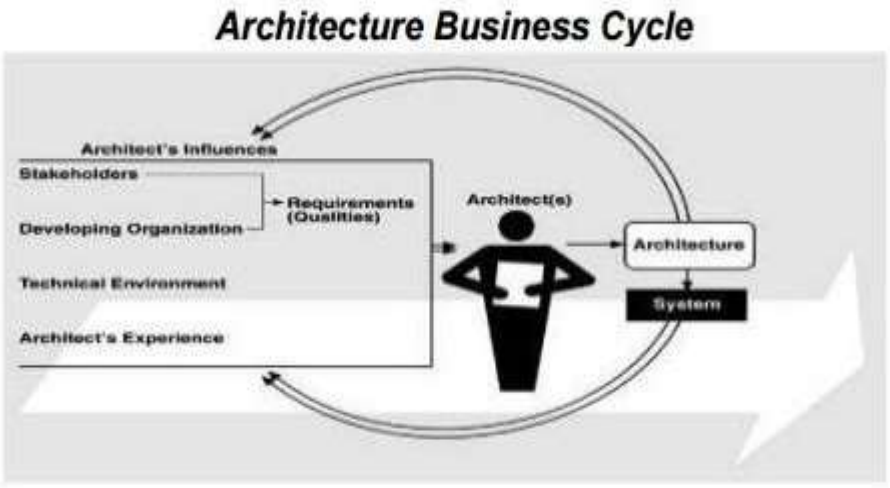
UNIT-I

1.1 Software Architecture Business Cycle

- An architecture is the result of a set of business and technical decisions.
- The requirements make explicit some—but only some—of the desired properties



Architectures are influenced by:

- **System stakeholders**
  - Stakeholders have different concerns (which may be contradictor)
  - Having an acceptable system involves properties such as
    - performance
    - reliability
    - availability
    - platform compatibility
    - memory utilization
    - network usage
    - security
    - modifiability
    - usability
    - interoperability with other systems
  - Often requirements document do not capture these properties well enough
  - Architects must identify and actively engage stakeholders early in the life cycle to
    - understand the real constraints of the system
    - manage the stakeholders' expectations
    - negotiate the system's priorities
    - make tradeoffs

- Architects need more than just technical skill: diplomacy, negotiation, and communication skills are essential

➢ **The Developing Organization**

- An architecture is influenced by the structure or nature of the development organization
- immediate business investment (eg. existing architecture)
- long-term business investment (eg. long term infrastructure)
- organizational structure (e.g. subcontracting, skills of employes)

➢ **The Background and Experience of the Architects**

- What worked in the past and what not
- Education, training, exposure to architectural patterns
- wish to experiment with something new

➢ **The Technical Environment**

- practices and techniques of the professional community

1.2 What Software Architecture Is

Definition:

The software architecture of a program or computing system is the structure or structures of the system, which comprise

- *software elements,*
- *the externally visible properties of those elements,*
- *and the relationships among them.*

"Externally visible" properties are those assumptions other elements can make of an element, such as its

- provided services
- performance characteristics,
- fault handling,
- shared resource usage, and so on.

Implications of this definition:

- Architecture defines software elements
- how the elements relate to each other
- an architecture is foremost an abstraction of a system that suppresses details of elements that do not affect how they use, are used by, relate to, or interact with other elements. (only public part)
- Systems comprise more than one structure
- no one structure can claim to be the architecture
- Relationships and elements might be
- runtime related ("send data", processes)

- non-runtime related ("sub model of", "inherits from", class)
- Every computing system with software has a software architecture
- it does not necessarily follow that the architecture is known to anyone.
- An architecture can exist independently of its description or specification
- The behavior of each element is part of the architecture
- allows elements to interact with each other
- Add specification and properties to the elements and relationships
- The definition is indifferent as to whether the architecture for a system is a good one or a bad one

1.3 Architectural Patterns, Reference Models, and Reference Architectures

- **Architectural Pattern:**

  A description of element and relation types together with a set of constraints on how they may be used.

  - A set of constraints on an architecture
  - Define a set or family of architectures
  - An architectural pattern is not an architecture
  - Exhibit known quality attributes
  - Often the architect's first major design choice
  - "Architectural style" has also been widely used
  - Example: "Client-Server"

- **Reference Model**:

  A division of functionality together with data flow between the pieces.

  - Standard decomposition of a known problem into parts that cooperatively solve the problem.
  - characteristic of mature domains
  - Example: Compiler; database management system

- **Reference Architecture**

  A reference model mapped onto software elements (that cooperatively implement the functionality defined in the reference model) and the data flows between them.

  - Whereas a reference model divides the functionality, a reference architecture is the mapping of that functionality onto a system decomposition.
  - Reference models, architectural patterns, and reference architectures are not architectures; they are useful concepts that capture elements of an architure.
  - Each is the outcome of early design decisions

1.4 Architectural Structures and Views

- Definition

  View

A representation of a set of elements and the relations among them.

A view is a representation of a set of system elements and relations among them not all system elements, but those of a particular type. For example, a layered view of a system would show elements of type "layer" that is, it would show the system's decomposition into layers and the relations among those layers. A pure layered view would not, however, show the system's services, or clients and servers, or data model, or any other type of element.

Views may be driven by the need to document a particular pattern in your design. Some patterns are composed of modules, others of components and connectors, and still others have deployment considerations. Module views, component-and-connector (C&C) views, and allocation views are the appropriate mechanisms for representing these considerations.

Module Views

A module is an implementation unit that provides a coherent set of responsibilities. A module might take the form of a class, a collection of classes, a layer, an aspect, or any decomposition of the implementation unit.

Example module views are decomposition, uses, and layers. Every module has a collection of properties assigned to it. These properties are intended to express the important information associated with the module, as well as constraints on the module.

Sample properties are responsibilities, visibility information, and revision history. The relations that modules have to one another include is part of, depends on, and is a.

Properties of modules that help to guide implementation or are input to analysis should be recorded as part of the supporting documentation for a module view. The list of properties may vary but is likely to include the following:

Name. A module's name is, of course, the primary means to refer to it. A module's name often suggests something about its role in the system. In addition, a module's name may reflect its position in a decomposition hierarchy; the name A.B.C, for example, refers to a module C that is a sub module of a module B, itself a sub module of A

Responsibilities. The responsibility property for a module is a way to identify its role in the overall system and establishes an identity for it beyond the name. Whereas a module's name may suggest its role, a statement of responsibility establishes it with much more certainty. Responsibilities should be described in sufficient detail to make clear to the reader what each module does.

Visibility of interface(s). When a module has submodules, some interfaces of the submodules are public and some may be private; that is, the interfaces are used only by the submodules within the enclosing parent module. These private interfaces are not visible outside that context.

Implementation information. Modules are units of implementation. It is therefore useful to record information related to their implementation from the point of view of managing their development and building the system that contains them. This might include the following:

Mapping to source code units. This identifies the files that constitute the implementation of a module. For example, a module Account, if implemented

in Java, might have several files that constitute its implementation: IAccount.java (an interface), Accountimpl.java (the implementation of Account functionality), AccountBean.java (a class to hold the state of an account in memory), AccountOrmMapping.xml (a file that defines the mapping between AccountBean and a database table object-relational mapping), and perhaps even a unit test AccountTest.j a va.

Test information. The module's test plan, test cases, test scaffolding, and test data are important to document. This information may simply be a pointer to the location of these artifacts.

Management information. A manager may need information about the module's predicted schedule and budget. This information may simply be a pointer to the location of these artifacts.

Implementation constraints. In many cases, the architect will have an implementation strategy in mind for a module or may know of constraints that the implementation must follow.

Revision history. Knowing the history of a module including authors and particular changes may help when you perfonn maintenance activities.

Component-and-Connector Views

Component-and-connector views show elements that have some runtime presence, such as processes, objects, clients, servers, and data stores. These elements are termed components. Additionally, component-and-connector views include as elements the pathways of interaction, such as communication links and protocols, information flows, and access to shared storage. Such interactions are represented as connectors in C&C views. Sample C&C views are service-oriented architecture (SOA), client-server, or communicating process views.

Components have interfaces called ports. A port defines a point of potential interaction of a component with its environment. A port usually has an explicit type, which defines the kind of behavior that can take place at that point of interaction. A component may have many ports of the same type, each forming a different input or output channel at runtime.

An element (component or connector) of a C&C view will have various associated properties. Every element should have a name and type. Additional properties depend on the type of component or connector. Define values for the properties that support the intended analyses for the particular C&C view. For example, if the view will be used for performance analysis, latencies, queue capacities, and thread priorities may be necessary. The following are examples of some typical properties and their uses:

Reliability. What is the likelihood of failure for a given component or connector? This property might be used to help determine overall system availability.

Performance. What kinds of response time will the component provide under what loads? What kind of bandwidth, latency, jitter, transaction volume, or throughput can be expected for a given connector? This property can be used with others to determine system-wide properties such as response times, throughput, and buffering needs.

Resource requirements. What are the processing and storage needs of a component or a connector? This property can be used to determine whether a proposed

hardware configuration will be adequate.

Functionality. What functions does an element perform? This property can be used to reason about overall computation performed by a system.

Security. Does a component or a connector enforce or provide security features, such as encryption, audit trails, or authentication? This property can be used to determine system security vulnerabilities.

Concurrency. Does this component execute as a separate process or thread? This property can help to analyze or simulate the performance of concurrent components and identify possible deadlocks. · · · - 0 A a

Modifiability. Does the messaging structure support a structure to cater for evolving data exchanges? Can the components be adapted to process those new messages? This property can be defined to extend the functionality of a component.

Tier. For a tiered topology, what tier does the component reside in? This property helps to define the build and deployment procedures, as well as platform requirements for each tier.

C&C views are commonly used to show to developers and other stakeholders how the system works -one can "animate" or trace through a C&C view, showing an end-to-end thread of activity. C&C views are also used to reason about runtime system quality attributes, such as performance and availability. In particular, a well-documented view allows architects to predict overall system properties such as latency or reliability, given estimates or measurements of properties of the individual elements and their interactions

Allocation Views

Allocation views describe the mapping of software units to elements of an environment in which the software is developed or in which it executes. The environment might be the hardware, the operating environment in which the software is executed, the file systems supporting development or deployment, or the development organizations.

Allocation views consist of software elements and environmental elements. Examples of environmental elements are a processor, a disk farm, a file or folder, or a group of developers.

Quality Views

Module, C&C, and allocation views are all structural views: They primarily show the structures that the architect has engineered into the architecture to satisfy functional and quality attribute requirements.

These views are excellent for guiding and constraining downstream developers, whose primary job it is to implement those structures. However, in systems in which certain quality attributes (or, for that matter, some other kind of stakeholder concerns) are particularly important and pervasive, structural views may not be the best way to present the architectural solution to those needs. The reason is that the solution may be spread across multiple structures that are inconvenient to combine (for example, because the element types shown in each are different).

Another kind of view, which we call a quality view, can be tailored for specific stakeholders or to address specific concerns. These quality views are formed by extracting the relevant pieces of structural views and packaging them together.

Here are five examples:

A **security view** can show all of the architectural measures taken to provide security. It would show the components that have some security role or responsibility, how those components communicate, any data repositories for security information, and repositories that are of security interest. The view's context information would show other security measures (such as physical security) in the system's environment. The behavior part of a security view would show the operation of security protocols and where and how humans interact with the security elements. It would also capture how the system would respond to specific threats and vulnerabilities.

A **communications view** might be especially helpful for systems that are globally dispersed and heterogeneous. This view would show all of the component-to-component channels, the various network channels, quality-of- service parameter values, and areas of concurrency. This view can be used to analyze certain kinds of performance and reliability (such as deadlock or race condition detection). The behavior part of this view could show (for example) how network bandwidth is dynamically allocated.

An **exception or error-handling view** could help illuminate and draw attention to error reporting and resolution mechanisms. Such a view would show how components detect, report, and resolve faults or errors. It would help identify the sources of errors and appropriate corrective · · · - 0 A a actions for each. Root- cause analysis in those cases could be facilitated by such a view.

A **reliability view** would be one in which reliability mechanisms such as replication and switchover are modeled. It would also depict timing issues and transaction integrity.

A **performance view** would include those aspects of the architecture useful for inferring the system's performance. Such a view might show network traffic models, maximum latencies for operations, and so forth.

Structure

The set of elements itself, as they exist in software or hardware

- Restrict our attention at any one moment to one (or a small number) of the software system's structures.
- To communicate meaningfully about an architecture, we must make clear which structure or structures we are discussing at the moment

Software structures:

Module structures

Elements are modules, which are units of implementation.

- What is the primary functional responsibility assigned to each module?
- What other software elements is a module allowed to use?
- What other software does it actually use?

- **Decomposition**
  - shows how larger modules are decomposed into smaller ones recursively
- **Uses**
  - The units are: modules, procedures or resources on the interfaces of modules

    - The units are related by the uses relation
- **Layered**
    - "uses relations" structured into layers
- **Class, or generalization**
    - shows the “inherits-from” or “is-an-instance-of” relations among the modules

Component-and-connector structures

Elements are runtime components (units of computation) and connectors (communication vehicles among components)

The relation is attachment, showing how the components and connectors are hooked together

- What are the major executing components and how do they interact?
- What are the major shared data stores?
- Which parts of the system are replicated?
- How does data progress through the system?
- What parts of the system can run in parallel?
- How can the system’s structure change as it executes?

- **Process, or communicating processes**
    - units are processes or threads that are connected with each other by communication, synchronization, and/or exclusion operations
- **Concurrency**
    - The units are components and the connectors are “logical threads”
    - A logical thread is a sequence of computation that can be allocated to a separate physical thread
- **Shared data, or repository**
    - This structure comprises components and connectors that create, store, and access persistent data
- **Client-server**
    - The components are the clients and servers, and the connectors are protocols and messages
- **Allocation structures**

  The relationship between the software elements and the elements in one or more external environments
    - What processor does each software element execute on?
    - In what files is each element stored during development, testing, and system building?
    - What is the assignment of software elements to development teams?
    - Deployment

        - Shows how software (usually a process from a component-and-connector view) is assigned to hardware-processing and communication elements
        - Relations are “allocated-to” and “migrates-to” if the allocation is dynamic
    - Implementation
        - how software elements (usually modules) are mapped to the file structure(s)
    - Work assignment
        - assigns responsibility for implementing and integrating the modules to development teams

Relating structures to each other

- Elements of one structure are related to elements of other structures
- Often the dominant structure is module decomposition, because it spawns the project structure
- Structures represent a powerful separation-of-concerns approach for creating the architecture
- Structures are basis for architecture documentation

1.5 Architecture and Quality Attributes

- Architecture is critical to the realization of many qualities of interest in a system, and these qualities should be designed in and can be evaluated at the architectural level.
- Architecture, by itself, is unable to achieve qualities. It provides the foundation for achieving quality
- Quality attributes are achieved through:
    - architecture AND
    - implementation / deployment
- Quality attribute requirements significantly influence the architecture
- The degree to which a system meets quality attribute requirements depends on architectural decisions
- The achievement of a quality attribute effects positive or negative the achievement of other quality
- attributes
    - security vs reliability
    - performance vs all other quality attributes
- If architecture decision have nothing to do with quality attribute, you may do somebody else job.

System Quality Attributes

- availability, modifiability, performance, security, testability, and usability

- Quality attribute names by themselves are not enough:
  - Quality attribute requirements are often non-operational (meaningless to say a system shall be modifiable)
  - To which quality a particular aspect belongs to
  - The vocabulary describing quality attributes varies widely.



Quality attribute Scenarios

- Quality attribute scenarios helps to describe quality attribute requirements
- A quality attribute scenario is a quality-attribute-specific requirement. It consists:
  - Source of stimulus
  - Stimulus
  - Environment.
  - Artifact (whole system or part of it)
  - Response
  - Response measure
- Distinguish:
  - general (quality attribute) scenarios (system independent)

  For the six parts, the range of values they can take is shown
  - concrete (quality attribute) scenarios (system specific)
- **Quality Attributes**:

  Usually, business considerations determine the qualities that must be accommodated in a system architecture.

  Too often, functionality overrides maintainability, portability, scalability, and other factors determining the long-term success of a project.

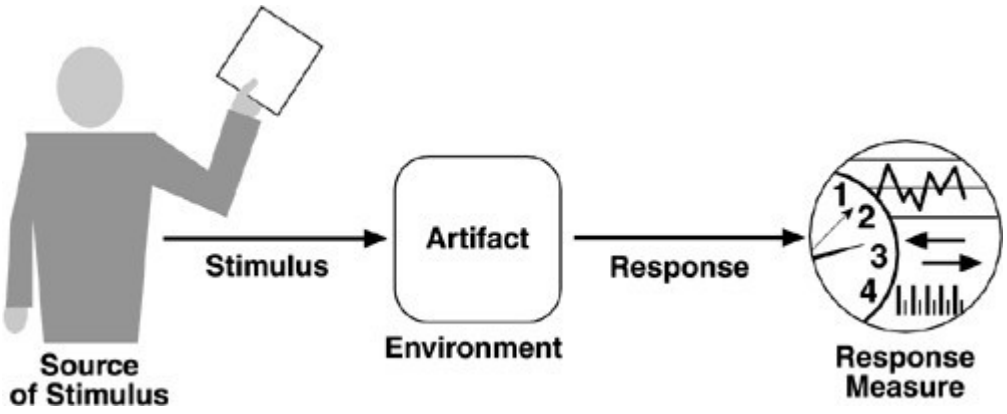
  Functionality and quality attributes are orthogonal, since a given functionality can be achieved by many different architectures.

  Quality requirements depend on the system architecture more than on the functional requirements.

  There are three main categories of quality attributes:
  - **System Qualities:** availability, modifiability, performance, security, testability, usability, others.
  - **Business Qualities:** time to market, cost and benefit, product lifetime, target market,

- roll-out schedule, integration, others.
- **Architectural Qualities:** conceptual integrity, correctness and completeness.

A quality attribute scenario has six parts, shown in the schematic:



- Source of Stimulus: the entity generating the stimulus. Could be an actor, an actuator, a sensor, and so on.
- Stimulus: a condition arriving at a system. Includes faults, stated intentions by actors, and so on.
- Environment: the conditions surrounding the stimulus. Might be normal operation, degraded operation, overload, and so on.
- Artifact: the part or parts of the system stimulated.
- Response: the response the system takes to the stimulus.
- Response Measure: how the response can be measured and test.

- **System Quality Attributes**:
  - Availability:

    The availability attribute is concerned with system failures. Faults are problems that are corrected or masked by the system. Failures are uncorrected errors that are user-visible.

    availability = [mean time to failure] / ([mean time to failure] + [mean time to repair])
  - Modifiability:

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

    The performance quality is concerned with response times and similar measures for various events.
  - Security:
    - Non-repudiation
    - Confidentiality
    - Integrity
    - Assurance or authenticity

        - Availability (no denial of service)
        - Auditing
    - Testability:

      The testability attribute is concerned with detecting failure modes. Typically, 40% of the cost of a large project is spent on testing. This means architectural support for testing that reduces test cost is time well spent. We need to control the internal state of and inputs to each unit, then observe the corresponding output of that unit.
    - Usability:
        - How easy it is to learn the features of the system
        - How efficiently the user can use the system
        - How well the system handles user errors
        - How well the system adapts to user needs
        - To what degree the system gives the user confidence in the correctness of its actions.
- **Business Quality Attributes**:
    - **Time to Market:** architectural reuse affects development time.
    - **Cost and Benefit:** in-house architectural expertise is cheaper than outside expertise.
    - **Projected Lifetime of the System:** long-lived systems require architectures that are
    - Modifiable and scalable.
    - **Targeted Market:** architecture affects what platforms will be compatible and incompatible with the system.
    - **Roll-out Schedule:** if functionality is planned to increase over time, the architecture needs to be customisable and flexible.
    - **Integration with Legacy Systems:** the architecture of the legacy system being integrated will influence the overall system's architecture.
- **Architectural Quality Attributes**:
    - Conceptual Integrity is the underlying vision or theme unifying the components and their interactions. The architecture should do similar things in similar ways.
    - Correctness and Completeness is concerned with checking the architecture for errors and omissions.
    - Buildability is concerned with the organization's capabilities to actually construct the architecture in question.

1.2 Achieving Qualities

Scenarios help describe the qualities of a system, but they don't describe how they will be achieved. Architectural tactics describe how a given quality can be achieved. For each quality there may be a large set of tactics available to an architect. It is the architect's job to select the right tactic in light of the needs of the system and the environment. For example, a performance tactics may include options to develop better processing algorithms, develop a system for parallel processing, or revise event scheduling policy. Whatever tactic is chosen, it

must be justified and documented.

Taxonomy of Software Qualities

Over the past 10 years software architects have devised a list of common software qualities. It would be naïve to claim that the list below is as a complete taxonomy of all software qualities – but it's a solid list of general software qualities compiled from respectable sources. Domain specific systems are likely to have an additional set of qualities in addition to the list below.

System qualities can be categorized into four parts: runtime qualities, non-runtime qualities, business qualities, and architecture qualities. Each of the categories and its associated qualities are briefly described below. Other articles on this site provide more information about each of the software quality attributes listed below, their applicable properties, and the conflicts the qualities.

Runtime System Qualities

Runtime System Qualities can be measured as the system executes.

Functionality

Definition: the ability of the system to do the work for which it was intended.

Performance

Definition: the response time, utilization, and throughput behavior of the system. Not to be confused with human performance or system delivery time.

Security

Definition: a measure of system's ability to resist unauthorized attempts at usage or behavior modification, while still providing service to legitimate users.

Availability *(Reliability quality attributes falls under this category)*

Definition: the measure of time that the system is up and running correctly; the length of time between failures and the length of time needed to resume operation after a failure.

Usability

Definition: the ease of use and of training the end users of the system. Sub qualities: learnability, efficiency, affect, helpfulness, control.

Interoperability

Definition: the ability of two or more systems to cooperate at runtime

Non-Runtime System Qualities

Non-Runtime System Qualities cannot be measured as the system executes.

Modifiability

Definition: the ease with which a software system can accommodate changes to its software

Portability

Definition: the ability of a system to run under different computing environments.

The environment types can be either hardware or software, but is usually a combination of the two.

> ***Reusability :****Definition: the degree to which existing applications can be reused in new applications.*

Integrability: *Definition: the ability to make the separately developed components of the system work correctly together.*

Testability

Definition: the ease with which software can be made to demonstrate its faults

Business Qualities

Non-Software System Qualities that influence other quality attributes.

Cost and Schedule

Definition: the cost of the system with respect to time to market, expected project lifetime, and utilization of legacy and COTS systems.

Marketability

Definition: the use of the system with respect to market competition.

Appropriateness for Organization

Definition: availability of the human input, allocation of expertise, and alignment of team and software structure. Business process re-engineering

Architecture Qualities

Quality attributes specific to the architecture itself.

Conceptual Integrity

Definition: the integrity of the overall structure that is composed from a number of small architectural structures.

Correctness

Definition: accountability for satisfying all requirements of the system.

Domain Specific Qualities

Quality attributes found in specific business domains.

Sensitivity

Definition: the degree to which a system component can pick up something being measured.

Calibrability

Definition: ability of a system to recalibrate itself to some specific working range.

1.3 Architectural Styles and patterns

Architectural patterns

An architectural pattern in software, also known as an architectural style, is analogous to an architectural style in buildings, such as Gothic or Greek Revival or Queen Anne. It consists of a few key features and rules for combining them so that architectural integrity is preserved. An architectural pattern is determined by:

- A set of element types (such as a data repository or a component that computesa mathematical function).
- A topological layout of the elements indicating their interrelation-ships.
- A set of semantic constraints (e.g., filters in a pipe-and-filter style are pure data transducers? they incrementally transform their input stream into an output stream, but do not

control either upstream or downstream elements).

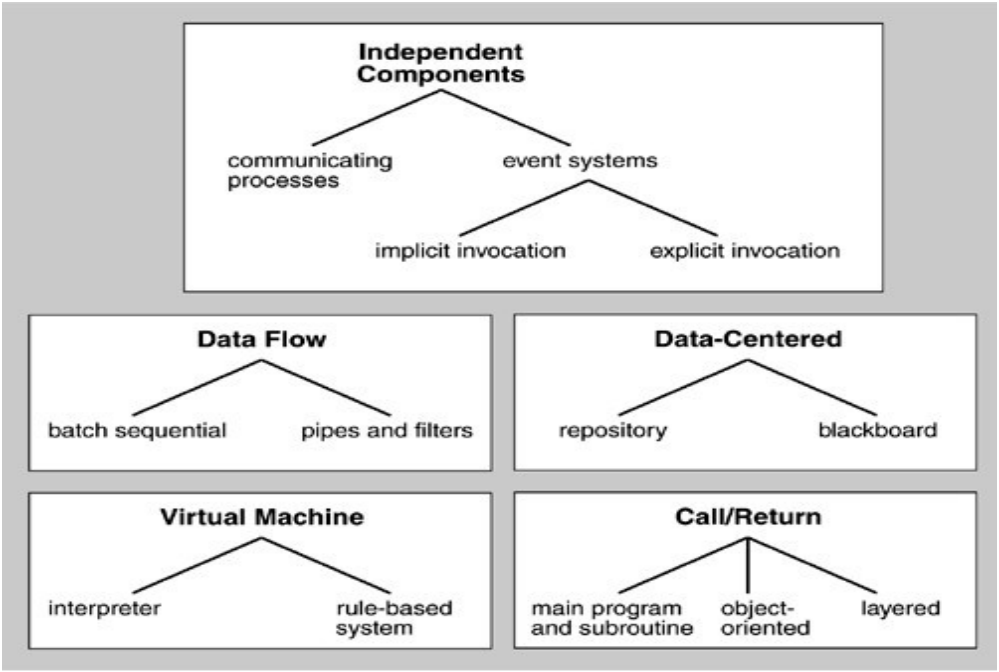
- A set of interaction mechanisms (e.g., subroutine call, event-subscriber, blackboard) that determine how the elements coordinate through the allowed topology.

Mary Shaw and David Garlan's influential work attempted to catalog a set of architectural patterns that they called architectural styles or idioms. This has been evolved by the software engineering community into what is now more commonly known as architectural patterns, analogous to design patterns and code patterns.

The motivation of [Shaw 96] for embarking on this project was the observation that high-level abstractions for complex systems exist but we do not study or catalog them, as is common in other engineering disciplines.

These patterns occur not only regularly in system designs but in ways that sometimes prevent us from recognizing them, because in different disciplines the same architectural pattern may be called different things. In response, a number of recurring architectural patterns, their properties, and their benefits have been cataloged. One such catalog is illustrated in the below figure

Figure . A small catalog of architectural patterns, organized by is-a relations



In this figure patterns are categorized into related groups in an inheritance hierarchy. For example, an event system is a substyle of independent elements. Event systems themselves have two subpatterns: implicit invocation.

What is the relationship between architectural patterns and tactics? As shown earlier, we view a tactic as a foundational "building block" of design, from which architectural patterns and strategies are created.

Software architecture is the blueprint of a software system. In this, we will learn about the different styles of software architecture like data-centric, layered and object-oriented styles.

Every software requires a proper plan and detailed blueprint before stepping into the development. Software architecture is the high level structure used for creating software systems and is actually a step-by-step blueprint of the entire software that is to be built. The purpose of the software and its specific functionalities are defined by the software's

architectural style and **pattern** used.

Architectural style

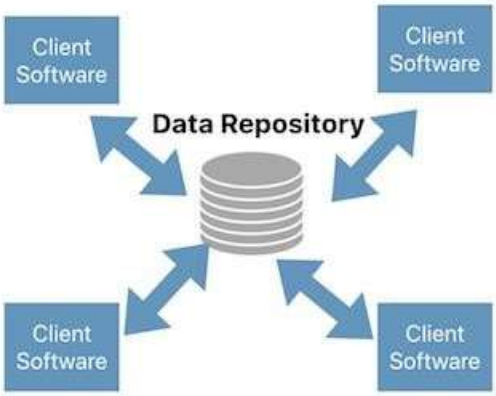
The architectural style is a very **specific solution** to a particular software which typically focuses on **how to organize the code** created for the software. It is the **granularity of highest level** that focuses on creating the layers and modules of the software and allowing an appropriate interaction between the various modules for giving the right results upon implementation.

The architectural pattern is the description of **relationship types** and **elements** along with a set of constraints to implementing a software system. The patterns are usually resuable solutions for common problems or models.

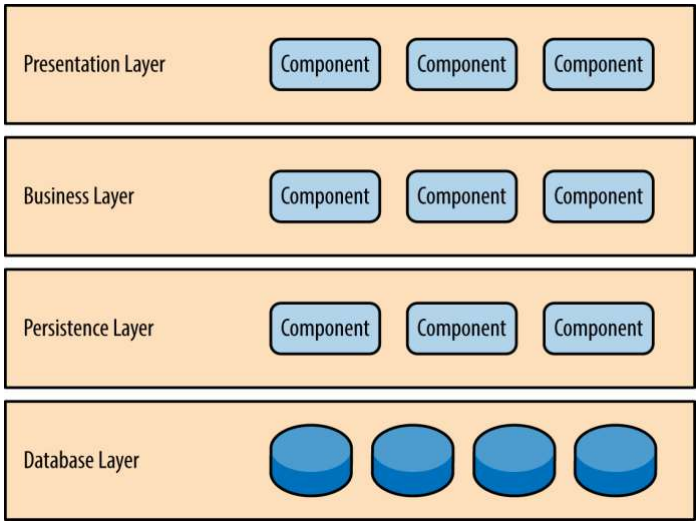
There are various types of architectural styles followed for software creation. Here, we discuss data-centric, object-oriented, and layered architectural patterns.

Data-Centric Style

As the name suggests, the data-centric architecture style has centralized data that is used by various components of the software, and these components access the data repositories of the software. The best example for such an architectural style is the relational database system that follows a database schema in which data stored in various tables inside the database is accessible by various components used in the software. Think about an inventory management software with huge amount of data to handle. Typically this type of a business requirement would use the data-centric style.



Layered Architecture Style



Also known as n-tier architectural style, it is one of the most common styles used in software development life cycle. This style is divided into various horizontal layers and each

layer has some specific function. Generally, this architectural style has four layers namely presentation, business, persistence, and database, where each layer has a different function. Presentation layer deals with the user interface, the business layer is responsible for the various business rules and conditions, persistence layer makes data accessible to the other layers and the database layer stores and provides data to the other components. Large business organizations with a focus on user interface, use this architectural style. Because of the layered approach, different teams can be allocated to work on the different layers.

Object- Oriented Architectural Style

Object Oriented Architecture is an important concept for developing the software. It is a design paradigm based on the division of responsibilities for an application or system into individual reusable and self-sufficient objects. The popular approach of object-oriented design is to view a software system as a collection of entities known as objects. Object oriented is based on modeling real-world objects.

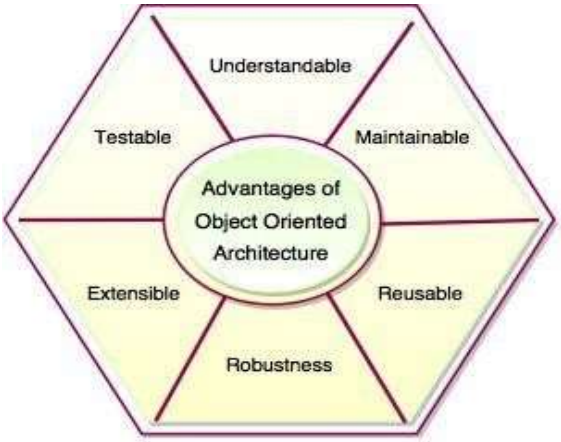


Fig. Advantages of Object Oriented Architecture

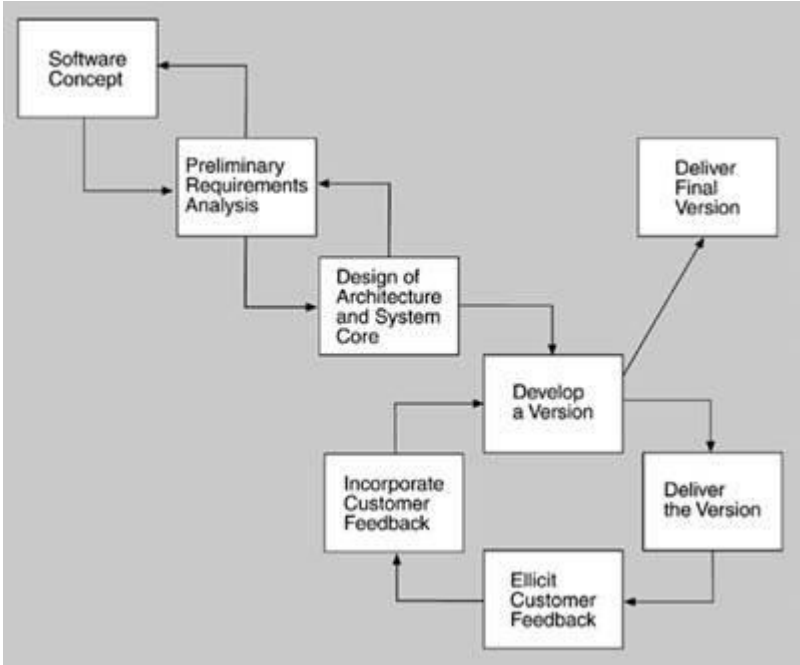
Advantages of Object-Oriented Architecture

- Object-Oriented architecture maps the application to real world objects for making it more understandable.
- It is easy to maintain and improves the quality of the system due to program reuse.
- This architecture provides reusability through polymorphism and abstraction.
- It has ability to manage the errors during execution. (Robustness)
- It has ability to extend new functionality and does not affected on the system.
- It improves testability through encapsulation.
- Object-Oriented architecture reduces the development time and cost.

1.4 Designing the Architecture

1. Architecture in the Life Cycle

Evolutionary Delivery Life Cycle



2. Attribute-Driven Design (ADD)

· ADD is an approach to defining a software architecture that bases the decomposition process on the quality attributes the software has to fulfill.

· It is a recursive decomposition process where, at each stage, tactics and architectural patterns are chosen to satisfy a set of quality scenarios and then functionality is allocated to instantiate the module types provided by the pattern.

· The output of ADD is the first several levels of a module decomposition view of an architecture and other views as appropriate.

· The input to ADD is a set of requirements. ADD assumes functional requirements (typically expressed as use cases) and constraints as input, as do other design methods. ADD mandates that quality requirements be expressed as a set of system-specific quality scenarios. Functional requirements, constraints, and quality scenarios are architectural drivers.

· ADD Steps

(1) Choose the module to decompose. The module to start with is usually the whole system. All required inputs for this module should be available (constraints, functional requirements, quality requirements).

(2) Refine the module according to these steps:

a. Choose the architectural drivers from the set of concrete quality scenarios and functional requirements. This step determines what is important for this decomposition.

b. Choose an architectural pattern that satisfies the architectural drivers. Create (or select) the pattern based on the tactics that can be used to achieve the drivers. Identify child modules required to implement the tactics.

c. Instantiate modules and allocate functionality from the use cases and represent using multiple views.

d. Define interfaces of the child modules. The decomposition provides

modules and constraints on the types of module interactions. Document this information in the interface document for each module.

e. Verify and refine use cases and quality scenarios and make them constraints for the child modules. This step verifies that nothing important was forgotten and prepares the child modules for further decomposition or implementation.

(3) Repeat the steps above for every module that needs further decomposition

3. Forming the Team Structure

· Organizing team structure to mirror architectural structure - modules.

· Assign team members according to their expertise on development of the module that is assigned to the team.

4. Creating a Skeletal System

· Once an architecture is sufficiently designed and teams are in place to begin building to it, a skeletal system can be constructed. The idea at this stage is to provide an underlying capability to implement a system's functionality in an order advantageous to the project.

· Classical software engineering practice recommends "stubbing out" sections of code so that portions of the system can be added separately and tested independently. However, which portions should be stubbed? By using the architecture as a guide, a sequence of implementation becomes clear.

a. Implement the software that deals with the execution and interaction of architectural components.

b. Choose which of the elements providing functionality should be added to the system. The choice may be based on lowering risk by addressing the most

c. problematic areas first, or it may be based on the levels and type of staffing available, or it may be based on getting something useful to market as quickly as possible.

d. Employ the uses structure to tell you what additional software should be running correctly in the system to support that functionality.

e. This process continues, growing larger and larger increments of the system, until it is all in place.

1.5 Documenting the Software Architecture

Documentation speaks for the architect. It speaks for the architect today, when the architect should be doing other things besides answering a hundred questions about the architecture. And it speaks for the architect tomorrow, when he or she has left the project and now someone else is in charge of its evolution and maintenance.

Fundamentally, architecture documentation has three uses:

1. Architecture documentation serves as a means of education. The educational use consists of introducing people to the system. The people may be new members of the team, external analysts, or even a new architect. In many cases, the "new" person is the customer to whom you're showing your solution for the first time, a presentation you hope will result in funding or go-ahead approval.

2. Architecture documentation serves as a primary vehicle for communication among stakeholders. An architecture's precise use as a communication vehicle depends on which stakeholders are doing the communicating.

3. Architecture documentation serves as the basis for system analysis and construction. Architecture tells implementers what to implement. Each module has interfaces that must be provided and uses interfaces from other modules. Not only does this provide instructions about the provided and used interfaces, but it also determines with what other teams the development team for the module must communicate.

1.6 Reconstructing Software Architecture

Architecture reconstruction is an interpretive, interactive, and iterative process involving many activities; it is not automatic. It requires the skills and attention of both the reverse engineering expert and the architect (or someone who has substantial knowledge of the architecture), largely because architectural constructs are not represented explicitly in the source code. There is no programming language construct for "layer" or "connector" or other architectural elements that we can easily pick out of a source code file. Architectural patterns, if used, are seldom labeled. Instead, architectural constructs are realized by many diverse mechanisms in an implementation, usually a collection of functions, classes, files, objects, and so forth. When a system is initially developed, its high-level design/architectural elements are mapped to implementation elements. Therefore, when we reconstruct those elements, we need to apply the inverses of the mappings. Coming up with those requires architectural insight.

Familiarity with compiler construction techniques and utilities such as grep, sed, awk, perl, python, and lex/yacc is also important.

The results of architectural reconstruction can be used in several ways. If no documentation exists or if it is out of date, the recovered architectural representation can be used as a basis for redocumenting the architecture. This approach can also be used to recover the as-built architecture, to check conformance against an "as-designed" architecture. This assures us that our maintainers (or our developers, for that matter) have followed the architectural edicts set forth

for them and are not eroding the architecture, breaking down abstractions, bridging layers, compromising information hiding, and so forth. The reconstruction can also be used as the basis for analyzing the architecture or as a starting point for re-engineering the system to a new desired architecture. Finally, the representation can be used to identify elements for re-use or to establish an architecture-based software product line.

Architecture reconstruction has been used in a variety of projects ranging from MRI scanners to public telephone switches and from helicopter guidance systems to classified NASA systems. It has been used

- to redocument architectures for physics simulation systems.
- to understand architectural dependencies in embedded control software for mining machinery.
- to evaluate the conformance of a satellite ground system's implementation to its reference architecture.
- to understand different systems in the automotive industry.

Reconstruction Activities

Software architecture reconstruction comprises the following activities, carried out iteratively:

1. **Information extraction:** The purpose of this activity is to extract information from various sources.

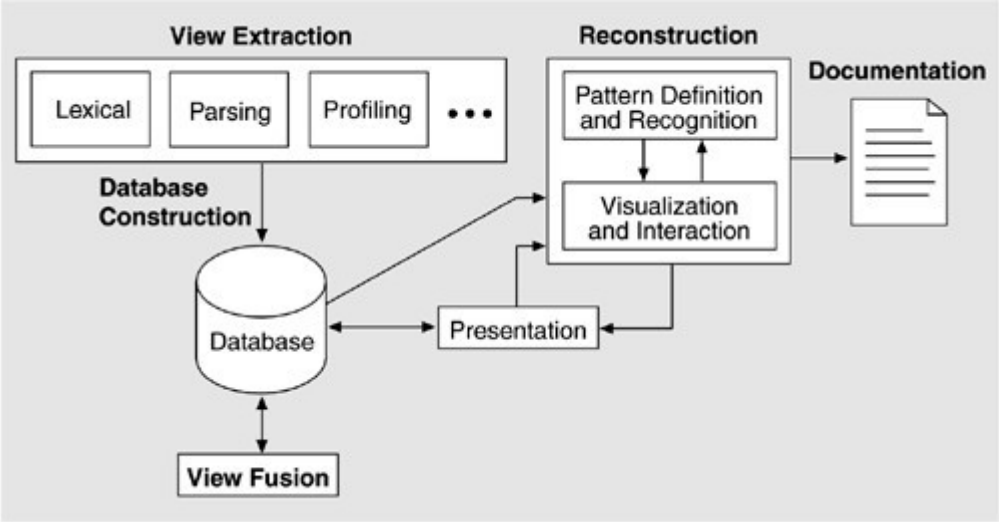
2. **Database construction:** Database construction involves converting this information into a standard form such as the Rigi Standard Form (a tuple-based data format in the form of relationship <entity1> <entity2>) and an SQL-based database format from which the database is created.

3. **View fusion:** View fusion combines information in the database to produce a coherent view of the architecture.

4. **Reconstruction:** The reconstruction activity is where the main work of building abstractions and various representations of the data to generate an architecture representation takes place.

These activities are highly iterative. The following figure depicts the architecture reconstruction activities and how information flows among them.

Architecture reconstruction activities. (The arrows show how information flows among the activities.)



The reconstruction process needs to have several people involved. These include the person doing the reconstruction (reconstructor) and one or more individuals who are familiar with the system being reconstructed (architects and software engineers).

The reconstructor extracts the information from the system and either manually or with the use of tools abstracts the architecture from it. The architecture is obtained by the reconstructor through a set of hypotheses about the system. These hypotheses reflect the inverse mappings from the source artifacts to the design (ideally the opposite of the design mappings). They are tested by generating the inverse mappings and applying them to the extracted information and validating the result. To most effectively generate these hypotheses and validate them, people familiar with the system must be involved, including the system architect or engineers who have worked on it (who initially developed it or who currently maintain it).