtime structures will reveal patterns of different styles in different areas.

For example, some branches of a Main-Program-and-Subroutines system might have a shared data repository (i.e. a database).



Hierarchically Heterogeneous means that a component of one style, when decomposed, is structured according to the rules of a different style

For example, an end-user interface sub-system might be built using Event System architectural style, while all other sub-systems – using Layered Architecture.



Simultaneously Heterogeneous means that any of several styles may well be apt descriptions of the system.

This last form of heterogeneity recognizes that styles do not partition software architectures into non-overlapping, clean categories. You may have noticed this already. The data-centered style at the beginning of this discussion was composed of thread-independent clients, much like an independent component architecture.

The layers in a layered system may comprise objects or independent components or even subroutines in a main-program-and-subroutines system. The components in a pipe-and-filter system are usually implemented as processes that operate independently, waiting until input is at their ports, again, this is similar to independent component systems whose order of execution is predetermined.