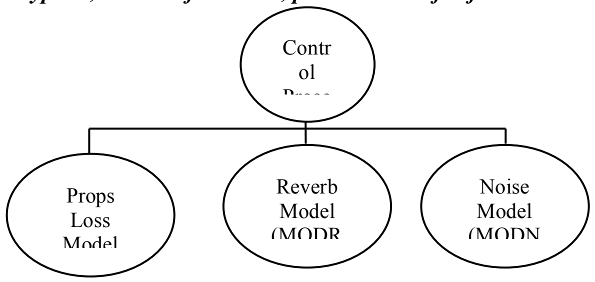
UNIT – I INTRODUCTION AND ARCHITECTURAL DRIVERS

PART- A

1. **What is software Architecture?**
The software architecture of a program or system is the structure or structures of the system, which comprise software components, the externally visible properties of these components, and the relationships among them. Architecture plays a pivotal role in allowing an organization to meet its business goals.
2. **Name some of the implications of software Architecture definition.**
    - Architecture defines software elements
    - The definition makes clear that systems can and do comprise more than one structure and that no one structure can irrefutably claim to be the architecture.
    - The definition implies that every computing system with software has a software architecture because every system can be shown to comprise elements and the relations among them.
3. **What is not Software Architecture? Give an Example.**
An architecture which does not tell the nature and responsibilities of the elements, significance of the connections and layout.
Eg: system description for an underwater acoustic simulation, purport to describe that system's "top-level architecture" and is precisely the kind of diagram most often displayed to help explain an architecture.

Typical, but uninformative, presentation of software architecture



4. **What Drives Software Architecture?**
System requirements:
    - Functional needs (what the system should do)
    - Quality needs (properties that the system must possess such as availability, performance, security)

Design constraints
    - Development process
    - Technical constraints
    - Business constraints
    - Contractual requirements
    - Legal obligations
    - Economic factors
5. **Define any one of the standard definition of Software Architecture.**
Architecture is the structure of the components of a program or system, their interrelationships, and the principles and guidelines governing their design and evolution over time. Any system has an architecture that can be discovered and analyzed independently of any knowledge of the process by which the architecture was designed or evolved.
6. **What Architecture Should Convey?**
    - Architecture defines components
    - Systems can comprise more than one structure and no one structure is the architecture
    - The behavior of each component is part of the architecture
7. **Why is Software Architecture important?**
    - Communication among stake holders
    - Early design decisions
    - Transferable abstraction of a system

8. **Define Architectural Drivers**

   Architectural drivers are the design forces that will influence the early design decisions the architects make. Architectural drivers are the combination of functional and quality requirements that "shape" the architecture or the particular module under consideration. The drivers will be found among the top-priority requirements for the module.

9. **What is an Architectural Structure?**

   A view is a representation of a coherent set of architectural elements, as written by and read by system stakeholders. It consists of a representation of a set of elements and the relations among them. A structure is the set of elements itself, as they exist in software or hardware.

   Example: A module structure is the set of the system's modules and their organization.

10. **What are the three groups of Architectural Structures?**
    - Module structures
    - Allocation structures
    - Component-and-connector structures

11. **Where do Architectures come from?**

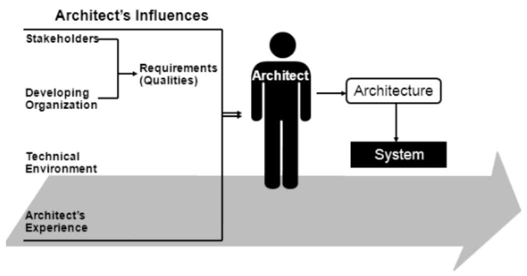
    Architecture is the result of a set of business and technical decisions. There are many influences at work in its design, and the realization of these influences will change depending on the environment in which the architecture is required to perform.

12. **Define Stake holders.**

    Many people and organizations interested in the construction of a software system are referred to as stakeholders. E.g. customers, end users, developers, project manager etc.

13. **What are the three classes of influence of Software Architecture that come from the developing organization?**
    - Immediate business
    - Organizational structure
    - Long-term business

14. **Draw a neat sketch describing the influences on Architecture.**

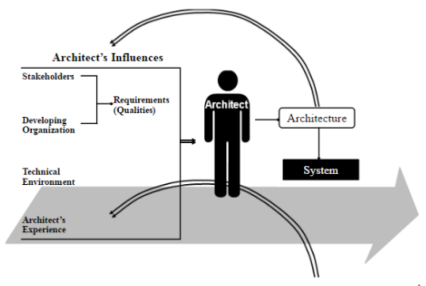


15. **What are the factors influencing Architecture?**
    - Stakeholders of a system
    - Architect's background
    - Technical and Organizational factors

16. **Define Architecture Business Cycle (ABC).**

    The Relationships among business goals, product requirements, architects experience, architectures and fielded systems form a cycle with feedback loops that a business can manage is known as ABC.

17. **Draw the working of ABC.**



18. Define Software Process.

Software process is the term given to the organization, ritualization, and management of software development activities.

19. What are the various activities involved in creating software architecture?

- Creating the business case for the system
- Understanding the requirements
- Creating or selecting the architecture
- Documenting and communicating the architecture
- Analyzing or evaluating the architecture
- Implementing the system based on the architecture
- Ensuring that the implementation conforms to the architecture

20. What makes a "GOOD" architecture?

Given the same technical requirements for a system, two different architects in different organizations will produce different architectures; determine which one of them is the right one? To determine divide the observations into two clusters: process recommendations and product (or structural) recommendations.

21. Give some few Process recommendations.

- The architecture should be the product of a single architect or a small group of architects with an identified leader.
- The architecture should be circulated to the system's stakeholders, who should be actively involved in its review.

22. Give some few Product recommendations.

- The architecture should feature well-defined modules whose functional responsibilities are allocated on the principles of information hiding and separation of concerns.
- Modules that produce data should be separate from modules that consume data. This tends to increase modifiability.

23. What is an architectural pattern?

An architectural pattern is a description of element and relation types together with a set of constraints on how they may be used.

For ex: client-server is a common architectural pattern. Client and server are two element types, and their coordination is described in terms of the protocol that the server uses to communicate with each of its clients.

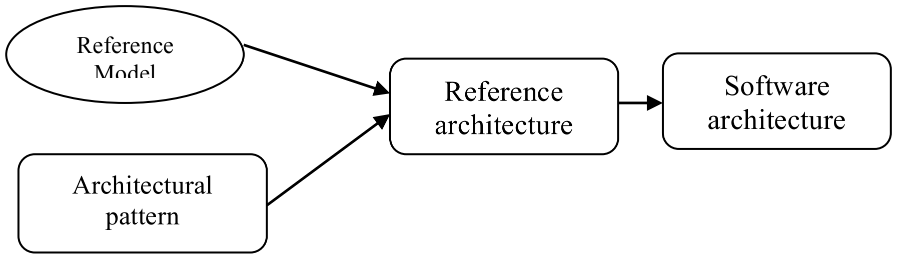
24. What is a reference model?

A reference model is a division of functionality together with data flow between the pieces. A reference model is a standard decomposition of a known problem into parts that cooperatively solve the problem.

25. What is reference architecture?

A reference architecture is 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.

26. Give the relationship between Reference models, architectural patterns, reference architectures and software architectures with neat sketch.



Reference models, architectural patterns, and reference architectures are not architectures; they are useful concepts that capture elements of an architecture. The arrows indicate that subsequent concepts contain more design elements.

27. What are the Kruchten's four views to choose the structures?

- Logical
- Development
- Process
- Physical

28. Define Functionality requirements.

Functional requirements specify what the software needs to do. They relate to the actions that the product must carry out in order to satisfy the fundamental reasons for its existence.

29. What are the different levels of functionality requirements?

- **Business Level:** defines the objective/goal of the project and the measurable business benefits for doing it.
- **User Level:** user requirements are written from the user's point-of-view.
- **System Level:** defines what the system must do to process input and provide the desired output.

30. Define Quality attributes.

Quality attributes are the overall factors that affect run-time behavior, system design, and user experience. Some are related to the overall system design, while others are specific to run time, design time, or user centric issues. Some of the attributes are such as usability, performance, reliability, and security indicates the success of the design and the overall quality of the software application.

31. Is functionality and quality attributes are orthogonal?

Functionality and quality attributes are orthogonal. If functionality and quality attributes were not orthogonal, the choice of function would dictate the level of security or performance or availability or usability

32. What are the three classes of Quality attributes?

a. Qualities of the system.
b. Business qualities that are affected by the architecture.
c. Qualities, such as conceptual integrity, that is about the architecture itself although they indirectly affect other qualities, such as modifiability.

PART- B

1. <u>Define the Software Architecture. Discuss in detail the implications of the definition.</u>

WHAT IS SOFTWARE ARCHITECTURE?

- The software architecture of a program or computing system is a representation of the system that aids in the understanding of how the system will behave.
- Software architecture serves as the blueprint for both the system and the project developing it, defining the work assignments that must be carried out by design and implementation teams.
- The architecture is the primary carrier of system qualities such as performance, modifiability, and security, none of which can be achieved without a unifying architectural vision.
- Architecture is an artifact for early analysis to make sure that a design approach will yield an acceptable system.
- By building effective architecture, the design risks can be identified and can be mitigate early in the development process.
- If a project has not achieved system architecture, including its rationale, the project should not proceed to full-scale system development.
- Specifying the architecture as a deliverable enables its use throughout the development and maintenance process.-Barry Boehm.

STANDARD DEFINITION AND THE IMPLICATION OF THE 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.
- There the primary building blocks were called "components," a term that has since become closely associated with the component-based software engineering movement, taking on a decidedly runtime flavor. "Element" was chosen here to convey something more general.

SOFTWARE ARCHITECTURE DEFINITIONS
Architectural Design: The process of defining a collection of hardware and software components and their interfaces to establish the framework for the development of a computer system. ***IEEE Glossary***
The Software Architecture of a program or a 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. ***Software Architecture in Practice, 2nd Edition, Len Bass, Paul Clements, Rick Kazman***
The architecture of a system defines the system in terms of computational components and interaction between those components. Components are such things as clients and servers, databases, filters and layers in a hierarchal system.
Software Architecture is the "structure of the components of a program/system, their interrelationships, and principles and guidelines governing their design and evolution over time". ***Garlan and Perry, guest editorial to the IEEE transactions on Software Engineering, April 1995***

- **First**, architecture defines software elements. The architecture embodies information about how the elements relate to each other.
- **Second**, the definition makes clear that systems can and do comprise more than one structure and that no one structure can irrefutably claim to be the architecture. For example, all nontrivial projects are partitioned into implementation units; these units are given specific responsibilities and are frequently the basis of work assignments for programming teams.
- **Third**, the definition implies that every computing system with software has software architecture because every system can be shown to comprise elements and the relations among them.
- **Fourth**, the behavior of each element is part of the architecture insofar as that behavior can be observed or discerned from the point of view of another element. Such behavior is what allows elements to interact with each other, which is clearly part of the architecture.
- This is another reason that the box-and-line drawings that are passed off as architectures are not architectures at all.
- Finally, the definition is indifferent as to whether the architecture for a system is a good one or a bad one, meaning that it will allow or prevent the system from meeting its behavioral, performance, and life-cycle requirements.

2. **Explain the working of Architecture Business Cycle with the help of block diagram.**
   THE ARCHITECTURE BUSINESS CYCLE

- 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.

- Software architecture is a result of technical, business and social influences.
- Its existence in turn affects the technical, business and social environments that subsequently influence future architectures.
- This cycle of influences, from environment to the architecture and back to the environment, is called the Architecture Business Cycle (ABC).

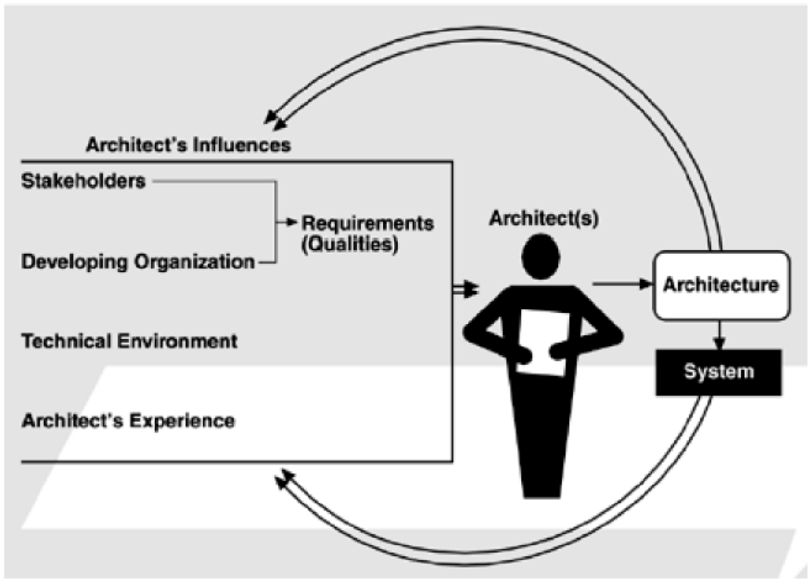
WHERE DO ARCHITECTURES COME FROM?

- Architecture is the result of a set of business and technical decisions.
- There are many influences at work in its design, and the realization of these influences will change depending on the environment in which the architecture is required to perform.
- Even with the same requirements, hardware, support software, and human resources available, an architect designing a system today is likely to design a different system than might have been designed five years ago.
    - Architectures are influenced by system stakeholders
    - Architectures are influenced by developing organizations
    - Architectures are influenced by the background and experience of the architects
    - Architectures are influenced by the technical environment

THE ARCHITECTURE AFFECTS THE FACTORS THAT INFLUENCE THEM

- Relationships among business goals, product requirements, architects experience, architectures and fielded systems form a cycle with feedback loops that a business can manage.
- A business manages this cycle to handle growth, to expand its enterprise area, and to take advantage of previous investments in architecture and system building.
- Figure 1 shows ABC cycle with the feedback loops. Some of the feedback comes from the architecture itself, and some comes from the system built from it.

Figure 1. The Architecture Business Cycle



WORKING OF ARCHITECTURE BUSINESS CYCLE:

1. The architecture affects the structure of the developing organization. An architecture prescribes a structure for a system it particularly prescribes the units of software that must be implemented and integrated to form the system. Teams are formed for individual software units; and the development, test, and integration activities around the units. Likewise, schedules and budgets allocate resources in chunks corresponding to the units.

Teams become embedded in the organization's structure. This is feedback from the architecture to the developing organization.

2. The architecture can affect the goals of the developing organization. A successful system built from it can enable a company to establish a foothold in a particular market area. The architecture can provide opportunities for the efficient production and deployment of the similar systems, and the organization may adjust its goals to take advantage of its newfound expertise to plumb the market. This is feedback from the system to the developing organization and the systems it builds.
3. The architecture can affect customer requirements for the next system by giving the customer the opportunity to receive a system in a more reliable, timely and economical manner than if the subsequent system were to be built from scratch.
4. The process of system building will affect the architect's experience with subsequent systems by adding to the corporate experience base.
5. A few systems will influence and actually change the software engineering culture. i.e, the technical environment in which system builders operate and learn.

3. <u>Explain in detail the different activities which are involved in creating Software Architecture.</u>

SOFTWARE PROCESSES AND THE ARCHITECTURE BUSINESS CYCLE

Software process is the term given to the organization, reutilization, and management of software development activities. The various activities involved in creating software architecture are:

Creating the business case for the system

- It is an important step in creating and constraining any future requirements.
- How much should the product cost?
- What is its targeted market?
- What is its targeted time to market?
- Will it need to interface with other systems?
- Are there system limitations that it must work within?
- These are all the questions that must involve the system's architects.
- They cannot be decided solely by an architect, but if an architect is not consulted in the creation of the business case, it may be impossible to achieve the business goals.

Understanding the requirements

- There are a variety of techniques for eliciting requirements from the stakeholders.
- For ex:
  - Object oriented analysis uses scenarios, or "use cases" to embody requirements.
  - Safety-critical systems use more rigorous approaches, such as finite-state-machine models or formal specification languages.
- Another technique that helps us understand requirements is the creation of prototypes.
- Regardless of the technique used to elicit the requirements, the desired qualities of the system to be constructed determine the shape of its structure.

Creating or selecting the architecture

- In the landmark book *The Mythical Man-Month*, Fred Brooks argues forcefully and eloquently that conceptual integrity is the key to sound system design and that conceptual integrity can only be had by a small number of minds coming together to design the system's architecture.

Documenting and communicating the architecture

- For the architecture to be effective as the backbone of the project's design, it must be communicated clearly and unambiguously to all of the stakeholders.
- Developers must understand the work assignments it requires of them, testers must understand the task structure it imposes on them, management must understand the scheduling implications it suggests, and so forth.

Analyzing or evaluating the architecture

- Choosing among multiple competing designs in a rational way is one of the architect's greatest challenges.
- Evaluating an architecture for the qualities that it supports is essential to ensuring that the system constructed from that architecture satisfies its stakeholders needs.
- Use scenario-based techniques or architecture tradeoff analysis method (ATAM) or cost benefit analysis method (CBAM).

Implementing the system based on the architecture

- This activity is concerned with keeping the developers faithful to the structures and interaction protocols constrained by the architecture.
- Having an explicit and well-communicated architecture is the first step toward ensuring architectural conformance.

Ensuring that the implementation conforms to the architecture

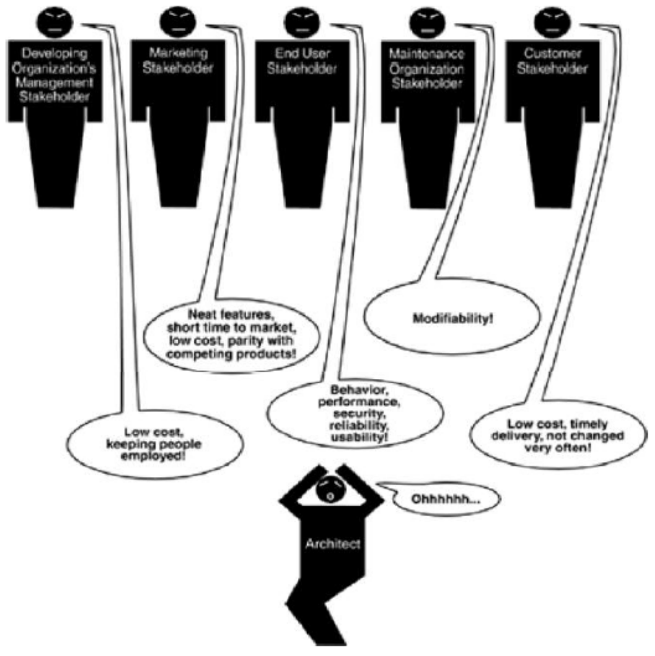
- Finally, when an architecture is created and used, it goes into a maintenance phase.
- Constant vigilance is required to ensure that the actual architecture and its representation remain to each other during this phase.

4. <u>Explain all the factors which affect the influence on Software Architecture.</u>

- Architecture is the result of a set of business and technical decisions.
- There are many influences at work in its design, and the realization of these influences will change depending on the environment in which the architecture is required to perform.
- Even with the same requirements, hardware, support software, and human resources available, an architect designing a system today is likely to design a different system than might have been designed five years ago.
  - Architectures are influenced by system stakeholders
  - Architectures are influenced by developing organizations
  - Architectures are influenced by the background and experience of the architects
  - Architectures are influenced by the technical environment

ARCHITECTURES ARE INFLUENCED BY SYSTEM STAKEHOLDERS

- Many people and organizations interested in the construction of a software system are referred to as stakeholders. E.g. customers, end users, developers, project manager etc.
- Figure below shows the architect receiving helpful stakeholder "suggestions".


Developing Organization's Management Stakeholder
Marketing Stakeholder
End User Stakeholder
Maintenance Organization Stakeholder
Customer Stakeholder
Neat features, short time to market, low cost, parity with competing products!
Modifiability!
Behavior, performance, security, reliability, usability!
Low cost, keeping people employed!
Low cost, timely delivery, not changed very often!
Ohhhhhh...
Architect

- Having an acceptable system involves properties such as performance, reliability, availability, platform compatibility, memory utilization, network usage, security, modifiability, usability, and interoperability with other systems as well as behavior.
- The underlying problem, of course, is that each stakeholder has different concerns and goals, some of which may be contradictory.
- The reality is that the architect often has to fill in the blanks and mediate the conflicts.

ARCHITECTURES ARE INFLUENCED BY THE DEVELOPING ORGANIZATIONS.

- Architecture is influenced by the structure or nature of the development organization.
- There are three classes of influence that come from the developing organizations: immediate business, long-term business and organizational structure.
    - An organization may have an immediate business investment in certain assets, such as existing architectures and the products based on them.
    - An organization may wish to make a long-term business investment in an infrastructure to pursue strategic goals and may review the proposed system as one means of financing and extending that infrastructure.
    - The organizational structure can shape the software architecture.

ARCHITECTURES ARE INFLUENCED BY THE BACKGROUND AND EXPERIENCE OF THE ARCHITECTS.

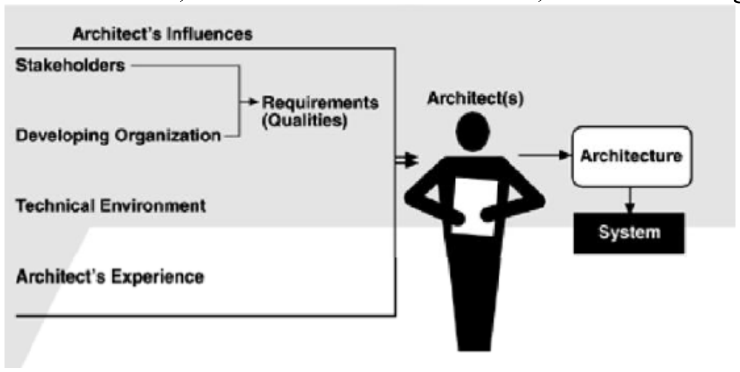
- If the architects for a system have had good results using a particular architectural approach, such as distributed objects or implicit invocation, chances are that they will try that same approach on a new development effort.
- Conversely, if their prior experience with this approach was disastrous, the architects may be reluctant to try it again.
- Architectural choices may also come from an architect's education and training, exposure to successful architectural patterns, or exposure to systems that have worked particularly poorly or particularly well.
- The architects may also wish to experiment with an architectural pattern or technique learned from a book or a course.

ARCHITECTURES ARE INFLUENCED BY THE TECHNICAL ENVIRONMENT

- A special case of the architect's background and experience is reflected by the *technical environment*.
- The environment that is current when an architecture is designed will influence that architecture.
- It might include standard industry practices or software engineering prevalent in the architect's professional community.

RAMIFICATIONS OF INFLUENCES ON AN ARCHITECTURE

- The influences on the architect, and hence on the architecture, are shown in Figure below



- Influences on an architecture come from a wide variety of sources. Some are only implied, while others are explicitly in conflict.
- Architects need to know and understand the nature, source, and priority of constraints on the project as early as possible.
- Therefore, they must identify and actively engage the stakeholders to solicit their needs and expectations.
- Architects are influenced by the requirements for the product as derived from its stakeholders, the structure and goals of the developing organization, the available technical environment, and their own background and experience.

5. Explain briefly the properties of a good Software Architecture design.

Given the same technical requirements for a system, two different architects in different organizations will produce different architectures, how can we determine if either one of them is the right one?

We divide our observations into two clusters: process recommendations and product (or structural) recommendations.

Process recommendations are as follows:

- The architecture should be the product of a single architect or a small group of architects with an identified leader.
- The architect (or architecture team) should have the functional requirements for the system and an articulated, prioritized list of quality attributes that the architecture is expected to satisfy.
- The architecture should be well documented, with at least one static view and one dynamic view, using an agreed-on notation that all stakeholders can understand with a minimum of effort.
- The architecture should be circulated to the system's stakeholders, who should be actively involved in its review.
- The architecture should be analyzed for applicable quantitative measures (such as maximum throughput) and formally evaluated for quality attributes before it is too late to make changes to it.
- The architecture should lend itself to incremental implementation via the creation of a "skeletal" system in which the communication paths are exercised but which at first has minimal functionality. This skeletal system can then be used to "grow" the system incrementally, easing the integration and testing efforts.
- The architecture should result in a specific (and small) set of resource contention areas, the resolution of which is clearly specified, circulated and maintained.

Product (structural) recommendations are as follows:

- The architecture should feature well-defined modules whose functional responsibilities are allocated on the principles of information hiding and separation of concerns.
- Each module should have a well-defined interface that encapsulates or "hides" changeable aspects from other software that uses its facilities. These interfaces should allow their respective development teams to work largely independent of each other.
- Quality attributes should be achieved using well-known architectural tactics specific to each attribute.
- The architecture should never depend on a particular version of a commercial product or tool.
- Modules that produce data should be separate from modules that consume data. This tends to increase modifiability.
- For parallel processing systems, the architecture should feature well-defined processors or tasks that do not necessarily mirror the module decomposition structure.

- Every task or process should be written so that its assignment to a specific processor can be easily changed, perhaps even at runtime.
- The architecture should feature a small number of simple interaction patterns

6. i. <u>Explain the following terms</u>

a. Architectural Model **c. Reference Architecture**

b. Reference Model

a. ARCHITECTURAL MODEL

- An architectural pattern is a description of element and relation types together with a set of constraints on how they may be used.
- A pattern can be thought of as a set of constraints on an architecture-on the element types and their patterns of interaction-and these constraints define a set or family of architectures that satisfy them.
- For example, client-server is a common architectural pattern.
- Client and server are two element types, and their coordination is described in terms of the protocol that the server uses to communicate with each of its clients.
- Use of the term client-server implies only that multiple clients exist; the clients themselves are not identified, and there is no discussion of what functionality, other than implementation of the protocols, has been assigned to any of the clients or to the server.
- Countless architecture is of the client-server pattern under this (informal) definition, but they are different from each other.
- An architectural pattern is not an architecture, then, but it still conveys a useful image of the system-it imposes useful constraints on the architecture and, in turn, on the system.
- One of the most useful aspects of patterns is that they exhibit known quality attributes.
- This is why the architect chooses a particular pattern and not one at random.
- Some patterns represent known solutions to performance problems, others lend themselves well to high-security systems, still others have been used successfully in high-availability systems.
- Choosing an architectural pattern is often the architect's first major design choice.
- The term architectural style has also been widely used to describe the same concept.

b. REFERENCE MODEL

- A division of functionality together with data flow between the pieces.
- A reference model is a standard decomposition of a known problem into parts that cooperatively solve the problem.
- Reference models are a characteristic of mature domains, e.g. Compiler, DBMS

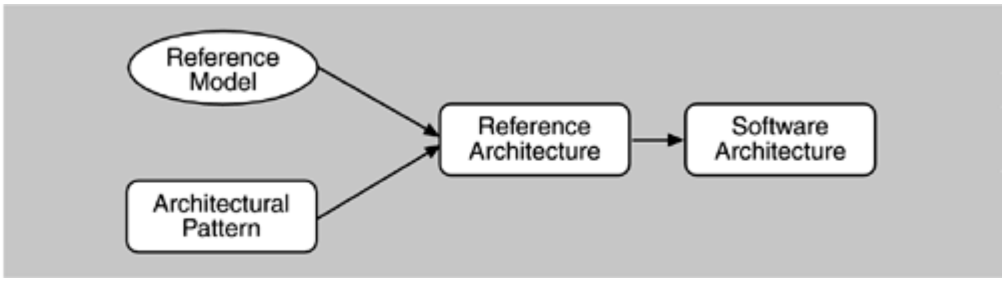
c. REFERENCE ARCHITECTURE

- 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.
- The mapping may be, but by no means necessarily is, one to one. A software element may implement part of a function or several functions.

RELATIONSHIPS

Reference model + architectural pattern => reference architecture => software architecture

- 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.
- The relationship among these design elements is shown in Figure below.



The relationships of reference models, architectural patterns, reference architectures, and software architectures. (Arrows indicates subsequent concepts contain more design elements.)

- People often make analogies to other uses of the word architecture, about which they have some view.
- They commonly associate architecture with physical structure (buildings, streets, hardware) and physical arrangement.
- A building architect must design a building that provides accessibility, aesthetics, light, maintainability, and so on.
- A software architect must design a system that provides concurrency, portability, modifiability, usability, security, and the like, and that reflects consideration of the tradeoffs among these needs.
- Analogies between buildings and software systems should not be taken too far, as they break down fairly quickly.
- Rather, they help us understand that the viewer's perspective is important and that structure can have different meanings depending on the motivation for examining it.
- A precise definition of software architecture is not nearly as important as what investigating the concept allows us to do.

ii. Explain why Software Architecture is important.

Communication among stakeholders

- Software architecture represents a common abstraction of a system that most if not all of the system's stakeholders can use as a basis for mutual understanding, negotiation, consensus, and communication.

Early design decisions

- Software architecture manifests the earliest design decisions about a system, and these early bindings carry weight far out of proportion to their individual gravity with respect to the system's remaining development, its deployment, and its maintenance life.
- It is also the earliest point at which design decisions governing the system to be built can be analyzed.
  - The architecture defines constraints on implementation
  - The architecture inhibits or enables a system's quality attributes
  - Predicting system qualities by studying the architecture
  - The architecture makes it easier to reason about and manage change
  - The architecture helps in evolutionary prototyping
  - The architecture enables more accurate cost and schedule estimates

Transferable abstraction of a system

- Software architecture constitutes a relatively small, intellectually graspable model for how a system is structured and how its elements work together, and this model is transferable across systems.
- In particular, it can be applied to other systems exhibiting similar quality attribute and functional requirements and can promote large-scale re-use.
  - Software product lines share a common architecture
  - Systems can be built using large, externally developed elements

Less is more: It pays to restrict the vocabulary of design alternatives

An architecture permits template-based development
An architecture can be the basis for training

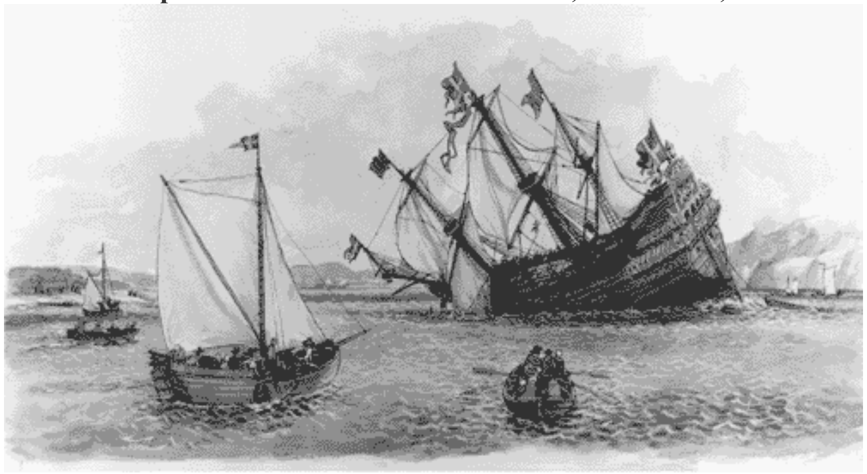
7. **<u>Explain Hybertsson's three views for Software Architecture with an example.</u>**

- For decades, software designers have been taught to build systems based exclusively on the technical requirements.
- Conceptually, the requirements document is tossed over the wall into the designer's cubicle, and the designer must come forth with a satisfactory design.
- Requirements produce design, which produce system.
- Of course, modern software development methods recognize the naïveté of this model and provide all sorts of feedback loops from designer to analyst.
- But they still make the implicit assumption that design is a product of the system's technical requirements, period.
- Software architecture encompasses the structures of large software systems.
- The architectural view of a system is abstract, distilling away details of implementation, algorithm, and data representation and concentrating on the behavior and interaction of "black box" elements.
- A software architecture is developed as the first step toward designing a system that has a collection of desired properties.
- 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.
- Architecture serves as an important communication, reasoning, analysis, and growth tool for systems.

The Swedish Ship *Vasa*

In the 1620s, Sweden and Poland were at war. The king of Sweden, Gustavus Adolphus, was determined to put a fast end to it and commissioned a new warship the likes of which had never been seen before. The *Vasa*, shown in Figure below, was to be the world's most formidable instrument of war: 70 meters long, able to carry 300 soldiers, and with an astonishing 64 heavy guns mounted on two gun decks. Seeking to add overwhelming firepower to his navy to strike a important blow, the king insisted on stretching the *Vasa*'s weapons to the limits. Her architect, Henrik Hybertsson, was a seasoned Dutch shipbuilder with an perfect character, but the *Vasa* was beyond even his broad experience. Two-gun-deck ships were rare, and none had been built of the *Vasa*'s size and guardian.

The warship *Vasa.*
Used with permission of The Vasa Museum, Stockholm, Sweden.



Like all architects of systems that push the envelope of experience, Hybertsson had to balance many concerns. Swift time to deployment was critical, but so were performance, functionality, safety, reliability, and cost. He was also responsible to a variety of stakeholders. In this case, the primary customer was the king, but Hybertsson also was responsible to the crew that would sail his creation. Also like all architects, Hybertsson brought his experience with him to the task. In this case, his experience told him to design the *Vasa* as though it were a single-gun-deck ship and then extrapolate, which was in accordance with the technical environment of the day. Faced with an impossible task, Hybertsson had the good sense to die about a year before the ship was finished.

The project was completed to his specifications, however, and on Sunday morning, August 10, 1628, the mighty ship was ready. She set her sails, waddled out into Stockholm's deep-water harbor, fired her guns in salute, and promptly rolled over. Water poured in through the open gun ports, and the *Vasa* dropped. A few minutes later her first and only voyage ended 30 meters under the surface. Dozens among her 150-man crew drowned.

Inquiries followed, which concluded that the ship was well built but "badly proportioned." In other words, its architecture was flawed. Today we know that Hybertsson did a poor job of balancing all of the conflicting constraints levied(charged) on him. In particular, he did a poor job of risk management and a poor job of customer management (not that anyone could have fared better). He simply submit in the face of impossible requirements.

The story of the *Vasa*, although more than 375 years old, well illustrates the Architecture Business Cycle: organization goals beget requirements, which beget an architecture, which begets a system. The architecture flows from the architect's experience and the technical environment of the day. Hybertsson suffered from the fact that neither of those were up to the task before him.

Three things that Hybertsson could have used:

1. Case studies of successful architectures crafted to satisfy demanding requirements, so as to help set the technical playing field of the day.
2. Methods to assess an architecture before any system is built from it, so as to mitigate the risks associated with launching unprecedented designs.
3. Techniques for incremental architecture-based development, so as to uncover design flaws before it is too late to correct them.

Our goal is to give architects another way out of their design dilemmas than the one that befell the ill-fated Dutch ship designer. Death before deployment is not nearly so admired these days.

8. <u>What Software Architecture Is and What It Isn't? Discuss with an example.</u>

WHAT SOFTWARE ARCHITECTURE IS AND WHAT IT ISN'T?

The software architecture of a system is the set of structures needed to reason about the system, which comprise software elements, relations among them, and properties of both. System architecture is concerned with a total system, including hardware, software, and humans.

- Architecture is a set of software structures
  - (program) decomposition structures, component-and-connector structures (C&C), allocation structures
- Architecture is an abstraction
  - Model of a software system
- Every software system has a software architecture
  - documented or undocumented
- Architecture includes behaviour
  - (external) behaviours/properties of architectural elements are part of architecture
- Not all architectures are good architectures
  - determined by architecture's quality attributes

Software architecture must be distinguished from lower-level design (e.g., design of component internals and algorithms) and implementation, on the one hand, and other kinds of related architectures, on the other. Software architecture is not the information (or data) model, though it uses the information model to get type information for method signatures on interfaces, for example. It is also not the architecture of the physical system, including processors, networks, and the like, on which the software will run. However, it uses this information in evaluating the impact of architectural choices on system qualities such as performance and reliability. More obviously, perhaps, it is also not the hardware architecture of a product to be manufactured. While each of these other architectures typically have their own specialists leading their design, these architectures impact and are impacted by the software architecture, and where possible, should not be designed in isolation from one another. This is the domain of *system* architecting.

UNDERSTANDING SOFTWARE ARCHITECTURE

An Example taken from a system description for an underwater acoustic simulation purports to describe that system's "top-level architecture" and is precisely the kind of diagram most often displayed to help explain the architecture.

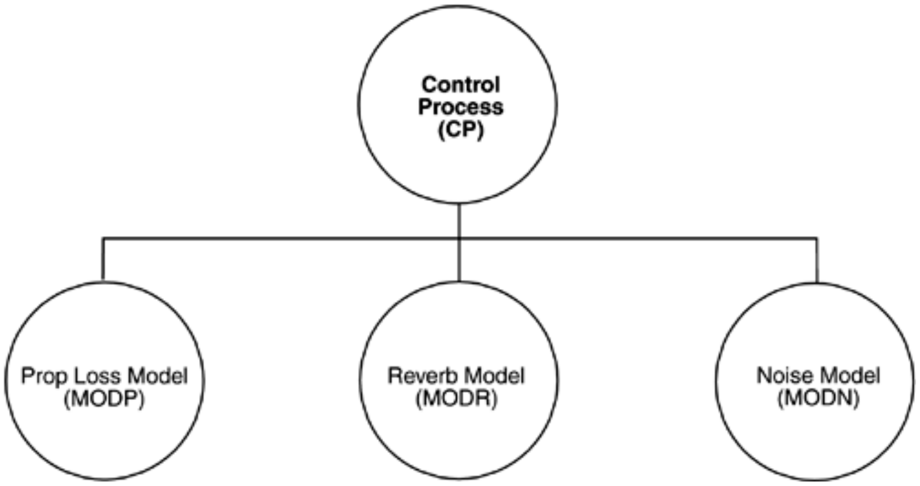


Figure.1 Typical but uninformative, presentation of a software architecture

WHAT **IS OBSERVED FROM THIS DIAGRAM?**

- The system consists of four elements.
- Three of the four elements may have something in common because they are positioned next to each other.
- All elements have some sort of relationship with each other since they are fully Connected
- Does the diagram show software architecture?

WHAT CANNOT BE DETERMINED FROM THIS DIAGRAM?

Nature of the elements

- What is the significance of their separation?
- Are they separate processes?
- Do they run on separate processors?
- Do they run at separate times?
- Are they objects, tasks, modules? Classes? Functions? What are they?

Responsibilities of the elements

- What are the functionalities of the elements?

Significance of the connections

- Do the elements communicate with each other?
- Do they control each other?
- Do they send data and use each other?
- Do they invoke each other and share some information hiding secrets?

Significance of the layout

- Why is CP on a separate level?
- Does it call the other three elements?
- Are the others allowed to call it?

Answers of all of the above questions are vital for the implementation team hence need to be discussed, defined and documented. Complete definition of **Architectural structures and views** are important.

9. <u>Explain about the Architectural Structures and Views.</u>

Architectural structures can by and large be divided into three groups, depending on the broad nature of the elements they show.

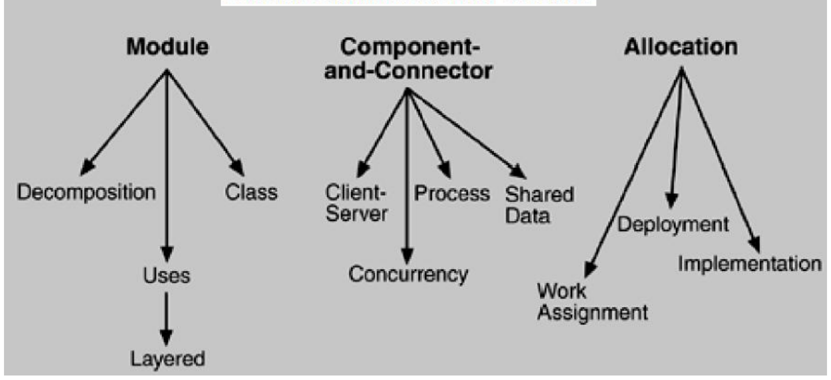
MODULE STRUCTURES.

- Here the elements are modules, which are units of implementation.
- Modules represent a code-based way of considering the system.
- They are assigned areas of functional responsibility.
- There is less emphasis on how the resulting software manifests itself at runtime.
- Module structures allow us to answer questions such as
  - 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?
  - What modules are related to other modules by generalization or specialization (i.e., inheritance) relationships?

COMPONENT-AND-CONNECTOR STRUCTURES.

- Here the elements are runtime components (which are the principal units of computation) and connectors (which are the communication vehicles among components).
- Component-and-connector structures help answer questions such as
  - ▪ 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?

Common software architecture structures



ALLOCATION STRUCTURES.

- Allocation structures show the relationship between the software elements and the elements in one or more external environments in which the software is created and executed.
- They answer questions such as
  - o What processor does each software element execute on?
  - o In what files is each element stored during development, testing, and system building?
  - o What is the assignment of software elements to development teams?

SOFTWARE STRUCTURES

Module

Module-based structures include the following structures.

Decomposition: The units are modules related to each other by the "is a submodule of " relation, showing how larger modules are decomposed into smaller ones recursively until they are small enough to be easily understood.
Uses: The units are related by the uses relation. One unit uses another if the correctness of the first requires the presence of a correct version (as opposed to a stub) of the second.
Layered: Layers are often designed as abstractions (virtual machines) that hide implementation specifics below from the layers above, engendering portability.
Class or generalization: The class structure allows us to reason about re-use and the incremental addition of functionality.

Component-and-connector

Component-and-connector structures include the following structures
Process or communicating processes: The units here are processes or threads that are connected with each other by communication, synchronization, and/or exclusion operations.
Concurrency: The concurrency structure is used early in design to identify the requirements for managing the issues associated with concurrent execution.
Shared data or repository: This structure comprises components and connectors that create, store, and access persistent data
Client-server: This is useful for separation of concerns (supporting modifiability), for physical distribution, and for load balancing (supporting runtime performance).

Allocation

Allocation structures include the following structures
Deployment: This view allows an engineer to reason about performance, data integrity, availability, and security
Implementation: This is critical for the management of development activities and builds processes.
Work assignment: This structure assigns responsibility for implementing and integrating the modules to the appropriate development teams.

RELATING STRUCTURES TO EACH OTHER

- Each of these structures provides a different perspective and design handle on a system, and each is valid and useful in its own right.
- In general, mappings between structures are many to many.
- Individual structures bring with them the power to manipulate one or more quality attributes.
- They represent a powerful separation-of-concerns approach for creating the architecture.

WHICH STRUCTURES TO CHOOSE?

Kruchten's four views follow:

Logical. The elements are "key abstractions," which are manifested in the object-oriented world as objects or object classes. This is a module view.
Process. This view addresses concurrency and distribution of functionality. It is a component-and-connector view.
Development. This view shows the organization of software modules, libraries, subsystems, and units of development. It is an allocation view, mapping software to the development environment.
Physical. This view maps other elements onto processing and communication nodes and is also an allocation view

10. **<u>Explain the following</u>**

a. Functional Requirements **b. Quality Attributes**

<u>a. FUNCTIONAL REQUIREMENTS</u>

WHAT DRIVES SOFTWARE ARCHITECTURE?

System requirements:

- **Functional needs** (what the system should do)
- **Quality needs** (properties that the system must possess such as availability, performance, security,

FUNCTIONAL REQUIREMENTS

Functional requirements specify what the software needs to do. They relate to the **actions** that the product must carry out in order to satisfy the fundamental reasons for its existence.

- **Business Level**: defines the objective/goal of the project and the measurable business benefits for doing it.
- **User Level**: user requirements are written from the user's point-of-view.
- **System Level**: defines what the system must do to process input and provide the desired output.

MOSCOW METHOD:

- **M - MUST**: Describes a requirement that must be satisfied in the final solution for the solution to be considered a success.
- **S - SHOULD**: Represents a high-priority item that should be included in the solution if it is possible. This is often a critical requirement but one which can be satisfied in other ways if strictly necessary.
- **C - COULD**: Describes a requirement which is considered desirable but not necessary. This will be included if time and resources permit.
- **W - WON'T**: Represents a requirement that stakeholders have agreed will not be implemented in a given release, but may be considered for the future.

FUNCTIONALITY AND SOFTWARE ARCHITECTURE

- It is the ability of the system or application to **satisfy the purpose** for which it was designed.
- It drives the initial **decomposition** of the system.
- It is the **basis** upon which all other quality attributes are specified.
- It is **related to quality attributes** like validity, correctness, interoperability, and security.
- Functional requirements often get the most focus in a development project. But systems are often redesigned, **not** because of functional requirements.

b.QUALITY ATTRIBUTES

- Quality is a measure of excellence or the state of being free from deficiencies or defects.
- Quality attributes are the system properties that are separate from the functionality of the system.
- Implementing quality attributes makes it easier to differentiate a good system from a bad one.

Attributes are overall factors that affect runtime behavior, system design, and user experience.**They can be classified as**

Static Quality Attributes

- Reflect the structure of a system and organization, directly related to architecture, design, and source code.
- They are invisible to end-user, but affect the development and maintenance cost, e.g.: modularity, testability, maintainability, etc.

Dynamic Quality Attributes

- Reflect the behavior of the system during its execution.
- They are directly related to system's architecture, design, source code, configuration, deployment parameters, environment, and platform.
- They are visible to the end-user and exist at runtime, e.g. throughput, robustness, scalability, etc.

COMMON QUALITY ATTRIBUTES

The following table lists the common quality attributes software architecture must have

Category	Quality Attribute	Description
Design Qualities	Conceptual Integrity	Defines the consistency and coherence of the overall design.
	Maintainability	Ability of the system to undergo changes with a degree of ease.

	Reusability	Defines the capability for components and subsystems to be suitable for use in other applications.
Run-time Qualities	Interoperability	Ability of a system or different systems to operate successfully by communicating and exchanging information with other external systems written and run by external parties.
	Manageability	Defines how easy it is for system administrators to manage the application.
	Reliability	Ability of a system to remain operational over time.
	Scalability	Ability of a system to either handle the load increase without impacting the performance of the system or the ability to be readily enlarged.
	Security	Capability of a system to prevent malicious or accidental actions outside of the designed usages.
	Performance	Indication of the responsiveness of a system to execute any action within a given time interval.
	Availability	Defines the proportion of time that the system is functional and working. It can be measured as a percentage of the total system downtime over a predefined period.
System Qualities	Supportability	Ability of the system to provide information helpful for identifying and resolving issues when it fails to work correctly.
	Testability	Measure of how easy it is to create test criteria for the system and its components.
User Qualities	Usability	Defines how well the application meets the requirements of the user and consumer by being intuitive.
Architecture Quality	Correctness	Accountability for satisfying all the requirements of the system.
Non-runtime Quality	Portability	Ability of the system to run under different computing environment.
	Integrality	Ability to make separately developed components of the system work correctly together.
	Modifiability	Ease with which each software system can accommodate changes to its software.
Business quality attributes	Cost and schedule	Cost of the system with respect to time to market, expected project lifetime & utilization of legacy.
	Marketability	Use of system with respect to market competition.

UNIT – II QUALITY ATTRIBUTE WORKSHOP

PART- A

1. **Define Quality Attribute Workshop (QAW)?**
   The QAW is a facilitated method that engages a diverse group of system stakeholders early in the life cycle to discover the driving quality attributes of a software-intensive system.
   The QAW provides a way to identify important quality attributes and the scenarios associated with them and to clarify system requirements before the software architecture has been created.
2. **What are the benefits of QAW?**
   - QAW provides a forum for a wide variety of stakeholders to gather in one room at one time very early in the development process.

- It is often the first time such a meeting takes place and generally leads to the identification of conflicting assumptions about system requirements.

3. **What are the steps involved in QAW?**

i. QAW Presentation and Introductions	v. Scenario Brainstorming
ii. Business/Mission Presentation	vi. Scenario Consolidation
iii. Architectural Plan Presentation	vii. Scenario Prioritization
iv. Identification of Architectural Drivers	viii. Scenario Refinement

4. **What are the problems with System Quality attributes?**
   - The definitions provided for an attribute are not operational. It is meaningless to say that a system will be modifiable. Every system is modifiable with respect to one set of changes and not modifiable with respect to another. The other attributes are similar.
   - A focus of discussion is often on which quality a particular aspect belongs to. Is a system failure an aspect of availability, an aspect of security, or an aspect of usability? All three attribute communities would claim ownership of a system failure
5. **What is meant by quality attribute scenario?**
   A quality attribute scenario is a quality-attribute-specific requirement. It capture non-functional requirements of a system in terms of standard quality attributes, such as Availability, Modifiability, and so on.
6. **Name the different quality attributes?**

• Availability	• Security
• Modifiability	• Testability
• Performance	• Usability

7. **Define Availability.**
   Availability is concerned with system failure and duration of system failures. System failure means when the system does not provide the service for which it was intended.
8. **Define Failure.**
   Failures are usually a result of system errors that are derived from faults in the system. It is typically defines as

$$\alpha = \frac{\text{mean time to failure}}{\text{mean time to failure} + \text{mean time to repair}}$$

9. **Define Modifiability.**
   Modifiability is about the cost of change, both in time and money. It brings up two concerns.
   - What can change (the artifact)?
   - When is the change made and who makes it (the environment)?
10. **Define Performance**
    Performance is about timeliness. Events occur and the system must respond in a timely fashion.
11. **Define Security.**
    Security is the ability of the system to prevent or resist unauthorized access while providing access to legitimate users. An attack is an attempt to breach security.
12. **Define Testability**
    Testability refers to the ease with which the software can be made to demonstrate its faults or lack thereof. To be testable the system must control inputs and be able to observe outputs.

13. **Define Usability.**
    Usability is how easy it is for the user to accomplish tasks and what support the system provides for the user to accomplish this. Dimensions are learning system features, using the system efficiently and minimizing the impact of errors.
14. **What are the six parts of quality attribute scenario?**

i. Source of stimulus (e.g., human, computer system, etc.)
ii. Stimulus – a condition that needs to be considered
iii. Environment - what are the conditions when the stimulus occurs?
iv. Artifact – what elements of the system are stimulated?
v. Response – the activity undertaken after arrival of the stimulus
vi. Response measure – when the response occurs it should be measurable so that the requirement can be tested.

15. Define Non-repudiation.

Non-repudiation is the property that a transaction (access to or modification of data or services) cannot be denied by any of the parties to it. This means you cannot deny that you ordered that item over the Internet if, in fact, you did.

16. Define Confidentiality.

Confidentiality is the property that data or services are protected from unauthorized access. This means that a hacker cannot access your income tax returns on a government computer.

17. Define Integrity.

Integrity is the property that data or services are being delivered as intended. This means that your grade has not been changed since your instructor assigned it.

18. Define Assurance.

Assurance is the property that the parties to a transaction are who they purport to be. This means that, when a customer sends a credit card number to an Internet merchant, the merchant is who the customer thinks they are.

19. Define Auditing.

Auditing is the property that the system tracks activities within it at levels sufficient to reconstruct them. This means that, if you transfer money out of one account to another account, in Switzerland, the system will maintain a record of that transfer.

20. Name few different stimuli possible for each of the quality attributes?

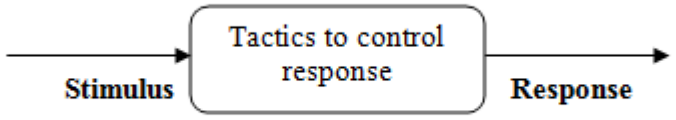
Quality attributes	**Stimulus**
Availability	Unexpected event, nonoccurrence of expected event
Performance	Periodic, stochastic or sporadic
Testability	Completion of phase of system development

21. What are the different business qualities?

- Time to market
- Cost and benefit
- Projected lifetime of the system
- Targeted market
- Rollout schedule
- Integration with legacy systems

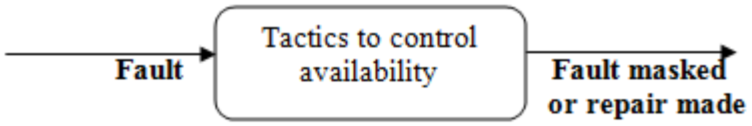
22. Define tactic.

A tactic is a design decision that influences the control of a quality attribute response.



23. Define Availability tactics

All approaches to maintaining availability involve some type of redundancy, some type of health monitoring to detect a failure, and some type of recovery when a failure is detected. In some cases, the monitoring or recovery is automatic and in others it is manual.



24. Define Active Redundancy.

Active redundancy (hot restart). All redundant components respond to events in parallel. The response from only one component is used (usually the first to respond), and the rest are discarded. Active redundancy is often used in a client/server configuration, such as database management systems, where quick responses are necessary even when a fault occurs

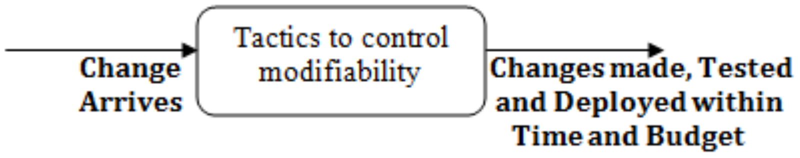
25. Define Passive Redundancy.

Passive redundancy (warm restart/dual redundancy/triple redundancy). One component (the primary) responds to events and informs the other components (the standbys) of state updates they must make. When a fault occurs, the system must first ensure that the backup state is sufficiently fresh before resuming services.

26. Define Checkpoint/rollback.

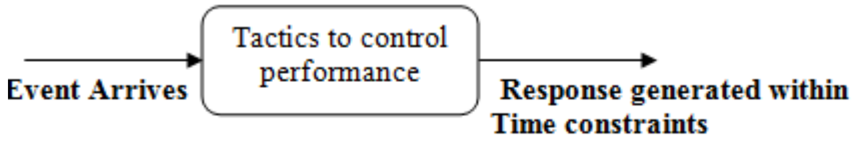
A checkpoint is a recording of a consistent state created either periodically or in response to specific events. Sometimes a system fails in an unusual manner, with a detectably inconsistent state. In this case, the system should be restored using a previous checkpoint of a consistent state and a log of the transactions that occurred since the snapshot was taken.

27. Define Modifiability tactics.


Change Arrives
Tactics to control modifiability
Changes made, Tested and Deployed within Time and Budget

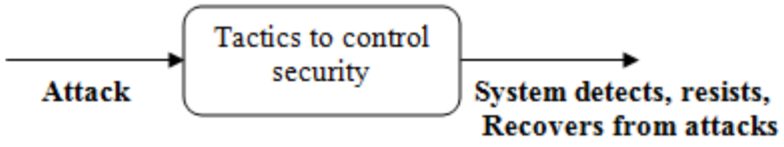
28. Define Performance tactics.

Performance tactics control the time within which a response is generated. Goal of performance tactics is shown below:


Event Arrives
Tactics to control performance
Response generated within Time constraints

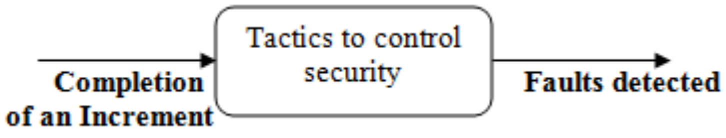
29. Define Security tactics.

Tactics for achieving security can be divided into those concerned with resisting attacks, those concerned with detecting attacks, and those concerned with recovering from attacks.


Attack
Tactics to control security
System detects, resists, Recovers from attacks

30. Define Testability tactics.

The goal of tactics for testability is to allow for easier testing when an increment of software development is completed. The use of tactics for testability is displayed below



PART- B

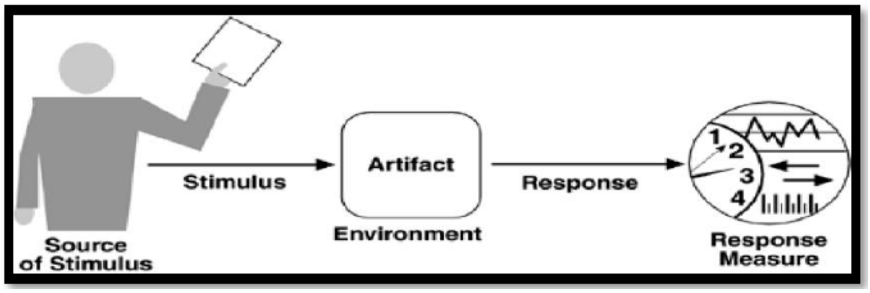
1. <u>Explain the quality attributes scenario in detail.</u>

<u>QUALITY SCENARIOS</u>

- Quality scenarios specify how to prevent a fault from becoming a failure.
- They can be divided into six parts based on their attribute specifications

- **Source** − An internal or external entity such as people, hardware, software, or physical infrastructure that generate the stimulus.
- **Stimulus** − A condition that needs to be considered when it arrives on a system.
- **Environment** − The stimulus occurs within certain conditions.
- **Artifact** − A whole system or some part of it such as processors, communication channels, persistent storage, processes etc.
- **Response** − An activity undertaken after the arrival of stimulus such as detect faults, recover from fault, disable event source etc.
- **Response measure** − Should measure the occurred responses so that the requirements can be tested.

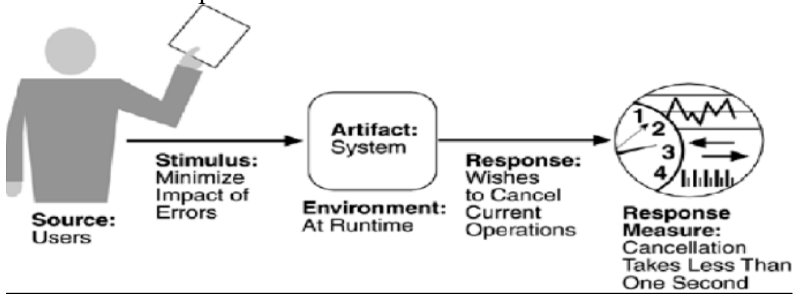
Figure below shows the parts of a quality attribute scenario.



ANY ONE QUALITY ATTRIBUTE SCENARIO IN DETAIL

USABILITY SCENARIO

- Usability is concerned with how easy it is for the user to accomplish a desired task and the kind of user support the system provides. It can be broken down into the following areas:
- ***Learning system features***. If the user is unfamiliar with a particular system or a particular aspect of it, what can the system do to make the task of learning easier?
- ***Using a system efficiently***. What can the system do to make the user more efficient in its operation?
- ***Minimizing the impact of errors.*** What can the system do so that a user error has minimal impact?
- ***Adapting the system to user needs***. How can the user (or the system itself) adapt to make the user's task easier?
- ***Increasing confidence and satisfaction***. What does the system do to give the user confidence that the correct action is being taken?
- A user, wanting to minimize the impact of an error, wishes to cancel a system operation at runtime; cancellation takes place in less than one second.


Source: Users
Stimulus: Minimize Impact of Errors
Artifact: System
Environment: At Runtime
Response: Wishes to Cancel Current Operations
Response Measure: Cancellation Takes Less Than One Second
1
2
3
4

The portions of the usability general scenarios are

- ***Source of stimulus***. The end user is always the source of the stimulus.
- ***Stimulus***. The stimulus is that the end user wishes to use a system efficiently, learn to use the system, minimize the impact of errors, adapt the system, or feel comfortable with the system. In our example, the user wishes to cancel an operation, which is an example of minimizing the impact of errors.

- ***Artifact***. The artifact is always the system.
- ***Environment***. The user actions with which usability is concerned always occur at runtime or at system configuration time. In Figure, the cancellation occurs at runtime.
- ***Response***. The system should either provide the user with the features needed or anticipate the user's needs. In our example, the cancellation occurs as the user wishes and the system is restored to its prior state.
- ***Response measure***. The response is measured by task time, number of errors, number of problems solved, user satisfaction, gain of user knowledge, ratio of successful operations to total operations, or amount of time/data lost when an error occurs. In Figure, the cancellation should occur in less than one second.

2. **Define availability. Explain general scenario for availability.**

AVAILABILITY SCENARIO

- **Availability** is concerned with system failure and its associated consequences Failures are usually a result of system errors that are derived from faults in the system.
- It is typically defines as

$$\alpha = \frac{\text{mean time to failure}}{\text{mean time to failure} + \text{mean time to repair}}$$

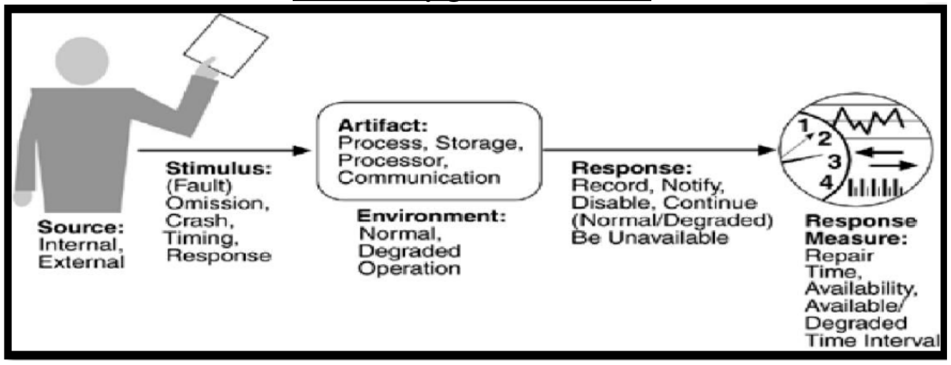
Source of stimulus.

- Differentiate between internal and external indications of faults or failure since the desired system response may be different.
- In the example, the unexpected message arrives from outside the system.

Stimulus. A fault of one of the following classes occurs.

- *Omission*-A component fails to respond to an input.
- *Crash*-The component repeatedly suffers omission faults.
- T*iming*-A component responds but the response is early or late.
- *Response*- A component responds with an incorrect value.

Availability general scenario


Source: Internal, External
Stimulus: (Fault) Omission, Crash, Timing, Response
Artifact: Process, Storage, Processor, Communication
Environment: Normal, Degraded Operation
Response: Record, Notify, Disable, Continue (Normal/Degraded) Be Unavailable
Response Measure: Repair Time, Availability, Available/ Degraded Time Interval

- ***Artifact.*** This specifies the resource that is required to be highly available, such as a processor, communication channel, process, or storage.
- ***Environment***. The state of the system when the fault or failure occurs may also affect the desired system response.
- For example, if the system has already seen some faults and is operating in other than normal mode, it may be desirable to shut it down totally.
- However, if this is the first fault observed, some degradation of response time or function may be preferred. In our example, the system is operating normally.
- ***Response.*** There are a number of possible reactions to a system failure. These include logging the failure, notifying selected users or other systems, switching to a degraded

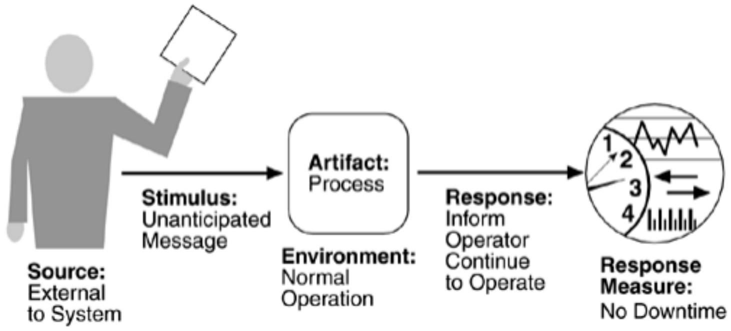
mode with less capacity or less function, shutting down external systems, or becoming unavailable during repair.

- In the example, the system should notify the operator of the unexpected message and continue to operate normally.
- ***Response measure.*** The response measure can specify an availability percentage, or it can specify a time to repair, times during which the system must be available, or the duration for which the system must be available

Scenario Portion	Possible Values
Source	Internal to system or external to system
Stimulus	Crash, omission, timing, no response, incorrect response
Artifact	System's processors, communication channels, persistent storage
Environment	Normal operation; degraded (failsafe) mode
Response	Log the failure, notify users/operators, disable source of failure, continue (normal/degraded)
RespMeasure	Time interval available, availability%, repair time, unavailability time interval

"An unanticipated external message is received by a process during normal operation. The process infor

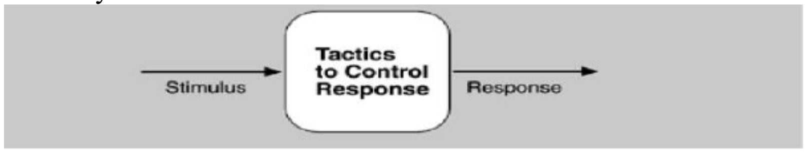
Availability sample scenario



The source of the stimulus is important since differing responses may be required depending on what it is. For example, a request from a trusted source may be treated differently from a request from an untrusted source in a security scenario. The environment may also affect the response, in that an event arriving at a system may be treated differently if the system is already overloaded. The artifact that is stimulated is less important as a requirement. It is almost always the system, and it is explicitly call it out for two reasons. First, many requirements make assumptions about the internals of the system (e.g., "a Web server within the system fails"). Second, when we utilize scenarios within an evaluation or design method, we refine the scenario artifact to be quite explicit about the portion of the system being stimulated. Finally, being explicit about the value of the response is important so that quality attribute requirements are made explicit. Thus, we include the response measure as a portion of the scenario.

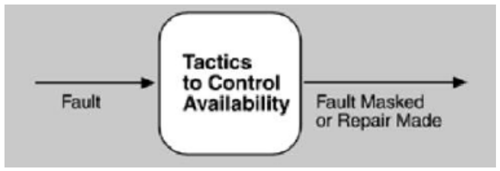
3. **What do you mean by tactics? Explain the availability tactics with neat diagram.**
   - A **tactic** is a design decision that influences the control of a quality attribute response.
   - Patterns package tactics. A pattern that supports availability will likely use both a redundancy tactic and a synchronization tactic.

- Tactics can refine other tactics. For each quality attribute that we discuss, we organize the tactics as a hierarchy.



Tactics are intended to control responses to stimuli.

AVAILABILITY TACTICS



- The above figure depicts goal of availability tactics. All approaches to maintaining availability involve some type of redundancy, some type of health monitoring to detect a failure, and some type of recovery when a failure is detected. In some cases, the monitoring or recovery is automatic and in others it is manual.

FAULT DETECTION

- ***Ping/echo***. One component issues a ping and expects to receive back an echo, within a predefined time, from the component under scrutiny. This can be used within a group of components mutually responsible for one task
- ***Heartbeat (dead man timer).*** In this case one component emits a heartbeat message periodically and nother component listens for it. If the heartbeat fails, the originating component is assumed to have failed and a fault correction component is notified. The heartbeat can also carry data.
- ***Exceptions***. The exception handler typically executes in the same process that introduced the exception.

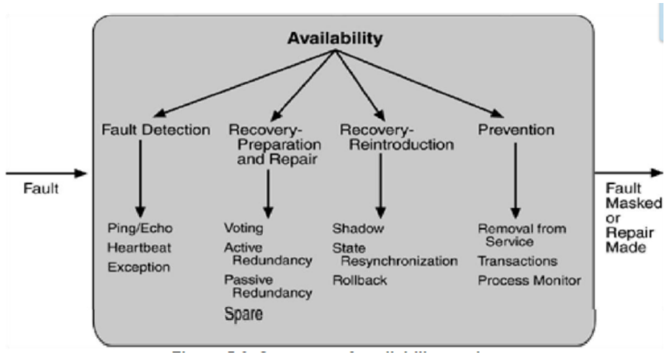
FAULT RECOVERY

- ***Voting.*** Processes running on redundant processors each take equivalent input and compute a simple output value that is sent to a voter. If the voter detects deviant behavior from a single processor, it fails it.
- ***Active redundancy (hot restart).*** All redundant components respond to events in parallel. The response from only one component is used (usually the first to respond), and the rest are discarded. Active redundancy is often used in a client/server configuration, such as database management systems, where quick responses are necessary even when a fault occurs
- ***Passive redundancy (warm restart/dual redundancy/triple redundancy)***. One component (the primary) responds to events and informs the other components (the standbys) of state updates they must make. When a fault occurs, the system must first ensure that the backup state is sufficiently fresh before resuming services.
- ***Spare***. A standby spare computing platform is configured to replace many different failed components. It must be rebooted to the appropriate software configuration and have its state initialized when a failure occurs.
- ***Shadow operation***. A previously failed component may be run in "shadow mode" for a short time to make sure that it mimics the behavior of the working components before restoring it to service.
- ***State resynchronization***. The passive and active redundancy tactics require the component being restored to have its state upgraded before its return to service.
- ***Checkpoint/rollback***. A checkpoint is a recording of a consistent state created either periodically or in response to specific events. Sometimes a system fails in an unusual manner,

with a detectably inconsistent state. In this case, the system should be restored using a previous checkpoint of a consistent state and a log of the transactions that occurred since the snapshot was taken.

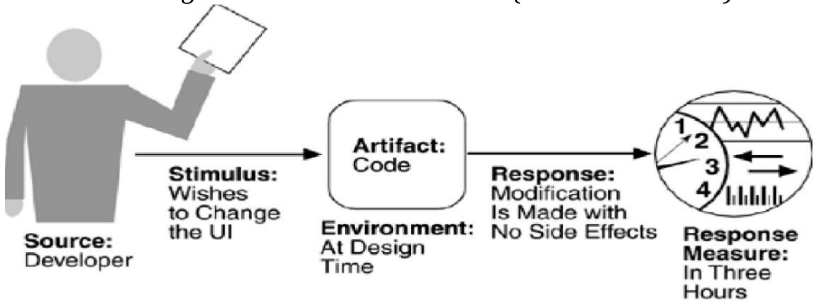
FAULT PREVENTION

- **R*emoval from service*.** This tactic removes a component of the system from operation to undergo some activities to prevent anticipated failures.
- ***Transactions***. A transaction is the bundling of several sequential steps such that the entire bundle can be undone at once. Transactions are used to prevent any data from being affected if one step in a process fails and also to prevent collisions among several simultaneous threads accessing the same data.
- ***Process monitor***. Once a fault in a process has been detected, a monitoring process can delete the nonperforming process and create a new instance of it, initialized to some appropriate state as in the spare tactic.



SUMMARY OF AVAILABILITY TACTICS

4. **<u>Describe modifiability general scenario in detail.</u>**
   - Modifiability is about the cost of change. It brings up two concerns.
     - *What can change (the artifact)?*
     - *When is the change made and who makes it (the environment)?*


Artifact: Code
Stimulus: Wishes to Change the UI
Source: Developer
Environment: At Design Time
Response: Modification Is Made with No Side Effects
Response Measure: In Three Hours
1
2
3
4

MODIFIABILITY GENERAL SCENARIO

- ***Source of stimulus***. This portion specifies who makes the changes—the developer, a system administrator, or an end user. Clearly, there must be machinery in place to allow the system administrator or end user to modify a system, but this is a common occurrence. In the above Figure the modification is to be made by the developer.
- ***Stimulus.*** This portion specifies the changes to be made. A change can be the addition of a function, the modification of an existing function, or the deletion of a function. It can also be made to the qualities of the system—making it more responsive, increasing its availability, and so forth. The capacity of the system may also change. Increasing the number of simultaneous users is a frequent requirement. In our example, the stimulus is a request to make a modification, which can be to the function, quality, or capacity.

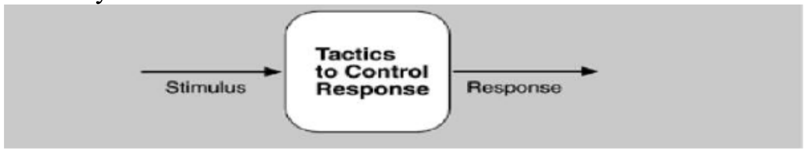
- ***Artifact.*** This portion specifies what is to be changed—the functionality of a system, its platform, its user interface, its environment, or another system with which it interoperates. In Figure 4.4, the modification is to the user interface.
- ***Environment.*** This portion specifies when the change can be made—design time, compile time, build time, initiation time, or runtime. In our example, the modification is to occur at design time.
- ***Response.*** Whoever makes the change must understand how to make it, and then make it, test it and deploy it. In our example, the modification is made with no side effects.
- ***Response measure***. All of the possible responses take time and cost money, and so time and cost are the most desirable measures. Time is not always possible to predict, however, and so less ideal measures are frequently used, such as the extent of the change (number of modules affected). In our example, the time to perform the modification should be less than three hours.

Modifiability General Scenario

Scenario Portion	**Possible Values**
Source	End-user, developer, system-administrator
Stimulus	Add/delete/modify functionality or quality attr.
Artifact	System user interface, platform, environment
Environment	At runtime, compile time, build time, design-time
Response	Locate places in architecture for modifying, modify, test modification, deploys modification
RespMeasure	Cost in effort, money, time, extent affects other system functions or qualities

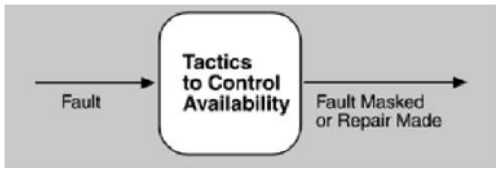
5. **Explain the following with respect to tactics**
   a. AVAILABILITY

- A **tactic** is a design decision that influences the control of a quality attribute response.
- Patterns package tactics. A pattern that supports availability will likely use both a redundancy tactic and a synchronization tactic.
- Tactics can refine other tactics. For each quality attribute that we discuss, we organize the tactics as a hierarchy.



Tactics are intended to control responses to stimuli.

AVAILABILITY TACTICS



- The above figure depicts goal of availability tactics. All approaches to maintaining availability involve some type of redundancy, some type of health monitoring to detect a failure, and some type of recovery when a failure is detected. In some cases, the monitoring or recovery is automatic and in others it is manual.

FAULT DETECTION

- ***Ping/echo***.
- ***Heartbeat (dead man timer).***
- ***Exceptions***

FAULT RECOVERY

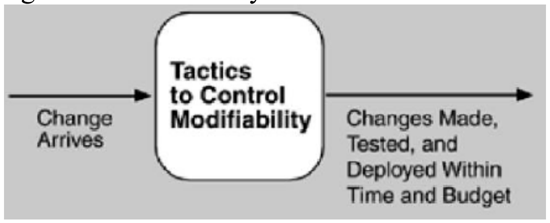
- ***Voting***
- ***Active redundancy (hot restart).***
- ***Passive redundancy (warm restart/dual redundancy/triple redundancy)***.
- ***Spare***.
- ***Shadow operation***.
- ***State resynchronization***.
- ***Checkpoint/rollback***.

FAULT PREVENTION

- **R*emoval from service***..
- ***Transactions***.
- ***Process monitor***.

b.<u>**MODIFIABILITY**</u>

The figure below represent goal of modifiability tactics.


Change Arrives
Tactics to Control Modifiability
Changes Made, Tested, and Deployed Within Time and Budget

LOCALIZE MODIFICATIONS

- ***Maintain semantic coherence***. Semantic coherence refers to the relationships among responsibilities in a module. The goal is to ensure that all of these responsibilities work together without excessive reliance on other modules.
- ***Anticipate expected changes***. Considering the set of envisioned changes provides a way to evaluate a particular assignment of responsibilities. In reality this tactic is difficult to use by itself since it is not possible to anticipate all changes.
- ***Generalize the module***. Making a module more general allows it to compute a broader range of functions based on input
- ***Limit possible options***. Modifications, especially within a product line, may be far ranging and hence affect many modules. Restricting the possible options will reduce the effect of these modifications

PREVENT RIPPLE EFFECTS We begin our discussion of the ripple effect by discussing the various types of dependencies that one module can have on another. We identify eight types:

1. Syntax of
   - *data.*
   - *service.*
2. Semantics of
   - *data.*
   - *service.*
3. Sequence of
   - *data.*
   - *control.*
4. *Identity of an interface of A*
5. *Location of A (runtime).*

6. *Quality of service/data provided by A.*
7. *Existence of A*
8. *Resource behaviour of A.*

With this understanding of dependency types, we can now discuss tactics available to the architect for preventing the ripple effect for certain types.

- ***Hide information***. Information hiding is the decomposition of the responsibilities for an entity into smaller pieces and choosing which information to make private and which to make public. The goal is to isolate changes within one module and prevent changes from propagating to others
- ***Maintain existing interfaces*** it is difficult to mask dependencies on quality of data or quality of service, resource usage, or resource ownership. Interface stability can also be achieved by separating the interface from the implementation. This allows the creation of abstract interfaces that mask variations.
- ***Restrict communication paths***. This will reduce the ripple effect since data production/consumption introduces dependencies that cause ripples.
- ***Use an intermediary*** If B has any type of dependency on A other than semantic, it is possible to insert an intermediary between B and A that manages activities associated with the dependency.

DEFER BINDING TIME Many tactics are intended to have impact at load time or runtime, such as the following.

- ***Runtime registration*** supports plug-and-play operation at the cost of additional overhead to manage the registration.
- ***Configuration files*** are intended to set parameters at startup.
- ***Polymorphism*** allows late binding of method calls.
- ***Component replacement*** allows load time binding.
- ***Adherence to defined protocols*** allows runtime binding of independent processes.

c. FAULT PREVENTION

- **R*emoval from service***. This tactic removes a component of the system from operation to undergo some activities to prevent anticipated failures.
- ***Transactions***. A transaction is the bundling of several sequential steps such that the entire bundle can be undone at once. Transactions are used to prevent any data from being affected if one step in a process fails and also to prevent collisions among several simultaneous threads accessing the same data.
- ***Process monitor***. Once a fault in a process has been detected, a monitoring process can delete the nonperforming process and create a new instance of it, initialized to some appropriate state as in the spare tactic.

d. DEFER BINDING TIME

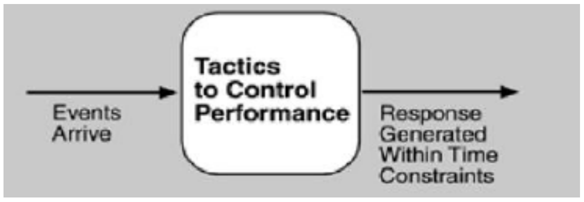
- **Defer Binding Time:** When a modification is made by the developer, there is usually a testing and distribution process that determines the time lag between the making of the change and the availability of that change to the end user.
- Binding at runtime means that the system has been prepared for that binding and all of the testing and distribution steps have been completed.
- Deferring binding time also supports allowing the end user or system administrator to make setting or provide input that affects behavior.
- **Runtime registration:** Publish/subscribe for example.
- **Configuration files:** are intended to set parameters at startup.
- **Polymorphism:** allows late binding of method calls

- **Component replacement:** allows load time binding.
- **Adherence to defined protocols:** allows runtime binding of independent processes.

6. <u>Explain the following with respect to tactics</u>

a. <u>PERFORMANCE</u>

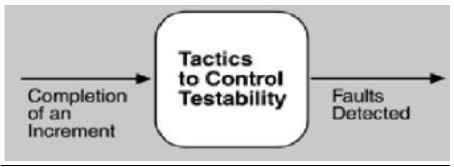
Performance tactics control the time within which a response is generated. Goal of performance tactics is shown below:


Events Arrive
Tactics to Control Performance
Response Generated Within Time Constraints

- After an event arrives, either the system is processing on that event or the processing is blocked for some reason. This leads to the two basic contributors to the response time: resource consumption and blocked time.
- ***Resource consumption.*** For example, a message is generated by one component, is placed on the network, and arrives at another component. Each of these phases contributes to the overall latency of the processing of that event.
- ***Blocked time.*** A computation can be blocked from using a resource because of contention for it, because the resource is unavailable, or because the computation depends on the result of other computations that are not yet available.
    - *Contention for resources*
    - *Availability of resources*
    - *Dependency on other computation.*
- **RESOURCE DEMAND**
    - One tactic for reducing latency is to reduce the resources required for processing an event stream.
        - ***Increase computational efficiency***.
        - ***Reduce computational overhead***.
        - ***Manage event rate.***
        - ***Control frequency of sampling.***
    - Other tactics for reducing or managing demand involve controlling the use of resources.
        - ***Bound execution times***.
        - ***Bound queue sizes***.
- **RESOURCE MANAGEMENT**
    - ***Introduce concurrency***.
    - ***Maintain multiple copies of either data or computations***.
    - ***Increase available resources***.
- **RESOURCE ARBITRATION**
    - ***First-in/First-out***.
    - ***Fixed-priority scheduling.***
    - ***Dynamic priority scheduling:***
        - o *round robin*
        - o *earliest deadline first.*
    - ***Static scheduling***.

b. <u>TESTABILITY</u>

The goal of tactics for testability is to allow for easier testing when an increment of software development is completed. Figure below displays the use of tactics for testability.



INPUT/OUTPUT

- ***Record/playback***. Record/playback refers to both capturing information crossing an interface and using it as input into the test harness. The information crossing an interface during normal operation is saved in some repository. Recording this information allows test input for one of the components to be generated and test output for later comparison to be saved.
- ***Separate interface from implementation.*** Separating the interface from the implementation allows substitution of implementations for various testing purposes. Stubbing implementations allows the remainder of the system to be tested in the absence of the component being stubbed.
- ***Specialize access routes/interfaces.*** Having specialized testing interfaces allows the capturing or specification of variable values for a component through a test harness as well as independently from its normal execution. Specialized access routes and interfaces should be kept separate from the access routes and interfaces for required functionality.

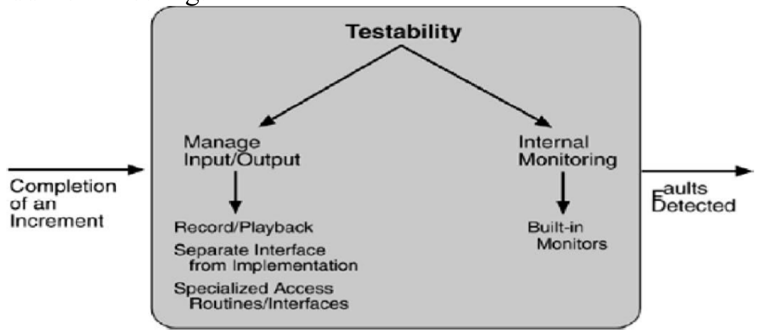
c. <u>RESOURCE ARBITRATION</u>

- ***First-in/First-out***. FIFO queues treat all requests for resources as equals and satisfy them in turn.
- ***Fixed-priority scheduling.*** Fixed-priority scheduling assigns each source of resource requests a particular priority and assigns the resources in that priority order.
- Three common prioritization strategies are
  - *Semantic importance.* Each stream is assigned a priority statically according to some domain characteristic of the task that generates it.
  - *Deadline monotonic.* Deadline monotonic is a static priority assignment that assigns higher priority to streams with shorter deadlines.
  - *Rate monotonic.* Rate monotonic is a static priority assignment for periodic streams that assigns higher priority to streams with shorter periods.
- ***Dynamic priority scheduling:***
  - *Round robin.* Round robin is a scheduling strategy that orders the requests and then, at every assignment possibility, assigns the resource to the next request in that order.
  - *Earliest deadline first.* Earliest deadline first assigns priorities based on the pending requests with the earliest deadline.
- ***Static scheduling***. A cyclic executive schedule is a scheduling strategy where the pre-emption points and the sequence of assignment to the resource are determined offline.

d. <u>INTERNAL MONITORING</u>

- ***Built-in monitors***. The component can maintain state, performance load, capacity, security, or other information accessible through an interface.
- This interface can be a permanent interface of the component or it can be introduced temporarily.
- A common technique is to record events when monitoring states have been activated.

- Monitoring states can actually increase the testing effort since tests may have to be repeated with the monitoring turned off.
- Increased visibility into the activities of the component usually more than outweigh the cost of the additional testing.


Testability
Manage Input/Output
Internal Monitoring
Completion of an Increment
Faults Detected
Record/Playback
Separate Interface from Implementation
Specialized Access Routines/Interfaces
Built-in Monitors

7. **<u>a. Explain about faults detection and recovery.</u>**

FAULT DETECTION

- ***Ping/echo***. One component issues a ping and expects to receive back an echo, within a predefined time, from the component under scrutiny. This can be used within a group of components mutually responsible for one task
- ***Heartbeat (dead man timer).*** In this case one component emits a heartbeat message periodically and nother component listens for it. If the heartbeat fails, the originating component is assumed to have failed and a fault correction component is notified. The heartbeat can also carry data.
- ***Exceptions***. The exception handler typically executes in the same process that introduced the exception.

FAULT RECOVERY

- ***Voting.*** Processes running on redundant processors each take equivalent input and compute a simple output value that is sent to a voter. If the voter detects deviant behavior from a single processor, it fails it.
- ***Active redundancy (hot restart).*** All redundant components respond to events in parallel. The response from only one component is used (usually the first to respond), and the rest are discarded. Active redundancy is often used in a client/server configuration, such as database management systems, where quick responses are necessary even when a fault occurs
- ***Passive redundancy (warm restart/dual redundancy/triple redundancy)***. One component (the primary) responds to events and informs the other components (the standbys) of state updates they must make. When a fault occurs, the system must first ensure that the backup state is sufficiently fresh before resuming services.
- ***Spare***. A standby spare computing platform is configured to replace many different failed components. It must be rebooted to the appropriate software configuration and have its state initialized when a failure occurs.
- ***Shadow operation***. A previously failed component may be run in "shadow mode" for a short time to make sure that it mimics the behavior of the working components before restoring it to service.
- ***State resynchronization***. The passive and active redundancy tactics require the component being restored to have its state upgraded before its return to service.
- ***Checkpoint/rollback***. A checkpoint is a recording of a consistent state created either periodically or in response to specific events. Sometimes a system fails in an unusual manner, with a detectably inconsistent state. In this case, the system should be restored using a previous

checkpoint of a consistent state and a log of the transactions that occurred since the snapshot was taken.

b. <u>Explain about the run-time and design time tactics.</u>

RUNTIME TACTICS

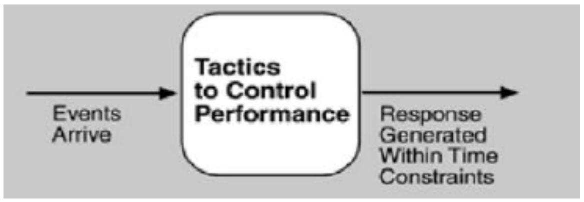
- Maintain a model of the task.
  - In this case, the model maintained is that of the task. The task model is used to determine context so the system can have some idea of what the user is attempting and provide various kinds of assistance. For example, knowing that sentences usually start with capital letters would allow an application to correct a lower-case letter in that position.
- Maintain a model of the user.
  - In this case, the model maintained is of the user. It determines the user's knowledge of the system, the user's behavior in terms of expected response time, and other aspects specific to a user or a class of users. For example, maintaining a user model allows the system to pace scrolling so that pages do not fly past faster than they can be read.
- Maintain a model of the system.
  - In this case, the model maintained is that of the system. It determines the expected system behavior so that appropriate feedback can be given to the user. The system model predicts items such as the time needed to complete current activity.

DESIGN-TIME TACTICS

- Separate the user interface from the rest of the application. Localizing expected changes is the rationale for semantic coherence. Since the user interface is expected to change frequently both during the development and after deployment, maintaining the user interface code separately will localize changes to it.
- The software architecture patterns developed to implement this tactic and to support the modification of the user interface are:
  - Model-View-Controller
  - Presentation-Abstraction-Control
  - Seeheim
  - Arch/Slinky

8. <u>Explain in detail about the performance tactics with neat diagram.</u>

Performance tactics control the time within which a response is generated. Goal of performance tactics is shown below:


Events Arrive
Tactics to Control Performance
Response Generated Within Time Constraints

- After an event arrives, either the system is processing on that event or the processing is blocked for some reason. This leads to the two basic contributors to the response time: resource consumption and blocked time.
- ***Resource consumption.*** For example, a message is generated by one component, is placed on the network, and arrives at another component. Each of these phases contributes to the overall latency of the processing of that event.

- ***Blocked time.*** A computation can be blocked from using a resource because of contention for it, because the resource is unavailable, or because the computation depends on the result of other computations that are not yet available.
- *Contention for resources*
- *Availability of resources*
- *Dependency on other computation.*

RESOURCE DEMAND

One tactic for reducing latency is to reduce the resources required for processing an event stream.

- ***Increase computational efficiency***. One step in the processing of an event or a message is applying some algorithm. Improving the algorithms used in critical areas will decrease latency. This tactic is usually applied to the processor but is also effective when applied to other resources such as a disk.
- ***Reduce computational overhead***. If there is no request for a resource, processing needs are reduced. The use of intermediaries increases the resources consumed in processing an event stream, and so removing them improves latency.
- Another tactic for reducing latency is to reduce the number of events processed. This can be done in one of two fashions.
- ***Manage event rate.*** If it is possible to reduce the sampling frequency at which environmental variables are monitored, demand can be reduced.
- ***Control frequency of sampling.*** If there is no control over the arrival of externally generated events, queued requests can be sampled at a lower frequency, possibly resulting in the loss of requests.
- Other tactics for reducing or managing demand involve controlling the use of resources.
- ***Bound execution times***. Place a limit on how much execution time is used to respond to an event. Sometimes this makes sense and sometimes it does not.
- ***Bound queue sizes***. This controls the maximum number of queued arrivals and consequently the resources used to process the arrivals.

RESOURCE MANAGEMENT

- ***Introduce concurrency***. If requests can be processed in parallel, the blocked time can be reduced.
- ***Maintain multiple copies of either data or computations***. Clients in a client-server pattern are replicas of the computation. The purpose of replicas is to reduce the contention that would occur if all computations took place on a central server.
- ***Increase available resources***. Faster processors, additional processors, additional memory, and faster networks all have the potential for reducing latency.

RESOURCE ARBITRATION

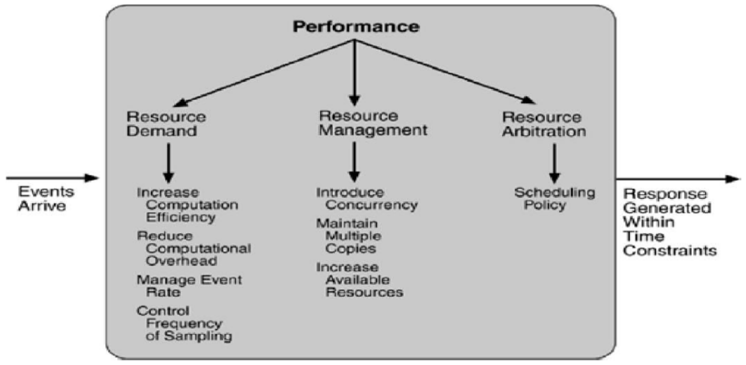
- ***First-in/First-out***. FIFO queues treat all requests for resources as equals and satisfy them in turn
- ***Fixed-priority scheduling.*** Fixed-priority scheduling assigns each source of resource requests a particular priority and assigns the resources in that priority order.

Three common prioritization strategies are

- *Semantic importance.* Each stream is assigned a priority statically according to some domain characteristic of the task that generates it.
- *Deadline monotonic.* Deadline monotonic is a static priority assignment that assigns higher priority to streams with shorter deadlines.
- *Rate monotonic.* Rate monotonic is a static priority assignment for periodic streams that assigns higher priority to streams with shorter periods.

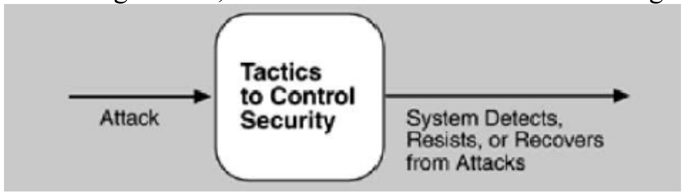
Dynamic priority scheduling:

- *Round robin.* Round robin is a scheduling strategy that orders the requests and then, at every assignment possibility, assigns the resource to the next request in that order.
- *Earliest deadline first.* Earliest deadline first assigns priorities based on the pending requests with the earliest deadline.
- ***Static scheduling***. A cyclic executive schedule is a scheduling strategy where the pre-emption points and the sequence of assignment to the resource are determined offline.


Performance
Resource Demand
Resource Management
Resource Arbitration
Events Arrive
Increase Computation Efficiency
Reduce Computational Overhead
Manage Event Rate
Control Frequency of Sampling
Introduce Concurrency
Maintain Multiple Copies
Increase Available Resources
Scheduling Policy
Response Generated Within Time Constraints

9. <u>a. Classify security tactics. What are the different tactics for resisting attacks?</u>

Tactics for achieving security can be divided into those concerned with resisting attacks, those concerned with detecting attacks, and those concerned with recovering from attacks.


Attack
Tactics to Control Security
System Detects, Resists, or Recovers from Attacks

Three Classification: Resisting Attack, Detecting and Recovering from attacks

RESISTING ATTACKS

- ***Authenticate users.*** Authentication is ensuring that a user or remote computer is actually who it purports to be. Passwords, one-time passwords, digital certificates, and biometric identifications provide authentication.
- ***Authorize users.*** Authorization is ensuring that an authenticated user has the rights to access and modify either data or services. Access control can be by user or by user class.
- ***Maintain data confidentiality***. Data should be protected from unauthorized access. Confidentiality is usually achieved by applying some form of encryption to data and to communication links. Encryption provides extra protection to persistently maintained data beyond that available from authorization.
- ***Maintain integrity.*** Data should be delivered as intended. It can have redundant information encoded in it, such as checksums or hash results, which can be encrypted either along with or independently from the original data.
- ***Limit exposure.*** Attacks typically depend on exploiting a single weakness to attack all data and services on a host. The architect can design the allocation of services to hosts so that limited services are available on each host.
- ***Limit access.*** Firewalls restrict access based on message source or destination port. Messages from unknown sources may be a form of an attack. It is not always possible to limit access to known sources.

DETECTING ATTACKS

- The detection of an attack is usually through an *intrusion detection* system.

- Such systems work by comparing network traffic patterns to a database.
- In the case of misuse detection, the traffic pattern is compared to historic patterns of known attacks.
- In the case of anomaly detection, the traffic pattern is compared to a historical baseline of itself. Frequently, the packets must be filtered in order to make comparisons.
- Filtering can be on the basis of protocol, TCP flags, payload sizes, source or destination address, or port number.
- Intrusion detectors must have some sort of sensor to detect attacks, managers to do sensor fusion, databases for storing events for later analysis, tools for offline reporting and analysis, and a control console so that the analyst can modify intrusion detection actions.

RECOVERING FROM ATTACKS

- Tactics involved in recovering from an attack can be divided into those concerned with restoring state and those concerned with attacker identification (for either preventive or punitive purposes).
- One difference is that special attention is paid to maintaining redundant copies of system administrative data such as passwords, access control lists, domain name services, and user profile data.
- The tactic for identifying an attacker is to *maintain an audit trail*.
- An audit trail is a copy of each transaction applied to the data in the system together with identifying information.
- Audit information can be used to trace the actions of an attacker, support nonrepudiation (it provides evidence that a particular request was made), and support system recovery.
- Audit trails are often attack targets themselves and therefore should be maintained in a trusted fashion.

b. <u>List out the Business and Architecture qualities</u>

<u>BUSINESS QUALITIES</u>

- ***Time to market.***
  - If there is competitive pressure or a short window of opportunity for a system or product, development time becomes important. This in turn leads to pressure to buy or otherwise re-use existing elements.
- ***Cost and benefit.***
  - 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) not resident in the developing organization will be more expensive to realize than one that takes advantage of assets already inhouse. An architecture that is highly flexible will typically be more costly to build than one that is rigid (although it will be less costly to maintain and modify).
- ***Projected lifetime of the system.***
  - If the system is intended to have a long lifetime, modifiability, scalability, and portability become important. 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.
- ***Rollout schedule.***

- If a product is to be introduced as base functionality with many features released later, the flexibility and customizability of the architecture are important. Particularly, the system must be constructed with ease of expansion and contraction in mind.

- ***Integration with legacy systems.***

  - If the new system has to *integrate* with existing systems, care must be taken to define appropriate integration mechanisms. This property is clearly of marketing importance but has substantial architectural implications.

ARCHITECTURE QUALITIES

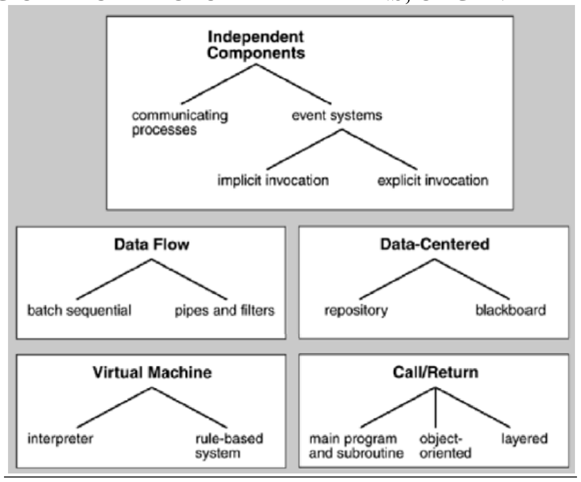
- ***Conceptual integrity*** is the underlying theme or vision that unifies the design of the system at all levels. The architecture should do similar things in similar ways.
- ***Correctness*** **and** ***completeness*** are essential for the architecture to allow for all of the system's requirements and runtime resource constraints to be met.
- ***Buildability*** allows the system to be completed by the available team in a timely manner and to be open to certain changes as development progresses.

10. <u>Explain the Architectural patterns and styles and describe the relationship of tactics to architectural patterns.</u>

An architectural pattern is determined by:

- A ***set of element types*** (such as a data repository or a component that computes a 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.

A SMALL CATALOG OF ARCHITECTURAL PATTERNS, ORGANIZED BY IS-A RELATIONS



RELATIONSHIP OF TACTICS TO ARCHITECTURAL PATTERNS

- **The pattern consists of six elements:**
  - a ***proxy***, which provides an interface that allows clients to invoke publicly accessible methods on an active object;

- a ***method request***, which defines an interface for executing the methods of an active object;
- an ***activation list***, which maintains a buffer of pending method requests;
- a ***scheduler***, which decides what method requests to execute next;
- a ***servant***, which defines the behavior and state modeled as an active object; and
- a ***future***, which allows the client to obtain the result of the method invocation.

The tactics involves the following:

- ***Information hiding*** *(modifiability).* Each element chooses the responsibilities it will achieve and hides their achievement behind an interface.
- ***Intermediary*** *(modifiability).* The proxy acts as an intermediary that will buffer changes to the method invocation.
- ***Binding time*** *(modifiability).* The active object pattern assumes that requests for the object arrive at the object at runtime. The binding of the client to the proxy, however, is left open in terms of binding time.
- ***Scheduling policy*** *(performance).* The scheduler implements some scheduling policy.

UNIT – III ARCHITECTURAL VIEWS

PART- A

1. What 4+1 architectural view?

Describing the architecture of software-intensive systems, based on the use of multiple, concurrent views.

2. What is the different 4+1 view?

- Logical view
- Development view
- Process view
- Physical view

3. What are the three broad types of decisions that architectural design involves?

- How is the system to be structured as a set of code units (modules?)
- How is the system to be structured as a set of elements that have runtime behavior (components) and interactions (connectors)
- How is the system to relate to non-software structures in its environment (i.e., CPUs, file systems, networks, development teams, etc. - allocation)

4. Define Common software architecture structures.

Module: Decomposition,class,uses,layered
Component connectore: client-server,concurrency,process,shared data
Allocation: Work Assignment,deployment,implementation

5. What do you mean by Shared data, or repository?

This structure comprises components and connectors that create, store, and access persistent data. If the system is in fact structured around one or more shared data repositories, this structure is a good one to illuminate.

It shows how data is produced and consumed by runtime software elements, and it can be used to ensure good performance and data integrity.

6. What is deployment?

Deployment shows how software is assigned to hardware-processing and communication elements.

7. How do structures relate each other?

Although the structures give different system perspectives, they are not independent.Elements of one structure are related to elements in another, and we need to reason about these relationships.

8. Define Kruchten's Four plus One Views.

- Logical - elements are "key abstractions" that are objects or classes in OO. This is a module view.
- Process - addresses concurrency & distribution of functionality. This is a C&C view.
- Development - shows organization of software modules, libraries, subsystems, and units of development. This is an allocation view.
- Physical - maps other elements onto processing & communication nodes, also an allocation view, but usually referred to specifically as the deployment view.

9. What do you mean by concurrency?

This component-and-connector structure allows the architect to determine opportunities for parallelism and the locations where resource contention may occur.

10. What is allocation structure?

Show relationships between software elements & external environments (creation or execution)

11. What is Module Architecture View?

The module architecture view explains how the components, connectors, ports, and roles are mapped to abstract modules and their interfaces. The system is decomposed into modules and subsystems. A module can also be assigned to a layer, which then constrains its dependencies on other modules.

12. What are the Conceptual View Activities?

- Central design tasks.
  - Create conceptual…
    - Components, connectors, ports, roles, protocols.
    - Configuration.
  - Global evaluation.
- Final design task.
  - Software resource budgeting (memory, CPU time).

13. Define Siemens' 4 View.

Elements	CComponent	Component
	CPort	Port
	CConnector	Connector
	CRole	Role
	Protocol	Protocol
Relations	Composition	Decomposition
	Cbinding	Binding
	Cconnection	Attachment
	Obeys, obeys congugate	Element property

14. Define client- server.

- If the system is built as a group of cooperating clients and servers, this is a good component-and-connector structure to illuminate.
- The components are the clients and servers, and the connectors are protocols and messages they share to carry out the system's work.

15. What are the different Module-based structures?

- Decomposition.
- Uses
- Layered

- Class or generalization

16. Which of the following include allocation structure?

a) Deployment.
b) Implementation
c) Work assignment.

17. What do you mean by logical view?

- (Object-oriented Decomposition)
- Viewer: End-user
- Considers: Functional requirements- What the system provide in terms of services to its users.
- Notation: The Booch notation (OMT) (object and dynamic models)
- Tool: Rational Rose

18. Define Iterative process.

- Not all software arch. Need all views.A scenario-driven approach to develop the system.
- Documentation:
    - Software architecture document
    - Software design guidelines

19. What are the essential parts of a process-control loop.

- Computational elements:
- Data element
- The control loop paradigm

20. Define 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.

PART- B

1. **Explain in detail about Standard views?**

- A view is a representation of one or more structural aspects of an architecture that illustrates how the architecture addresses one or more concerns held by one or more of its stakeholders.

- An architectural view is a way to portray those aspects or elements of the architecture that are relevant to the concerns the view intends to address—and, by implication, the stakeholders for whom those concerns are important.

- A view may be narrowly focused on one class of stakeholder or even a specific individual, or it may be aimed at a larger group whose members have varying interests and levels of expertise.

Context:

- Describes the relationships, dependencies, and interactions between the system and its environment (the people, systems, and external entities with which it interacts).
- Many architecture descriptions focus on views that model the system's internal structures, data elements, interactions, and operation.
- Architects tend to assume that the "outward-facing" information — the system's runtime context, its scope and requirements, and so forth – is clearly and unambiguously defined elsewhere.
- However, you often need to include a definition of the system's context as part of your architectural description.

Functional:

- Describes the system's functional elements, their responsibilities, interfaces, and primary interactions.
- A Functional view is the cornerstone of most ADs and is often the first part of the description that stakeholders try to read.
- It drives the shape of other system structures such as the information structure, concurrency structure, deployment structure, and so on.
- It also has a significant impact on the system's quality properties such as its ability to change, its ability to be secured, and its runtime performance.

Information:

- Describes the way that the architecture stores, manipulates, manages, and distributes information.
- The ultimate purpose of virtually any computer system is to manipulate information in some form, and this viewpoint develops a complete but high-level view of static data structure and information flow.
- The objective of this analysis is to answer the big questions around content, structure, ownership, latency, references, and data migration.

Concurrency:

- Describes the concurrency structure of the system and maps functional elements to concurrency units to clearly identify the parts of the system that can execute concurrently and how this is coordinated and controlled.
- This entails the creation of models that show the process and thread structures that the system will use and the inter process communication mechanisms used to coordinate their operation.

Development:

- Describes the architecture that supports the software development process.
- Development views communicate the aspects of the architecture of interest to those stakeholders involved in building, testing, maintaining, and enhancing the system.

Deployment:

- Describes the environment into which the system will be deployed, including capturing the dependencies the system has on its runtime environment.
- This view captures the hardware environment that your system needs (primarily the processing nodes, network interconnections, and disk storage facilities required), the technical environment requirements for each element, and the mapping of the software elements to the runtime environment that will execute them.

Operational:

- Describes how the system will be operated, administered, and supported when it is running in its production environment.
- For all but the simplest systems, installing, managing, and operating the system is a significant task that must be considered and planned at design time.
- The aim of the Operational viewpoint is to identify system-wide strategies for addressing the operational concerns of the system's stakeholders and to identify solutions that address these.

2. Explain in detail about Siemens four views with example?

Siemens four views are developed at Siemens Corporate Research

This is based on best architecture practices for industrial systems

The Siemens approach uses four views to document an architecture.

- Conceptual View
- Module view
- Code view
- Execution view

These four views separate different engineering concerns, thus reducing the complexity of the architecture design task.

1. Conceptual view: The primary engineering concerns in this view are to address how the system fulfills the requirements. The functional requirements are a central concern.

2. Module view: Modules are organized into two orthogonal structures: decomposition and layers. The decomposition structure captures how the system is logically decomposed into subsystems and modules. A module can be assigned to a layer, which then constrains its dependencies on other modules. The primary concerns of this view are to minimize dependencies between modules, maximize reuse of modules, and support testing.

3. Execution architecture view: This view describes the system's structure in terms of its runtime platform elements. Runtime properties of the system, such as performance, safety, and replication are also addressed here.

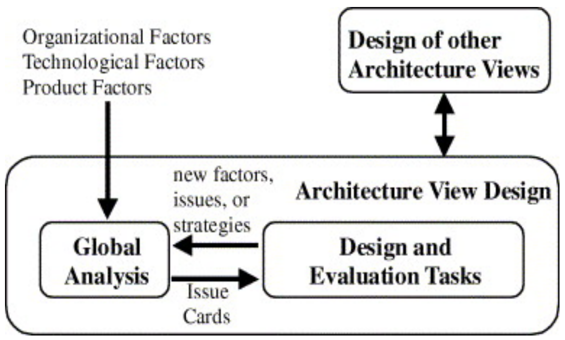
4. Code architecture view: This is concerned with the organization of the software artifacts. The engineering concerns of this view are to make support product versions and releases, minimize effort for product upgrades, minimize build time, and support integration and testing.

- Factors that influence the architecture are organized into three categories:
  i. organizational
  ii. technological
  iii. product factors.
- These factors are analyzed in order to determine which factors conflict, what are their relative priorities, how flexible and stable is each factor, what is the impact of a change in the factor, and what are strategies for reducing that impact.
- From these factors the key architectural issues or challenges are identified.

- The next step is to propose design strategies to solve the issues, and to apply the design strategies to one or more of the views.

- During design, the design decisions are evaluated, particularly for conflicts and unexpected interactions. This evaluation is ongoing. Thus Global Analysis activities are interleaved with view design activities, and the design activities of each view are also interleaved.

Workflow between Global Analysis and Architecture View Design.



Periodic architecture evaluation

- In contrast to the ongoing evaluation that is part of the design process, periodic architecture evaluation is done in order to answer a specific question, such as cost prediction, risk assessment, or some specific comparison or tradeoff.

- This typically involves other stakeholders in addition to the architect.

- Global Analysis provides inputs to this kind of architecture evaluation, for example: business drivers, quality attributes, architectural approaches, risks, tradeoffs, and architectural approaches.

3. Briefly describe about 4+1 views?

- The Rational Unified Process (RUP) is a software development process developed and commercialized by Rational Software, now IBM.

- For RUP
  Software architecture encompasses the set of significant decisions about the organization of a software system:

  • selection of the structural elements and their interfaces by which a system is composed,

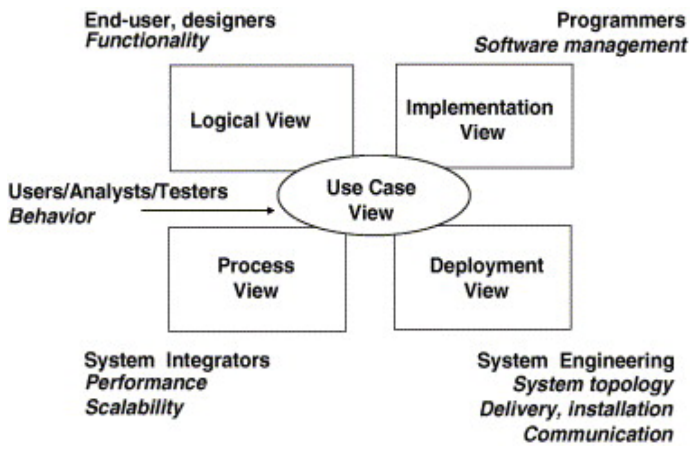
  • behavior as specified in collaborations among those elements,

  • composition of these structural and behavioral elements into larger subsystem,

  • architectural style that guides this organization.
  Software architecture also involves: usage, functionality, performance, resilience, reuse, comprehensibility, economic and technology constraints and tradeoffs, and aesthetic concerns.

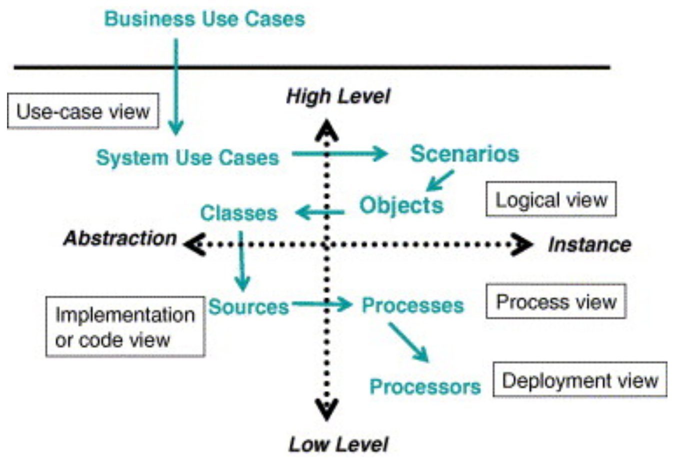
- RUP defines an architectural design method, using the concept of 4 + 1 view (RUP 4 + 1):

  The four views that are used to describe the design are: logical view, process view, implementation view and deployment view, and using a use-case view (+1) to relate the design to the context and goals.



In RUP, architectural design is spread over several iterations in an *elaboration phase*, iteratively populating the 4 views, driven by architecturally significant use cases, non-functional requirements in the supplementary specification, and risks.

Each iteration results in an *executable architectural prototype*, which is used to validate the architectural design.



- Architectural design activities in RUP start with the following artifacts:
  A vision document, a use-case model (functional requirements) and supplementary specification (non-functional requirements, quality attributes).

- The three main groups of activities are:

  (1) Define a Candidate Architecture: This usually starts with a use-case analysis, focusing on the use cases that are deemed architecturally significant, and with any reference architecture the organization may reuse.
  This activities leads to a first candidate architecture that can be prototyped and used to further reason with the architectural design, integrating more elements later on.

  (2) Perform Architectural Synthesis:
  Build an architectural proof-of-concept, and assess its viability, relative to functionality and to non-functional requirements.

  (3) Refine the Architecture

Identify design elements (classes, processes, etc.) and integrate them in the architectural prototype
Identify design mechanisms (e.g., architectural patterns and services), particular those that deal with concurrency, persistency, distribution.

4. Explain how the structures relate with each other with an example?

Software Structure	**Relations**	**Useful for**
Decomposition	Is a submodule of; shares secret with	Resource allocation and project structuring and planning; information hiding, encapsulation; configuration control
Uses	Requires the correct presence of	Engineering subsets; engineering extensions
Layered	Requires the correct presence of; uses the services of; provides abstraction to	Incremental development; implementing systems on top of "virtual machines" portability
Class	Is an instance of; shares access methods of	In object-oriented design systems, producing rapid almost-alike implementations from a common template
Client-Server	Communicates with; depends on	Distributed operation; separation of concerns; performance analysis; load balancing
Process	Runs concurrently with; may run concurrently with; excludes; precedes; etc.	Scheduling analysis; performance analysis
Concurrency	Runs on the same logical thread	Identifying locations where resource contention exists, where threads may fork, join, be created or be killed
Shared Data	Produces data; consumes data	Performance; data integrity; modifiability
Deployment	Allocated to; migrates to	Performance, availability, security analysis
Implementation	Stored in	Configuration control, integration, test activities
Work Assignment	Assigned to	Project management, best use of expertise, management of commonality

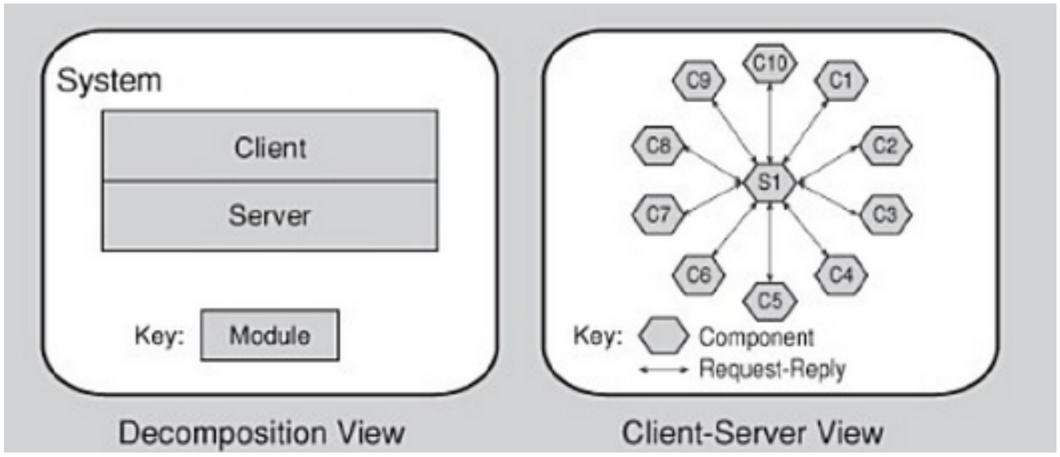
- Each structure provides a method for reasoning about some of the relevant quality attributes.
- The uses structure, for instance, must be engineered (not merely recorded) to build a system that can be easily extended or contracted.
- The process structure is engineered to eliminate deadlock and reduce bottlenecks.
- The module decomposition structure is engineered to produce modifiable systems, and so forth.

- Each structure provides the architect with a different view into the system and a different leverage point for design.

RELATING STRUCTURES TO EACH OTHER

- Each of these structures provides a different perspective and design handle on a system, and each is valid and useful in its own right.
- Although the structures give different system perspectives, they are not independent. Elements of one will be related to elements of others, and we need to reason about these relations.
- For example, a module in a decomposition structure may be manifested as one, as part of one, or as several components in one of the component-and-connector structures, reflecting its runtime alter ego. In general, mappings between structures are many to many.
- Individual projects sometimes consider one structure dominant and cast other structures, when possible, in terms of it. Often, but not always, the dominant structure is module decomposition.
- Not all systems warrant consideration of many architectural structures.
- The larger the system, the more dramatic the differences between these structures tend to be; however, for small systems we can often get by with less.
- Instead of working with each of several component-and-connector structures, a single one will do.
- If there is only one process, then the process structure collapses to a single node and need not be carried through the design. If there is to be no distribution (that is, if there is just one processor), then the deployment structure is trivial and need not be considered further.
- Structures represent the primary engineering leverage points of an architecture

  Example



  - In module decomposition view of tiny client server system, two modules are implemented: client software and server software

- Client server view of the system shows the component and connector view of the same system
- At run time, there are ten clients running and accessing the server.
- Thus, this system has two modules, eleven components and 10 connectors.
- The correspondence between the elements in the decomposition structure and client server structure is obvious and these two views are used for very different things
- For eg, client server view could be used for performance analysis, bottle neck prediction and network traffic analysis which could be difficult or impossible with decomposition view.

5. Describe in detail about how to represent views with available notations?

Notations for documenting views differ considerably in their degree of formality. there are three main categories of notation:

1. Informal notations. Views are depicted (often graphically) using general-purpose diagramming and editing tools and visual conventions chosen for the system at hand. The semantics of the description are characterized in natural language and cannot be formally analyzed.

2. Semiformal notations. Views are expressed in a standardized notation that prescribes graphical elements and rules of construction, but does not provide a complete semantic treatment of the meaning of those elements. Rudimentary analysis can be applied to determine if a description satisfies syntactic properties. Unified Modeling Language (UML) is a semiformal notation in this sense.

3. Formal notations. Views are described in a notation that has a precise (usually mathematically based) semantics. Formal analysis of both syntax and semantics is possible. There are a variety of formal notations for software architecture available, although none of them can be said to be in widespread use.

Notations for Module Views

Informal Notations A number of notations can be used to present a module view. One common informal notation uses boxes to represent the modules, with different kinds of lines between them representing the relations. Nesting is used to depict aggregation, and arrows typically represent a depends-on relation.

for example, nesting represents aggregation, and lollipops indicate interfaces.

A second common form of informal notation is a simple textual listing of the modules with descriptions of the responsibilities.

Various textual schemes can be used to represent the ispart-of relation, such as indentation, outline numbering, and parenthetical nesting.

Other relations may be indicated by keywords. For example, the description of module A might include the line "Imports modules B, C," indicating a dependency between module A and modules B and C.

Unified Modeling Language Software modeling notations, such as UML, provide a variety of constructs that can be used to represent modules.

for example, layers, subsystems, and collections of implementation units that live together in the implementation namespace. UML was originally created to model object-oriented systems. It is now considered a general-purpose modeling language. As a result, UML elements and relations are generic; that is, they are not specific to implementation technologies or platforms. But you can define stereotypes to specialize the UML symbols.

A stereotype is a UML extension mechanism and is represented in diagrams as a label in guillemets («stereotype label»).

Dependency Structure Matrix A dependency structure matrix (DSM) is a table that shows modules as the column and row headings and dependencies as the table cells. The DSM is built as a square matrix (that is, a matrix with same number of rows and columns) where element ij is nonzero if there is a dependency between module i and module j in the architecture.

Entity-Relationship Diagram An entity-relationship diagram (ERD) is a notation specifically used for data modeling. It shows data entities that require a representation in the system and their relationships. These relationships can be one-to-one, one-to-many, or many-to-many.

6. **Write about SEI's perspectives and views with a case study?**

- Software Engineering Institute (SEI) is a federally funded research and development center

- The SEI program of work is conducted in several principal areas: acquisition, process management, risk, security, software development, and system design

- The SEI defines specific initiatives aimed at improving organizations' software engineering capabilities.

Management practices
- Organizations need to effectively manage the acquisition, development, and evolution (ADE) of software-intensive systems.
- Success in software engineering management practices helps organizations predict and control quality, schedule, cost, cycle time, and productivity.

The best-known example of SEI work in management practices is the SEI's Capability Maturity Model (CMM) for Software (now Capability Maturity Model Integration (CMMI)).

Engineering practices
- SEI work in engineering practices increases the ability of software engineers to analyze, predict, and control selected functional and non-functional properties of software systems.

Acquisition practices
- The goal of SEI work is to improve organizations acquisition processes.

Security

- The SEI is also the home of the CERT/CC (CERT Coordination Center), a federally funded computer security organization.
- The SEI works closely with defense and government organizations, industry, and academia to continually improve software-intensive systems.
- Our core purpose is to help organizations improve their software engineering capabilities and to develop or acquire the right software, defect free, within budget and on time, every time.

To accomplish this, the SEI
- performs research to explore promising solutions to software engineering problems

- identifies and codifies technological and methodological solutions

- tests and refines the solutions through pilot programs that help industry and government solve their problems

- widely disseminates proven solutions through training, licensing, and publication of best practices

The activities of the SEI can be categorized into the following technical programs:

- The Dynamic Systems Program, which conceives and develops processes for system development and offers training for software engineers

- The Product Line Systems Program, which develops programs and systems to meet specialized requirements

- The Software Engineering Process Management Program, which provides guidance to software-dependent organizations with the goal of optimizing the efficiency of their processes

- The Networked Systems Survivability Program, which assists businesses, academic institutions and government agencies in the protection of their systems against security threats and helps them to deal with problems when they occur

- The Acquisition Support Program, which helps entities improve the ways in which they obtain and upgrade their software and operating systems

The SEI is a part of Carnegie Mellon University and serves as the headquarters for CERT (the Computer Emergency Readiness Team), which conducts a public awareness campaign concerning the development, maintenance and improvement of computer and network security systems.

7. Explain in detail about allocation structure?

Allocation structures include the following.

- Deployment.
    - ➢ The deployment structure shows how software is assigned to hardware-processing and communication elements.

- The elements are software (usually a process from a component-and-connector view), hardware entities (processors), and communication pathways.
- Relations are "allocated-to," showing on which physical units the software elements reside, and "migrates-to," if the allocation is dynamic.
- This view allows an engineer to reason about performance, data integrity, availability, and security. It is of particular interest in distributed or parallel systems.

• Implementation.

- This structure shows how software elements (usually modules) are mapped to the file structure(s) in the system's development, integration, or configuration control environments.
- This is critical for the management of development activities and build processes.

• Work assignment.

- This structure assigns responsibility for implementing and integrating the modules to the appropriate development teams.
- Having a work assignment structure as part of the architecture makes it clear that the decision about who does the work has architectural as well as management implications.
- The architect will know the expertise required on each team.
- Also, on large multi-sourced distributed development projects, the work assignment structure is the means for calling out units of functional commonality and assigning them to a single team, rather than having them implemented by everyone who needs them.

8. Explain about the views and how they serve the architecture with examples?

- A system is a collection of components organized to accomplish a specific function or set of functions.
- The architecture of a system is the system's fundamental organization, embodied in its components, their relationships to each other and to the environment, and the principles guiding its design and evolution.
- An architecture description is a collection of artifacts that document an architecture. In TOGAF, architecture views are the key artifacts in an architecture description.
- Stakeholders are people who have key roles in, or concerns about, the system; for example, as users, developers, or managers.
- Different stakeholders with different roles in the system will have different concerns.
- Stakeholders can be individuals, teams, or organizations (or classes thereof).
- Concerns are the key interests that are crucially important to the stakeholders in the system, and determine the acceptability of the system.

- Concerns may pertain to any aspect of the system's functioning, development, or operation, including considerations such as performance, reliability, security, distribution, and evolvability.
- A view is a representation of a whole system from the perspective of a related set of concerns.
- In capturing or representing the design of a system architecture, the architect will typically create one or more architecture models, possibly using different tools.
- A view will comprise selected parts of one or more models, chosen so as to demonstrate to a particular stakeholder or group of stakeholders that their concerns are being adequately addressed in the design of the system architecture.
- A viewpoint defines the perspective from which a view is taken. More specifically, a viewpoint defines: how to construct and use a view (by means of an appropriate schema or template); the information that should appear in the view; the modeling techniques for expressing and analyzing the information; and a rationale for these choices

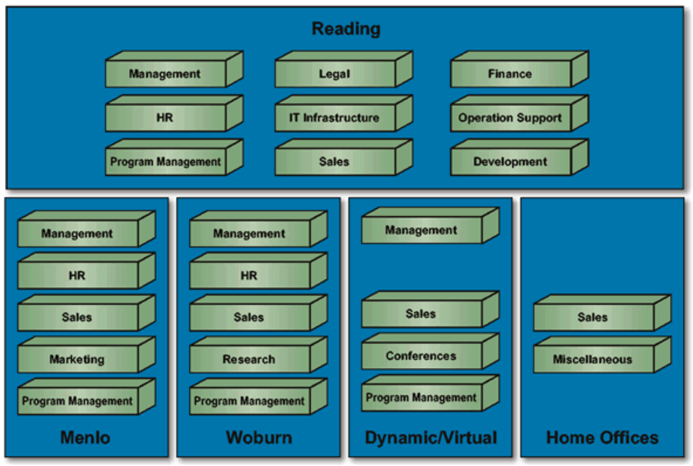
A Simple Example of a Viewpoint and View

For many architectures, a useful viewpoint is that of business domains, which can be illustrated by an example from The Open Group itself.

The viewpoint is specified as follows:

Viewpoint Element	**Description**
Stakeholders	Management Board, Chief Information Officer
Concerns	Show the top-level relationships between geographical sites and business functions.
Modeling technique	Nested boxes diagram. Blue = locations; brown = business functions. Semantics of nesting = functions performed in the locations.

The corresponding view of The Open Group (in 2001)



Developing a Business Architecture View

The Business Architecture view is concerned with addressing the concerns of users.

Stakeholder and Concerns

This view should be developed for the users. It focuses on the functional aspects of the system from the perspective of the users of the system.
Addressing the concerns of the users includes consideration of the following:

People: The human resource aspects of the system. It examines the human actors involved in the system.
Process: Deals with the user processes involved in the system.
Function: Deals with the functions required to support the processes.
Business Information: Deals with the information required to flow in support of the processes.
Usability: Considers the usability aspects of the system and its environment.
Performance: Considers the performance aspects of the system and its environment.

Modeling the View

Business scenarios are an important technique that may be used prior to, and as a key input to, the development of the Business Architecture view, to help identify and understand business needs, and thereby to derive the business requirements and constraints that the architecture development has to address.

Business scenarios are an extremely useful way to depict what should happen when planned and unplanned events occur. It is highly recommended that business scenarios be created for planned change, and for unplanned change.

Key Issues
The Business Architecture view considers the functional aspects of the system; that is, what the new system is intended to do. This can be built up from an analysis of the existing environment and of the requirements and constraints affecting the new system.

The new requirements and constraints will appear from a number of sources, possibly including:
- Existing internal specifications and lists of approved products
- Business goals and objectives
- Business process re-engineering activities
- Changes in technology

9. **Explain about module based structure?**

Module-based structures include the following.

Decomposition.

- ➢ The units are modules related to each other by the "is a submodule of " relation, showing how larger modules are decomposed into smaller ones recursively until they are small enough to be easily understood.

- Modules in this structure represent a common starting point for design, as the architect enumerates what the units of software will have to do and assigns each item to a module for subsequent (more detailed) design and eventual implementation.
- Modules often have associated products (i.e., interface specifications, code, test plans, etc.). The decomposition structure provides a large part of the system's modifiability, by ensuring that likely changes fall within the purview of at most a few small modules.
- It is often used as the basis for the development project's organization, including the structure of the documentation, and its integration and test plans.

The units in this structure often have organization-specific names.

Uses.

- The units of this important but overlooked structure are also modules, or (in circumstances where a finer grain is warranted) procedures or resources on the interfaces of modules.
- The units are related by the uses relation. One unit uses another if the correctness of the first requires the presence of a correct version (as opposed to a stub) of the second.
- The uses structure is used to engineer systems that can be easily extended to add functionality or from which useful functional subsets can be easily extracted.
- The ability to easily subset a working system allows for incremental development.

Layered.

When the uses relations in this structure are carefully controlled in a particular way, a system of layers emerges, in which a layer is a coherent set of related functionality.

In a strictly layered structure, layer n may only use the services of layer n ?

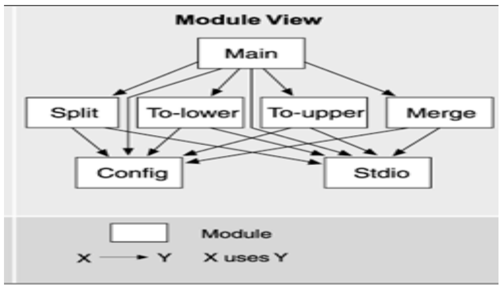
1. Many variations of this (and a lessening of this structural restriction) occur in practice, however. Layers are often designed as abstractions (virtual machines) that hide implementation specifics below from the layers above, engendering portability.
2. Class, or generalization. The module units in this structure are called classes. The relation is "inherits-from" or "is-an-instance-of." This view supports reasoning about collections of similar behavior or capability (i.e., the classes that other classes inherit from) and parameterized differences which are captured by subclassing. The class structure allows us to reason about re-use and the incremental addition of functionality.

The Module Structure: Elements & Relations

- Elements: **module**
  - **Unit of implementation** for software, representing a ***coherent entity*** from the point of view of ***functionality***
- Relations:

- **Is-part-of**: *A is part of B*
- **Depends on** (Uses or Allowed to use)
- **Is-a**: *A is a B*

- Elements properties:
  - Name: this is usually the primary information, it helps guessing the role of the element
  - Responsibilities: the role of the module should be described with more details than just its name



- The Module Viewpoint can be used for:
  - Construction blueprint
  - Analysis of certain system properties:
    - Impact analysis: predicts the impact of changes in a part of the system based on dependency relationships
    - Traceability to functional requirements: identify the modules where certain functionalities are implemented
  - Communication:
    - Understanding system functionality
    - Relationships between system parts

- The Module Viewpoint **cannot** be used for:
  - Behavior at runtime
  - Runtime qualities: performance, reliability, etc.

UNIT – IV ARCHITECTURAL STYLES

PART- A

1. What is Architectural Styles?

An architectural style is characterized by the features that make a building or other structure notable and historically identifiable. A style may include such elements as form, method of construction, building materials, and regional character. Most architecture can be classified as a chronology of styles which changes over time reflecting changing fashions, beliefs and religions, or the emergence of new ideas, technology, or materials which make new styles possible.

2. List out the common architectural styles?

- Dataflow systems: Batch sequential, Pipes and filters
- Virtual machines: Interpreters and Rule-based systems
- Call-and-return systems: Main program and subroutine, OO systems and Hierarchical layers.
- Data-centered systems: Databases, Hypertext systems and Blackboards.
- Independent components: Communicating processes

3. What do you mean by "PIPES AND FILTERS"?

Each component has set of inputs and set of outputs. A component reads streams of data on its input and produces streams of data on its output. By applying local transformation to the input streams and computing incrementally, so that output begins before input is consumed. Hence, components are termed as filters. Connectors of this style serve as conducts for the streams transmitting outputs of one filter to inputs of another. Hence, connectors are termed pipes.

4. **List out the Conditions (invariants) of this style are?**

   Filters must be independent entities. They should not share state with other filter. Filters do not know the identity of their upstream and downstream filters. Specification might restrict what appears on input pipes and the result that appears on the output pipes. Correctness of the output of a pipe-and-filter network should not depend on the order in which filter perform their processing.

5. **What Common specialization of this style includes?**
   - Pipelines: Restrict the topologies to linear sequences of filters.
   - Bounded pipes: Restrict the amount of data that can reside on pipe.
   - Typed pipes: Requires that the data passed between two filters have a well-defined type.

6. **Give a suitable example for Pipes?**

   Best known examples of pipe-and-filter architecture are programs written in UNIX-SHELL. UNIX supports this style by providing a notation for connecting components [Unix process] and by providing run-time mechanisms for implementing pipes.

7. **Give a suitable example for Filter?**

   Traditionally compilers have been viewed as pipeline systems. Stages in the pipeline include lexical analysis parsing, semantic analysis and code generation other examples of this type are. Signal processing domains, Parallel processing, Functional processing and Distributed systems.

8. **What are the advantages of Pipes and Filter?**

   They allow the designer to understand the overall input/output behavior of a system as a simple composition of the behavior of the individual filters. They support reuse: Any two filters can be hooked together if they agree on data. Systems are easy to maintain and enhance: New filters can be added to exciting systems. They permit certain kinds of specialized analysis

Eg: deadlock, throughput. They support concurrent execution.

9. **What are the disadvantages of Pipes and Filter?**

   They lead to a batch organization of processing. Filters are independent even though they process data incrementally. Not good at handling interactive applications. When incremental display updates are required. They may be hampered by having to maintain correspondences between two separate but related streams. Lowest common denominator on data transmission. This can lead to both loss of performance and to increased complexity in writing the filters.

10. **What is Object-Oriented and Data Abstraction?**

    In this approach, data representation and their associated primitive operations are encapsulated in the abstract data type (ADT) or object. The components of this style are- objects/ADT's objects interact through function and procedure invocations.

11. **What are the two important aspects of this style are?**
    - Object is responsible for preserving the integrity of its representation.
    - Representation is hidden from other objects.

12. **What are the advantages and disadvantages of Data Abstraction?**

    It is possible to change the implementation without affecting the clients because an object hides its representation from clients.

13. **What are the disadvantages of Data Abstraction?**

    To call a procedure, it must know the identity of the other object. Whenever the identity of object changes it is necessary to modify all other objects that explicitly invoke it.

14. **Define Event-Based Implicit Invocation**

    Instead of invoking the procedure directly a component can announce one or more events. Other components in the system can register an interest in an event by associating a procedure to it. When the event is announced, the system itself invokes all of the procedure that have been registered for the event. Thus an event announcement "implicitly" causes the invocation of procedures in other modules. Architecturally speaking, the components in an implicit invocation style are modules whose interface provides both a collection of procedures and a set of events.

15. Define Amdahl's Law

Speed up= Old execution time/New execution time

16. Define Layered Systems?

A layered system is organized hierarchically.Each layer provides service to the layer above it. Inner layers are hidden from all except the adjacent layers.Connectors are defined by the protocols that determine how layers interact each other.Goal is to achieve qualities of modifiability portability.

17. What are the examples of Examples layered system?

- Layered communication protocol
- Operating systems
- Database systems

18. Define REPOSITORIES: [data cantered architecture]

Goal of achieving the quality of integrability of data. In this style, there are two kinds of components.

i. Central data structure- represents current state.
ii. Collection of independent components which operate on central data store.

The choice of a control discipline leads to two major sub categories.

- Type of transactions is an input stream trigger selection of process to execute
- Current state of the central data structure is the main trigger for selecting processes to execute.

19. What are the three major parts of blackboard?

- Knowledge sources: Separate, independent parcels of application dependents knowledge.
- Blackboard data structure: Problem solving state data, organized into an application-dependent
- Hierarchy Control: Driven entirely by the state of blackboard

20. What is meant by blackboard model of problem solving?

The blackboard model of problem solving is a highly structured special case of opportunistic problem solving. In this model, the solution space is organized into several application-dependent hierarchies and the domain knowledge is partitioned into independent modules of knowledge that operate on knowledge within and between levels.

21. What do you mean by Interpreters?

An interpreter includes pseudo program being interpreted and interpretation engine. Pseudo program includes the program and activation record. Interpretation engine includes both definition of interpreter and current state of its execution.

22. List out the 4 components of Interpreter?

i.Interpretation engine: to do the work
ii.Memory: that contains pseudo code to be interpreted.
iii.Representation of control state of interpretation engine
iv. Representation of control state of the program being simulated.

23. What are the Heterogeneous Architectures?

Architectural styles can be combined in several ways: one way is through hierarchy. Example: UNIX pipeline. Second way is to combine styles is to permit a single component to use a mixture of architectural connectors. Example: "active database". Third way is to combine styles is to completely elaborate one level of architectural description in a completely different architectural style.

Example: case studies what makes a "GOOD" architecture?

24. Define Vignettes.

Vignette is a word that originally meant " **something that may be written on a vine-leaf**".In general, it is a commercial software content management system, records and documents management system, portal, and collaboration tools company.

25. What are the Three Vignettes In Mixed Style?

- Level 1: Process measurement and control: direct adjustment of final control elements.
- Level 2: Process supervision: operations console for monitoring and controlling Level 1.

- Level 3: Process management: computer-based plant automation, including management reports, optimization strategies, and guidance to operations console.
- Levels 4 and 5: Plant and corporate management: higher-level functions such as cost accounting, inventory control, and order processing/scheduling.

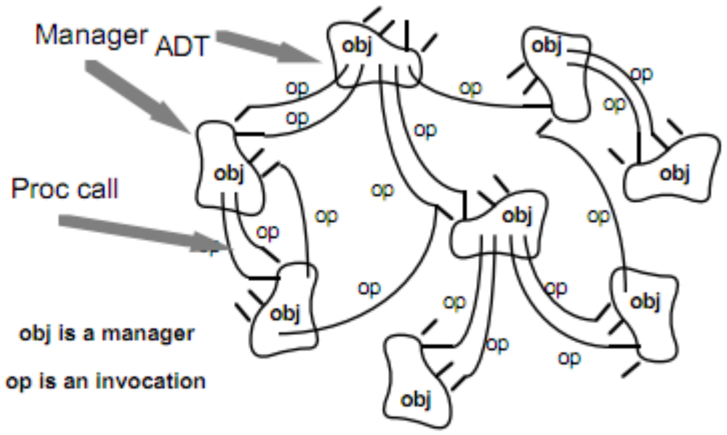
PART- B

1. Explain the architecture styles based on:

i. Data abstraction and object-oriented organization

- In this style data representations and their associated primitive operations are encapsulated in an abstract data type or object.
- The components of this style are the objects—or, if you will, instances of the abstract data types.
- Objects are examples of a sort of component we call a manager because it is responsible for preserving the integrity of a resource (here the representation).
- Objects interact through function and procedure invocations.
- Two important aspects of this style are

  (a) that an object is responsible for preserving the integrity of its representation (usually by maintaining some invariant over it), and
  (b) that the representation is hidden from other objects.



- The use of abstract data types, and increasingly the use of object-oriented systems, is, of course, widespread.
- Object-oriented systems have many nice properties, most of which are well known.
- Because an object hides its representation from its clients, it is possible to change the implementation without affecting those clients.
- Additionally, the bundling of a set of accessing routines with the data they manipulate allows designers to decompose problems into collections of interacting agents.
- But object-oriented systems also have some disadvantages. The most significant is that in order for one object to interact with another (via procedure call) it must know the identity of that other

- object.

- This is in contrast, for example, to pipe and filter systems, where filters do need not know what other filters are in the system in order to interact with them.

- The significance of this is that whenever the identity of an object changes it is necessary to modify all other objects that explicitly invoke it.

- In a module oriented language this manifests itself as the need to change the "import" list of every module that uses the changed module.

ii) Event-based, implicit invocation

- Traditionally, in a system in which the component interfaces provide a collection of procedures and functions, components interact with each other by explicitly invoking those routines.

- However, recently there has been considerable interest in an alternative integration technique, variously referred to as implicit invocation, reactive integration, and selective broadcast.

- This style has historical roots in systems based on actors, constraint satisfaction, daemons, and packet-switched networks.

- The idea behind implicit invocation is that instead of invoking a procedure directly, a component can announce (or broadcast) one or more events.
- Other components in the system can register an interest in an event by associating a procedure with the event.

- When the event is announced the system itself invokes all of the procedures that have been registered for the event.

- Thus an event announcement ``implicitly" causes the invocation of procedures in other modules.

- Architecturally speaking, the components in an implicit invocation style are modules whose interfaces provide both a collection of procedures (as with abstract data types) and a set of events.
- Procedures may be called in the usual way.

- But in addition, a component can register some of its procedures with events of the system.

- This will cause these procedures to be invoked when those events are announced at run time.

- Thus the connectors in an implicit invocation system include traditional procedure call as well as bindings between event announcements and procedure calls.

- The main invariant of this style is that announcers of events do not know which components will be affected by those events.
- Thus components cannot make assumptions about order of processing, or even about what processing, will occur as a result of their events.
- For this reason, most implicit invocation systems also include explicit invocation (i.e., normal procedure call) as a complementary form of interaction.
- One important benefit of implicit invocation is that it provides strong support for reuse. Any component can be introduced into a system simply by registering it for the events of that system.
- A second benefit is that implicit invocation eases system evolution.
- Components may be replaced by other components without affecting the interfaces of other components in the system.
- In contrast, in a system based on explicit invocation, whenever the identity of a that provides some system function is changed, all other modules that import that module must also be changed.
- The primary disadvantage of implicit invocation is that components relinquish control over the computation performed by the system.
- When a component announces an event, it has no idea what other components will respond to it.
- Another problem concerns exchange of data. Sometimes data can be passed with the event. But
- in other situations event systems must rely on a shared repository for interaction.
- Finally, reasoning about correctness can be problematic, since the meaning of a procedure that announces events will depend on the context of bindings in which it is invoked.
- This is in contrast to traditional reasoning about procedure calls, which need only consider a procedure's pre- and post-conditions when reasoning about an invocation of it

2. Explain architectural style. Mention any four commonly used styles.

- Software architects use a number of commonly recognized ìstylesî to develop the architecture of a system.
- Architectural style implies a set of design rules that identify the kinds of components and connectors that may be used to compose a system or subsystem, together with local or global constraints that are implemented.

- Components, including encapsulated subsystems, may be distinguished by the nature of their computation (e.g., whether they retain state from one invocation to another, and if so, whether that state is available to other components).

- Component types may also be distinguished by their packagingóthe ways they interact with other components.

- Packaging is usually implicit, which tends to hide important properties of the components. To clarify the abstractions we isolate the definitions of these interaction protocols in connectors (e.g., processes interact via message-passing protocols; unix filters interact via data flow through pipes).

- The connectors play a fundamental role in distinguishing one architectural style from another and have an important effect on the characteristics of a particular style .

Pipes and Filters

- The style of a specific system is usually established by appeal to common knowledge or intuition. Architectures have traditionally been expressed in box-and-line diagrams and informal prose, so the styles provide drawing conventions, vocabulary, and informal constraints Pipes and Filters

- In a pipe and filter style each component has a set of inputs and a set of outputs.

- A component reads streams of data on its inputs and produces streams of data on its outputs, delivering a complete instance of the result in a standard order.

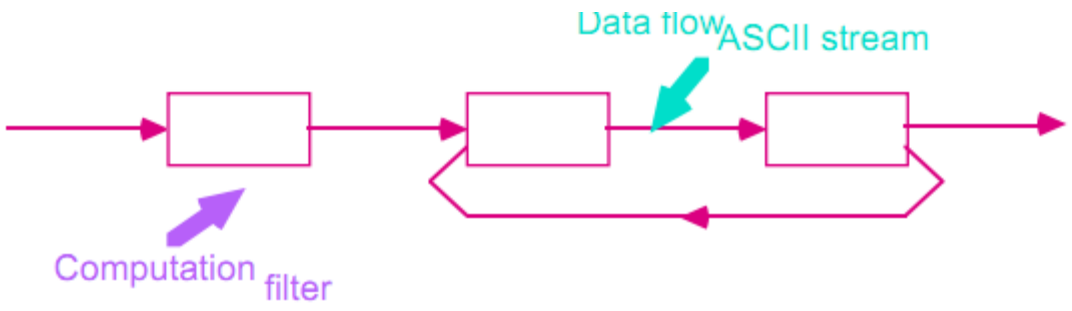
- This is usually accomplished by applying a local transformation to the input streams and computing incrementally so output begins before input is consumed.

- Hence components are termed “filters”. The connectors of this style serve as conduits for the streams, transmitting outputs of one filter to inputs of another. Hence the connectors are termed “pipes”.

- Among the important invariants of the style, filters must be independent entities: in particular, they should not share state with other filters.

- Another important invariant is that filters do not know the identity of their upstream and downstream filters.

- Their specifications might restrict what appears on the input pipes or make guarantees about what appears on the output pipes, but they may not identify the components at the ends of those pipes



Data abstraction and object-oriented organization

- In this style data representations and their associated primitive operations are encapsulated in an abstract data type or object.

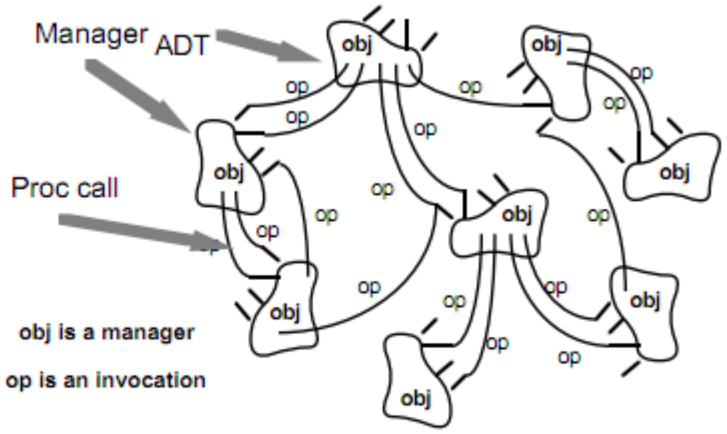
- The components of this style are the objects—or, if you will, instances of the abstract data types.

- Objects are examples of a sort of component we call a manager because it is responsible for preserving the integrity of a resource (here the representation).

- Objects interact through function and procedure invocations.

- Two important aspects of this style are

  (a) that an object is responsible for preserving the integrity of its representation (usually by maintaining some invariant over it), and
  (b) that the representation is hidden from other objects.



- The use of abstract data types, and increasingly the use of object-oriented systems, is, of course, widespread.

- Object-oriented systems have many nice properties, most of which are well known.

- Because an object hides its representation from its clients, it is possible to change the implementation without affecting those clients.

- Additionally, the bundling of a set of accessing routines with the data they manipulate allows designers to decompose problems into collections of interacting agents.

- But object-oriented systems also have some disadvantages. The most significant is that in order for one object to interact with another (via procedure call) it must know the identity of that other
- object.

- This is in contrast, for example, to pipe and filter systems, where filters do need not know what other filters are in the system in order to interact with them.

- The significance of this is that whenever the identity of an object changes it is necessary to modify all other objects that explicitly invoke it.

- In a module oriented language this manifests itself as the need to change the "import" list of every module that uses the changed module.

Event-based, implicit invocation

- Traditionally, in a system in which the component interfaces provide a collection of procedures and functions, components interact with each other by explicitly invoking those routines.

- However, recently there has been considerable interest in an alternative integration technique, variously referred to as implicit invocation, reactive integration, and selective broadcast.

- This style has historical roots in systems based on actors, constraint satisfaction, daemons, and packet-switched networks.

- The idea behind implicit invocation is that instead of invoking a procedure directly, a component can announce (or broadcast) one or more events.
- Other components in the system can register an interest in an event by associating a procedure with the event.

- When the event is announced the system itself invokes all of the procedures that have been registered for the event.

- Thus an event announcement ``implicitly" causes the invocation of procedures in other modules.

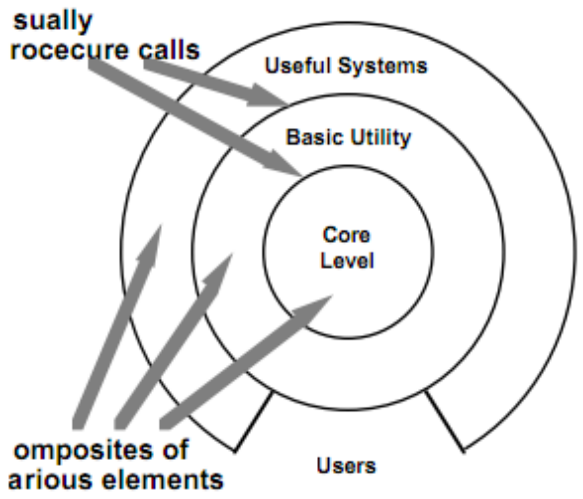
- Architecturally speaking, the components in an implicit invocation style are modules whose interfaces provide both a collection of procedures (as with abstract data types) and a set of events.
- Procedures may be called in the usual way.

- But in addition, a component can register some of its procedures with events of the system.

- This will cause these procedures to be invoked when those events are announced at run time.
- Thus the connectors in an implicit invocation system include traditional procedure call as well as bindings between event announcements and procedure calls.
- The main invariant of this style is that announcers of events do not know which components will be affected by those events.
- Thus components cannot make assumptions about order of processing, or even about what processing, will occur as a result of their events.

Layered Systems

- A layered system is organized hierarchically, each layer providing service to the layer above it and serving as a client to the layer below.
- In some layered systems inner layers are hidden from all except the adjacent outer layer, except for certain functions carefully selected for export.
- Thus in these systems the components implement a virtual machine at some layer in the hierarchy. (In other layered systems the layers may be only partially opaque.)
- The connectors are defined by the protocols that determine how the layers will interact.
- Topological constraints include limiting interactions to adjacent layers.



3. Consider the case study of building a software controlled mobile robot. Describe its challenging problems and design considerations with four requirements. Finally give the solution by layered architecture for all the four requirements.

Controls manned/partially manned vehicles Space exploration, hazardous waste disposal, underwater exploration

The software must deal with:

- External sensors and actuators
- Real-time responsiveness
- Acquire sensor I/P, control motion and plan

Future paths

- Many issues from imperfect inputs to unexpected/unpredictable obstacles, and events

Mobile Robots

Four basic requirements

Accommodate deliberate and reactive behavior
• Coordinate actions it must undertake to achieve its designated objective (collect rock sample, avoid obstacles)
Allow for uncertainty
• Framework for actions even when faced with incomplete or unreliable information (contradictory sensor readings)
Account for dangers
• Must be fault tolerant, safe and with high performance (e.g., cope with reduced power, dangerous vapors, etc.)
Give design flexibility
• Development requires frequent experimentation and reconfiguration

Four Architectural Designs

- Lozano's Control Loops
- Elfes's Layered Organization
- Simmons's Task Control Architecture
- Shafer's Application of Blackboards

Solution 1: Control Loop

- Industrial robots need only handle minimally unpredictable events
- Tasks are fully defined (no need for a planer) and has no responsibility wrt its environment

• Open loop paradigm applies
• Robot initiates actions without caring about
Consequences
Lets add feedback for closed loop
• Robot adjusts the future plans based on monitored information

Requirements Trade-Off Analysis
Req1– Advantage: simplicity

- Simplicity is a drawback in more unpredictable environments
- Robots mostly confronted with disparate discrete events that require them to switch between very different behavior modes
- For complex tasks, gives no leverage for decomposition into cooperating components

Req2– Advantage: reducing unknowns through iteration

- Is biased toward one method (only).
- Trial and error process of action-reaction to eliminate possibilities at each turn.
- No framework for integrating these with the basic loop or for delegating them to separate entities.

Req3– Advantage: supports fault tolerance and safety

- Simplicity makes duplication easy

• Reduces the chance of errors creeping into the system

Req4 – Advantage: clearly partition-able into supervisor, sensors and motors that are independent and replaceable

- More refined tuning is however not really supported (inside the modules)
- Conclusion: Most appropriate for simple robotic systems

Solution 2: Layered Architecture

Influenced the sonar and navigational systems design used on the Terregator and Neptune mobile Robots
Level 1 (core) control routines (motors, joints,..),
Level 2-3 real world I/P (sensor interpretation and integration (analysis of combined I/Ps)
Level 4 maintains the real world model for robot
Level 5 manage navigation
Level 6-7 Schedule & plan robot actions (including exception handling and replanning)

Top level deals with UI and overall superviosory functions

Requirements Trade-Off Analysis
Req1

- Avoids some problems encountered in the control loop style by defining more components to delegate tasks.
- Defines abstraction levels (robot control versus navigation) to guide the design
- Does not however fit the actual data / control flow patterns!!
- Information exchange is less straightforward because the layers suggest that services and requests be
  passed between layers

• Fast reaction times drives the need to bypass layers to go directly to the problem-handling agent at level 7 skip layers to improve response time!
Two separate abstractions are needed that are not supported
• Data hierarchy
• Control hierarchy
Req2 Abstraction layers address the need to manage uncertainty
What is uncertain at the lower layers may become clear with added knowledge available from the higher layers For Example
• The context embodied in the world model can provide the clues to disambiguate conflicting sensor data
Req3 Fault tolerance and passive safety (strive not to do something)

- Thumbs up data and commands are analyzed from different perspectives
- Possible to incorporate many checks and balances
- Performance and active safety may require that layers be short circuited

Req4 Flexibility in replacement and addition of components

- Interlayer dependencies are an obstacle
- Complex relationships between layers can become more difficult to decipher with each change
- Success because the layers provide precision in defining the roles of each layer

Solution 3: Implicit Invocation
Basis and Specifics

- Based on various hierarchies of tasks
- Utilizes dynamic task trees
- Run-time configurable

- Permits selective concurrency

Supports 3 different functions

Exceptions

- Suited to handle spontaneous events
- Manipulate task trees

Wiretapping

- Messages intercepted by tasks superimposed on a task tree

Monitors

- Read info and execute some actions if data fulfills a criterion

Req1 Advantage: clear cut separation of action

- Explicit incorporation of concurrent agents in its model

Req2 Disadvantage: uncertainty not well addressed

- Task tree could be built by exception handler

Req3 Advantage: accounts for performance, safety, & fault tolerances

- Redundant fault handlers
- Multiple requests handled concurrently

Req4 advantage: Incremental development & replacement straightforward

- Possible to use wiretaps, monitors, or new handlers without affecting existing components
- Based on CODGER system used in NAVLAB project (known as whiteboard arch)
- Relies on abstractions similar to those found in the layered architecture example
- Utilizes a shared repository for communication between components

Blackboard Arch.

- Based on CODGER system used in NAVLAB project (known as whiteboard arch)
- Relies on abstractions similar to those found in the layered architecture example
- Utilizes a shared repository for communication between components
- Components register interest in certain types of data
- This info is returned immediately or when it is inserted onto blackboard by some other module

Components of CODGER architecture are:

Captain: overall supervisor

Map navigator: high-level path planner

Lookout: monitors environment for landmarks

Pilot: low-level path planner and motor controller

Perception subsystems: accept sensor input and integrate it into a coherent situation interpretation

Req1 Deliberative and reactive

- Components register for the type of information they are interested in and receive it as it becomes available
- This shared communication mechanism supports both deliberative and reactive behavior requirements
- However, the control flow must be worked around the database mechanism; rather than communication between components

Req2 Allow for uncertainty

- provides means for resolving conflicts or uncertainties as all data is in database (from all components)
- modules responsible for resolution simply register for required data and process it accordingly

4. Expalin the following with an example:

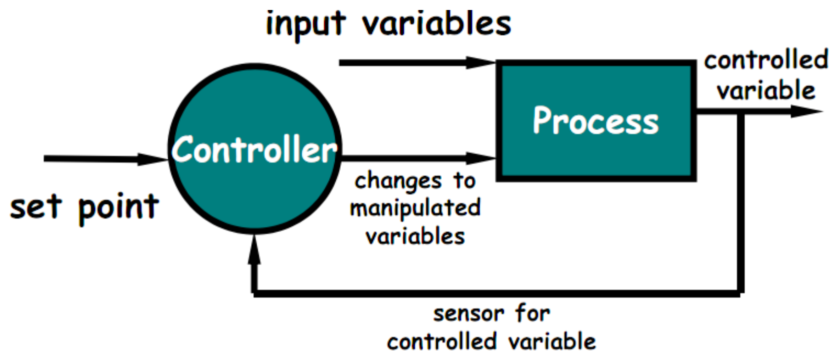
i. **Controlled variable**
ii. **Set point**
iii. **Open loop system**
iv. **Feedback control system**
v. **Feed forward control system**

Connectors: are the data flow relations for:

- Process Variables: –Controlled variable whose value the system is intended to control.
- Input variable that measures an input to the process.
- Manipulated variable whose value can be changed by the controller.
- Set Point is the desired value for a controlled variable.
- Sensors to obtain values of process variables pertinent to control

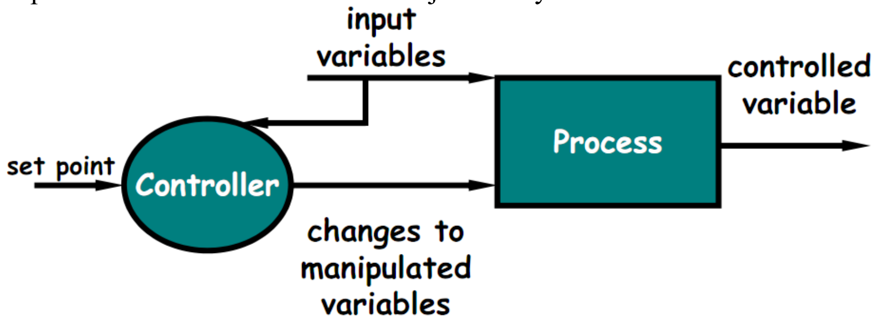
Feed-Back Control System

The controlled variable is measured and the result is used to manipulate one or more of the process variables.


input variables
controlled variable
Process
Controller
set point
changes to manipulated variables
sensor for controlled variable

Open-Loop Control System

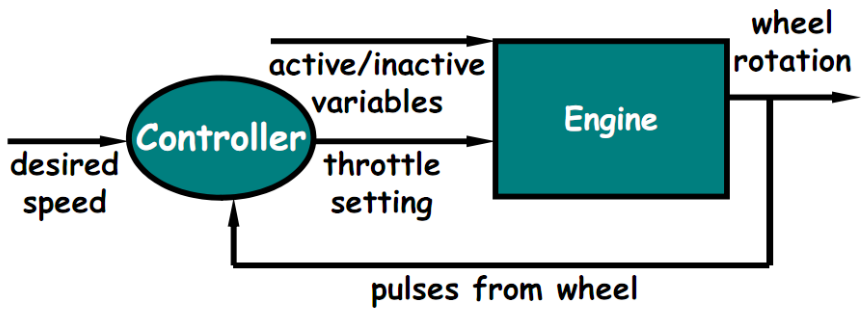
Information about process variables is not used to adjust the system.



Process Control Examples

Real-Time System Software to Control:

- Automobile Anti-Lock Brakes
- NuclearPower Plants
- Automobile Cruise-Control



- The first task is to model the current speed from the wheel pulses; the designer should validate this model carefully.

- The model could fail if the wheels spin; this could affect control in two ways.

- If the wheel pulses are being taken from a drive wheel and the wheel is spinning, the cruise control would keep the wheel spinning (at constant speed) even if the vehicle stops moving.

- The controller is implemented as a continuously-evaluating function that matches the dataflow character of the inputs and outputs.

- Several implementations are possible, including variations on simple on/off control, proportional control, and more sophisticated disciplines.

- The set point calculation divides naturally into two parts:

  (a) determining whether or not the automatic system is active—in control of the throttle and

  (b) determining the desired speed for use by the controller in automatic mode.

5. State the problem of KWIC. Propose implicit invocation and pipes and filters style to implement a solution for the same

- The KWIC [Key Word in Context] index system accepts an ordered set of lines, each line is an ordered set of words, and each word is an ordered set of characters.

- Any line may be ``circularly shifted" by repeatedly removing the first word and appending it at the end of the line.

- The KWIC index system outputs a listing of all circular shifts of all lines in alphabetical order.

- From the point of view of software architecture, the problem derives its appeal from the fact that it can be used to illustrate the effect of changes on software design.

- Parnas shows that different problem decompositions vary greatly in their ability to withstand design changes.

Among the changes he considers are:

• Changes in processing algorithm: For example, line shifting can be performed on each line as it is read from the input device, on all the lines after they are read, or on demand when the alphabetization requires a new set of shifted lines.

• Changes in data representation: For example, lines can be stored in various ways. Similarly, circular shifts can be stored explicitly or implicitly (as pairs of index and offset).

• Enhancement to system function: For example, modify the system so that shifted lines to eliminate circular shifts that start with certain noise words (such as “a”, “an”, “and”, etc.).

• Performance: Both space and time.

• Reuse: To what extent can the components serve as reusable entities.

Solution 1:

- Main Program/Subroutine with Shared Data The first solution decomposes the problem according to the four basic functions performed: input, shift, alphabetize, and output.

- These computational components are coordinated as subroutines by a main program that sequences through them in turn.

- Data is communicated between the components through shared storage (“core storage”). Communication between the computational components and the shared data is an unconstrained read-
- write protocol.

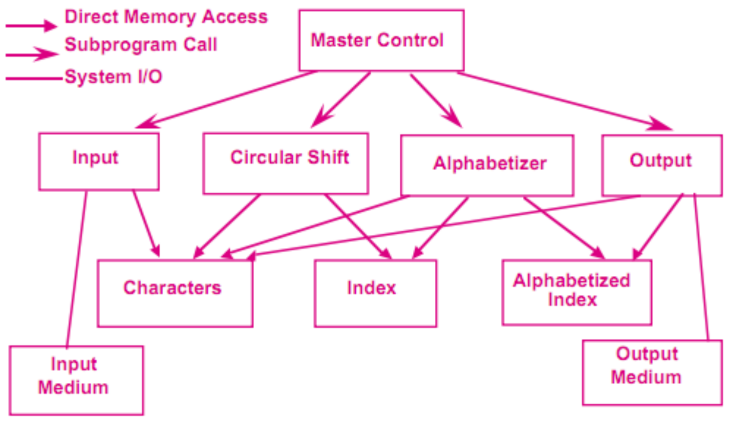
- This is made possible by the fact that the coordinating program guarantees sequential access to the data.

- Using this solution data can be represented efficiently, since computations can share the same storage.

- The solution also has a certain intuitive appeal, since distinct computational aspects are isolated in different modules.

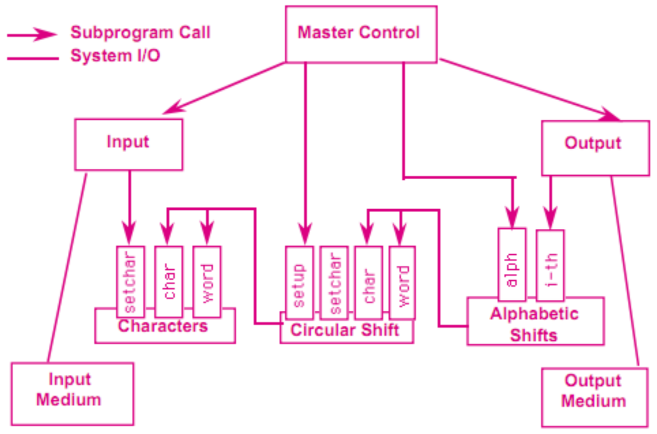
- However, as Parnas argues, it has a number of serious drawbacks in terms of its ability to handle changes. In particular, a change in data storage format will affect almost all of the modules. Similarly changes in the overall processing algorithm and enhancements to system function are not easily accomodated.

- Finally, this decom-position is not particularly supportive of reuse.



Solution 2: Abstract Data Types

- The second solution decomposes the system into a similar set of five modules.
- However, in this case data is no longer directly shared by the computational components.
- Instead, each module provides an interface that permits other components to access data only by invoking procedures in that interface.
- This solution provides the same logical decomposition into processing modules as the first. However, it has a number of advantages over the first solution when design changes are considered.
- In particular, both algorithms and data representations can be changed in individual modules without
affecting others.
- Moreover, reuse is better supported than in the first solution because modules make fewer assumptions about the others with which they interact.



- On the other hand, as discussed by Garlan, Kaiser, and Notkin, the solution is not particularly well-suited to enhancements.

- The main problem is that to add new functions to the system, the implementor must either modify the existing modules—compromising their simplicity and integrity—or add new modules that lead to performance penalties.

Solution 3: Implicit Invocation

- The third solution uses a form of component integration based on shared data similar to the first solution. However, there are two important differences.

- First, the interface to the data is more abstract. Rather than exposing the storage formats to the computing modules, data is accessed abstractly.

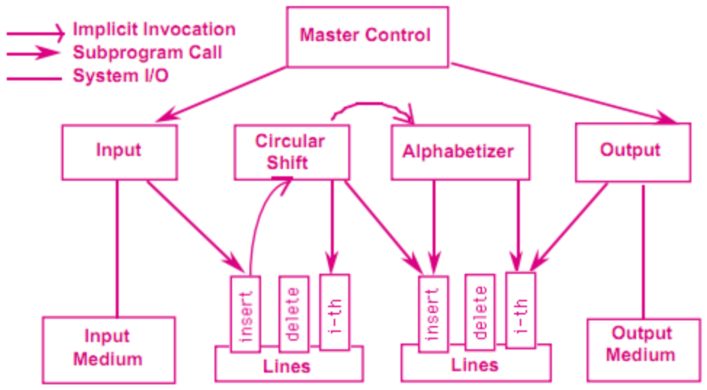
- Second, computations are invoked implicitly as data is modified. Thus interaction is based on an active data model. This in turn causes the alphabetizer to be implicitly invoked so that it can alphabetize the lines.

- This solution easily supports functional enhancements to the system: additional modules can be attached to the system by registering them to be invoked on data-changing events. Because data is accessed abstractly, it also insulates computations from changes in data representation.

- Reuse is also supported, since the implicitly invoked modules only rely on the existence of certain externally triggered events

- However, the solution suffers from the fact that it can be difficult to control the order of processing of the implicitly invoked modules.

- Further, because invocations are data driven, the most natural implementations of this kind of decomposition tend to use more space than the previously considered decompositions.



Solution 4: Pipes and Filters

The fourth solution uses a pipeline solution.

In this case there are four filters: input, shift, alphabetize, and output. Each filter processes the data and sends it to the next filter.

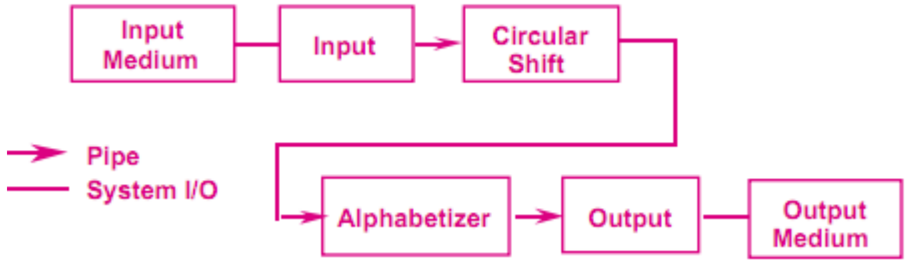
Control is distributed: each filter can run whenever it has data on which to compute. Data sharing between filters is strictly limited to that transmitted on pipes.

This solution has several nice properties.

First, it maintains the intuitive flow of processing.

Second, it supports reuse, since each filter can function in isolation (provided upstream filters produce data in the form it expects). New functions are easily added to the system by inserting filters at the appropriate point in the processing sequence.

Third, it supports ease of modification, since filters are logically independent of other filters.



On the other hand it has a number of drawbacks. First, it is virtually impossible to modify the design to support an interactive system. For example, in order to delete a line, there would have to be some persistent shared storage, violating a basic tenet of this approach.

Second, the solution is inefficient in terms of its use of space, since each filter must copy all of the data to its output ports.

6. Discuss the importance and advantages of the architectural styles with reference to an appropriate application area.

The Layered System Style

- Components
  - Programs or subprograms
- Connectors
  - Procedure calls or system calls
- Configurations
  - "Onion" or "stovepipe" structure, possibly replicated
- Underlying computational model
  - Procedure call/return
- Stylistic invariants
  - Each layer provides a service only to the immediate layer "above" (at the next higher level of abstraction) and uses the service only of the immediate layer "below" (at the next lower level of abstraction)
- Advantages
  - Decomposability: Effective separation of concerns
  - Maintainability: Changes that do not affect layer interfaces are easy to make
  - Evolvability: Potential for adding layers
  - Adaptability/Portability: Can replace inner layers as long as interfaces remain the same (consider swapping out a Solaris JVM for a Linux one)
  - Understandability: Strict set of dependencies allow you to ignore outer layers
- Disadvantages

- Performance degrades with too many layers
- Can be difficult to cleanly assign functionality to the "right" layer

The Blackboard Style

- Components
  - Blackboard client programs
- Connector
  - Blackboard: shared data repository, possibly with finite capacity
- Configurations
  - Multiple clients sharing single blackboard
- Underlying computational model
  - Synchronized, shared data transactions, with control driven entirely by blackboard state
- Stylistic invariants
  - All clients see all transactions in the same order
- Advantages
  - Simplicity: Only one connector (the blackboard) that everyone uses
  - Evolvability: New types of components can be added easily
  - Reliability(?): Concurrency controls of information, traditionally a tricky problem, can be largely addressed in the blackboard
- Disadvantages
  - Blackboard becomes a bottleneck with too many clients
  - Implicit "partitions" of information on the blackboard may cause confusion, reduce understandability

Event-Based Systems and the Implicit Invocation Style

- Components
  - Programs or program entities that *announce* and/or *register interest in* events
- Connectors
  - Event broadcast and registration infrastructure
- Configurations
  - Implicit dependencies arising from event announcements and registrations
- Underlying computational model

1. Event announcement is broadcast
2. Procedures associated with registrations (if any) are invoked

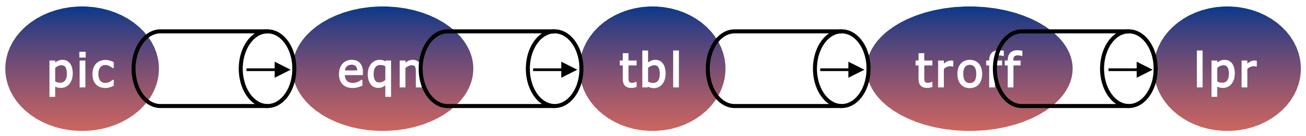
- Implicit Invocation Advantages & Disadvantages Advantages
  - Reusability: Components can be put in almost any context
  - Distributability: Events are independent and can travel across the network
  - Interoperability: Components may be very heterogeneous
  - Visibility: Events are a reified form of communication that can be logged and viewed
  - Robustness: Components in this style generally have to be written to tolerate failure or unexpected circumstances well
- Disadvantages
  - Reliability: Components announcing events have no guarantee of getting a response
  - Simplicity: Components announcing events have no control over the order of responses, so they must be robust enough to handle this
  - Understandability: Difficult to reason about the behavior of an announcing component independently of the components that register for its events
  - Event abstraction does not cleanly lend itself to data exchange

7. Discuss the invariants, advantages and disadvantages of pipes and filters architectural style

- Components
  - Individual programs transforming input data to output data
- Connectors
  - Unidirectional or bidirectional data streams
- Configurations
  - Parallel linear composition of program invocations
- Underlying computational model
  - Sequential data flow and transformation
- Stylistic invariants
  - Every component has one input predecessor and one output successor
- Common specializations
  - **Pipelines:** *single* linear composition of pipes and filters
  - Bounded pipes, typed pipes

- Advantages
  - Simplicity: Simple, intuitive, efficient composition of components
  - Reusability: High potential for reuse of components
  - Evolvability: Changing architectures is trivial
  - Efficiency: Limited amount of concurrency (contrast batch-sequential)
  - Consistency: All components have the same interfaces, only one type of connector
  - Distributability: Byte streams can be sent across networks
- Disadvantages
  - Batch-oriented processing
  - Must agree on lowest-common-denominator data format
  - Does not guarantee semantics
  - Limited application domain: stateless data transformation

Pipe-and-Filter Example:
UNIX Text Processing



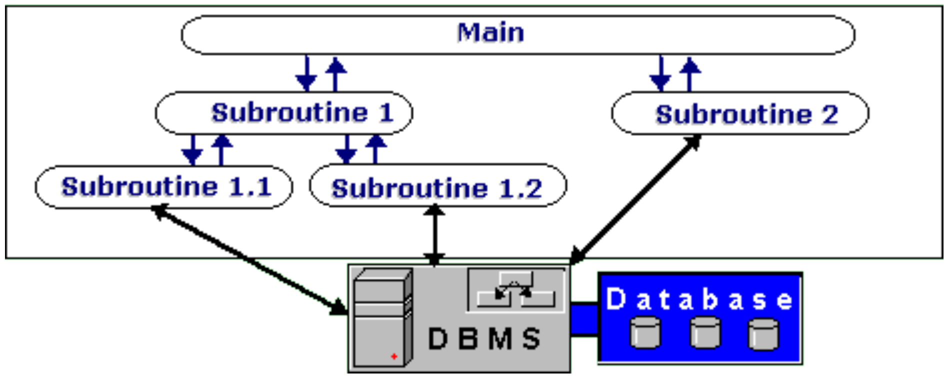
8. Write a note on heterogeneous architectures

- 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.

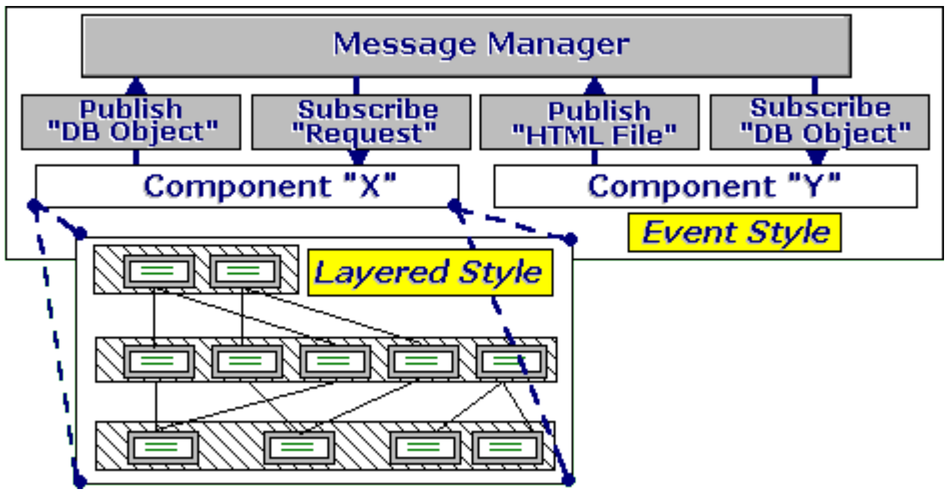
- ***Locationally heterogeneous*** means that a drawing of its runtime structures will reveal patterns of different styles in different areas.

- 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.

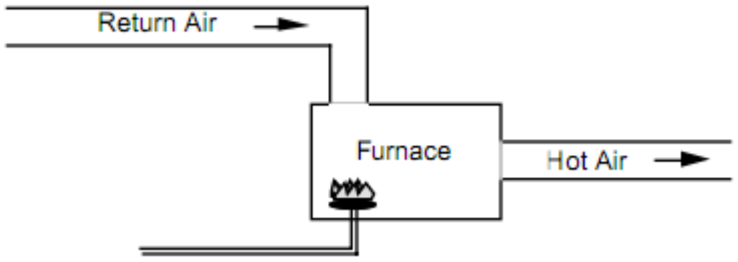
- This last form of heterogeneity recognizes that styles do not partition software architectures into non-overlapping, clean categories.
- The data-centered style at the beginning of this discussion was composed of thread-independent clients, much like an independent component architecture.
- The layers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

9. Explain the process control paradigm with various process control definitions

- This system organization is not widely recognized in the software community; nevertheless it seems to quietly appear within designs dominated by other models.
- Unlike object-oriented or functional design, which are characterized by the kinds of components that appear, control loop designs are characterized both by the k inds of components and the special relations that must hold among the components.
- Process control models and derives a software paradigm for control loop organizations.

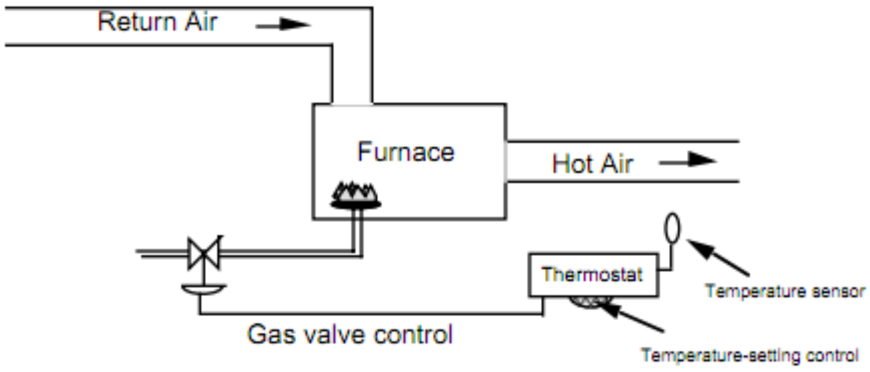
Process control paradigms

- Continuous processes of many kinds convert input materials to products with specific properties by performing operations on the inputs and on intermediate products.
- The values of measurable properties of system state (materials, equipment settings, etc.) are called the variables of the process.
- Process variables that measure the output materials are called the controlled variables of the process.
- The properties of the input materials, intermediate products, and operations are captured in other process variables.
- In particular, the manipulated variables are associated with things that can be changed by the control system in order to regulate the process.
- Process variables must not be confused with program variables; this error can lead to disaster
- The purpose of a control system is to maintain specified properties of the outputs of the process at (sufficiently near) given reference values called the set points.
- If the input materials are pure, if the process is fully-defined, and if the operations are completely repeatable, the process can simply run without surveillance. Such a process is called an open loop system.
- A hot-air furnace that uses a constant burner setting to raise the temperature of the air that passes through.
- A similar furnace that uses a timer to turn the burner off and on at fixed intervals is also an open loop system.



- The open-loop assumptions are rarely valid for physical processes in the real world.
- More often, properties such as temperature, pressure and flow rates are monitored, and their values are used to control the process by changing the settings of apparatus such as valves, heaters, and chillers. Such systems are called closed loop systems.

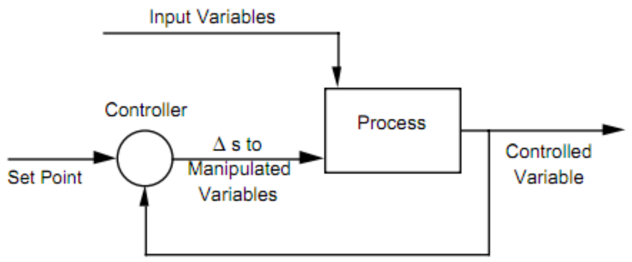
- A home thermostat is a common example: the air temperature at the thermostat is measured, and the furnace is turned on and off as necessary to maintain the desired temperature (the set point).



- There are two general forms of closed loop control. Feedback control, illustrated in Figure 3, adjusts the process based on measurements of the controlled variable.

- The important components of a feedback controller are the process definition, the process variables (including designated input and controlled variables), a sensor to obtain the controlled variable from the physical output, the set point (target value for the controlled variable), and a control
- algorithm.

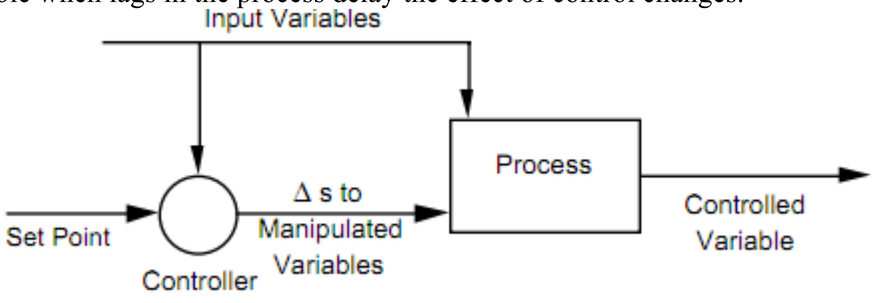
- The furnace with burner is the process; the thermostat is the controller; the return air temperature is the input variable; the hot air temperature is the controlled variable; the thermostat setting is the set point; and the temperature sensor is the sensor



- Feedforward control, anticipates future effects on the controlled variable by measuring other process variables whose values may be more timely; it adjusts the process based on these variables.

- The important components of a feedforward controller are essentially the same as for a feedback controller except that the sensor(s) obtain values of input or intermediate variables.

- It is valuable when lags in the process delay the effect of control changes.



Process Control Definitions

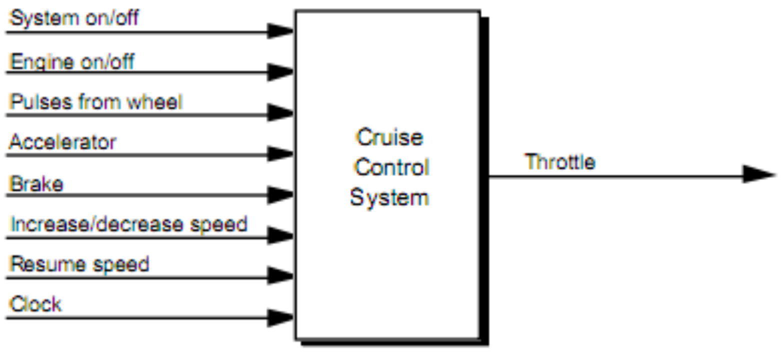
- Process variables: properties of the process that can be measured; several specific kinds are often distinguished. Do not confuse process variables with program variables.

- Controlled variable: process variable whose value the system is intended to control
- Input variable: process variable that measures an input to the process
- Manipulated variable: process variable whose value can be changed by the controller
- Set point: the desired value for a controlled variable
- Open loop system: system in which information about process variables is not used to adjust the system.
- Closed loop system: system in which information about process variables is used to manipulate a process variable to compensate for variations in process variables and operating conditions.
- Feedback control system: the controlled variable is measured and the result is used to manipulate one or more of the process variables
- Feedforward control system: some of the process variables are measured and disturbances are compensated for without waiting for changes in the controlled variable to be visible.

10. Explain the process control view of cruise control architectural style and obtain the complete cruise
control system

A cruise control system exists to maintain the speed of a car, even over varying terrain.

The block diagram of the hardware for such a system.



There are several inputs:

• System on/off If on, denotes that the cruise-control system should maintain the car speed.
• Engine on/off If on, denotes that the car engine is turned on; the cruise-control system is only active if the engine is on.
• Pulses from wheel A pulse is sent for every revolution of the wheel.
• Accelerator Indication of how far the accelerator has been pressed.
• Brake On when the brake is pressed; the cruise-control system temporarily reverts to manual control if the brake is pressed.
• Increase/Decrease Speed Increase or decrease the maintained speed; only applicable if the cruise-control system is on.
• Resume: Resume the last maintained speed; only applicable if the cruise control system is on.
• Clock Timing pulse every millisecond.

There is one output from the system:

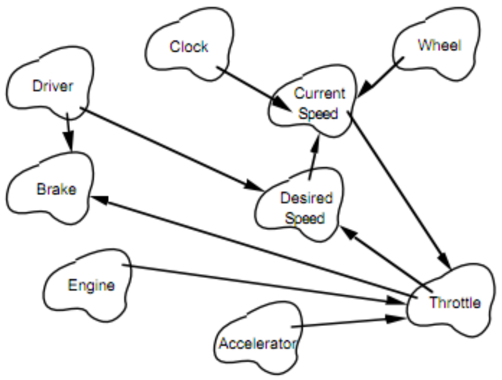
• Throttle Digital value for the engine throttle setting.

- The problem does not clearly state the rules for deriving the output from the set of inputs.
- Booch provides a certain amount of elaboration in the form of a data flow diagram, but some questions remain unanswered.
- In the design below, missing details are supplied to match the apparent behavior of the cruise control on the author's car.
- Moreover, the inputs provide two kinds of information: whether the cruise control is active, and if so what speed it should maintain.
- The problem statement says the output is a value for the engine throttle setting.
- In classical process control the corresponding signal would be a change in the throttle setting; this avoids calibration and wear problems with the sensors and engine.
- A more conventional cruise control requirement would thus specify control of the current speed of the vehicle.
- However, current speed is not explicit in the problem statement, though it does appear implicitly as “maintained speed” in the descriptions of some of the inputs.

- If the requirement addresses current speed, throttle setting remains an appropriate output from the control algorithm.
- To avoid unnecessary changes in the problem we assume accurately calibrated digital control and achieve the effect of incremental signals by retaining the previous throttle value in the controller.
- The problem statement also specifies a millisecond c lock.
- In the object-oriented solution, the clock is used only in combination with the wheel pulses to determine the current speed.
- Presumably the process that computes the speed will count the number of clock pulses between wheel pulses.
- A typical automobile tire has a circumference of about 6 feet, so at 60 mph (88 ft/sec) there will be about 15 wheel pulses per second.
- The problem is over specified in this respect: a slower clock or one that delivered current time on demand with sufficient precision would also work and would require less computing.
- Further, a single system clock is not required by the problem, though it might be convenient for other reasons.
- These considerations lead to a restatement of the problem: Whenever the system is active, determine the desired speed and control the engine throttle setting to maintain that speed.

Object view of cruise control

- Booch structures an object-oriented decomposition of the system around objects that exist in the task description.
- This yields a decomposition whose elements correspond to important quantities and physical entities in the system.



Process control view of cruise control

- The selection of a control loop architecture when the software is embedded in a physical system that involves continuing behavior, especially when the system is subject to external perturbations.
- These conditions hold in the case of cruise control: the system is supposed to maintain constant speed in an automobile despite variations in terrain, vehicle load, air resistance, fuel quality, etc.

Computational elements

• Process definition: the process receives a throttle setting and turns the car's wheels.

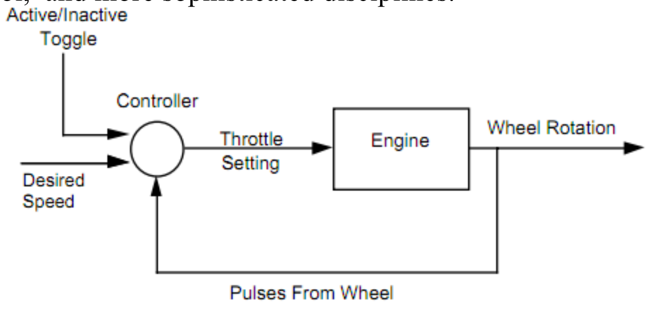
• Control algorithm: This algorithm models the current speed based on the wheel pulses, compares it to the desired speed, and changes the throttle setting.

Data elements

• Controlled variable: For the cruise control, this is the current speed of the vehicle.
• Manipulated variable: For the cruise control, this is the throttle setting.
• Set point: The desired speed is set and modified by the accelerator input and the increase/decrease speed input, respectively.
• Sensor for controlled variable: For cruise control, the current state is the current speed, which is modeled on data from a sensor that delivers wheel pulses using the clock.

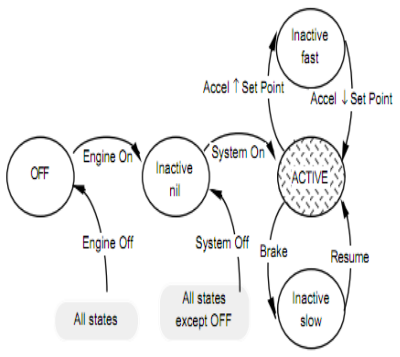
The first task is to model the current speed from the wheel pulses; the designer should validate this model carefully. The model could fail if the wheels spin; this could affect control in two ways. If the wheel pulses are being taken from a drive wheel and the wheel is spinning, the cruise control would keep the wheel spinning (at constant speed) even if the vehicle stops moving.

The controller is implemented as a continuously-evaluating function that matches the dataflow character of the inputs and outputs. Several implementations are possible, including variations on simple on/off control, proportional control, and more sophisticated disciplines.



The set point calculation divides naturally into two parts: (a) determining whether or not the automatic system is active—in control of the throttle and (b) determining the desired speed for use by the controller in automatic mode.

- The active/inactive toggle is triggered by a variety of events, so a state transition design is natural.
- The system is completely off whenever the engine is off. Otherwise there are three inactive and one
- active states.
- In the first inactive state no set point has been established.
- The active/inactive toggle input of the control system is set to active exactly when this state machine is in state Active.



UNIT – V DOCUMENTING THE ARCHITECTURE

PART- A

1. What is Architecture in the Life Cycle?

Any organization that embraces architecture as a foundation for its software development processes needs to understand its place in the life cycle. Several life-cycle models exist in the literature, but one that puts architecture squarely in the middle of things is the evolutionary delivery life cycle model shown in figure

2. What are the ways for designing the Architecture?

A method for designing architecture to satisfy both quality requirements and functional requirements is called attribute-driven design (ADD). ADD takes as input a set of quality attribute scenarios and employs knowledge about the relation between qualities attribute achievement and architecture in order to design the architecture.

3. What do you mean by Attribute Driven Design?

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 architecture and other views as appropriate.

4. What are the steps for Attribute Driven Design?

ADD Steps: Steps involved in attribute driven design (ADD) 1. Choose the module to decompose o Start with entire system o Inputs for this module need to be available o Constraints, functional and quality requirements 2. Refine the module a) Choose architectural drivers relevant to this decomposition b) Choose architectural pattern that satisfies these drivers c) Instantiate modules and allocate functionality from use cases representing using multiple views d) Define interfaces of child modules e) Verify and refine use cases and quality scenarios 3. Repeat for every module that needs further decomposition

5. How to choose an Choose Architectural Pattern ?

For each quality requirement there are identifiable tactics and then identifiable patterns that implement these tactics. o The goal of this step is to establish an overall architectural pattern for the module o The pattern needs to satisfy the architectural pattern for the module tactics selected to satisfy the drivers

6. Define Interfaces Of Child Modules.

It documents what this module provides to others. o Analyzing the decomposition into the 3 views provides interaction information for the interface · Module view: ü Producers/consumers relations ü patterns of communication · Concurrency view: ü Interactions among threads ü Synchronization information · Deployment view ü Hardware requirement ü Timing requirements ü Communication requirements Define Failure.

7. How to Verify And Refine Use Cases And Quality Scenarios As Constraints For The Child Modules?

Functional requirements:

- Using functional requirements to verify and refine

- Decomposing functional requirements assigns responsibilities to child modules
- We can use these responsibilities to generate use cases for the child module
- User interface: Handle user requests ,Translate for raising/lowering module, Display responses
- Raising/lowering door module :Control actuators to raise/lower door,Stop when completed
- opening or closing
- Obstacle detection: Recognize when object is detected, Stop or reverse the closing of the door
- Manage communication with house information system (HIS)

8. What do you mean by Quality scenarios?

Quality scenarios also need to be verified and assigned to child modules. A quality scenario may be satisfied by the decomposition itself, i.e., no additional impact on child modules. A quality scenario may be satisfied by the decomposition but generating constraints for the children. The decomposition may be "neutral" with respect to a quality scenario a quality scenario may not be satisfied with the current decomposition

9. What are the Uses Of Architectural Documentation?

Architecture documentation is both prescriptive and descriptive. That is, for some audiences it prescribes what should be true by placing constraints on decisions to be made. For other audiences it describes what is true by recounting decisions already made about a system's design. All of this tells us that different stakeholders for the documentation have different needs different kinds of information, different levels of information, and different treatments of information.

10. How to choose an relevant view?

- Produce a candidate view list
- Combine views
- Prioritize

11. What is documenting a view?

- Primary presentation- elements and their relationships, contains main information about this system, usually graphical or tabular.
- Element catalog- details of those elements and relations in the picture,
- Context diagram- how the system relates to its Define Confidentiality.
- Confidentiality is the property that data or services are protected from unauthorized access. This means that a hacker cannot access your income tax returns on a government computer.

12. What is Documenting Behavior?

Views present structural information about the system. However, structural information is not sufficient to allow reasoning about some system properties behavior description add information that reveals the ordering of interactions among the elements, opportunities for concurrency, and time dependencies of interactions. Behavior can be documented either about an ensemble of elements working in concert.

13.Why The Architecture Is The Way It Is: Rationale

Cross-view rationale explains how the overall architecture is in fact a solution to its requirements. One might use the rationale to explain: The implications of system-wide design choices on meeting the requirements or satisfying constraints. The effect on the architecture when adding a foreseen new requirement or changing an existing one. The constraints on the developer in implementing a solution. Decision alternatives that were rejected. In general, the rationale explains why a decision was made and what the implications are in changing it.

14. What are the Uses and Audience for Architecture Documentation?

Architecture documentation must – be sufficiently transparent and accessible to be quickly understood by new employees – be sufficiently concrete to serve as a blueprint for construction – have enough information to serve as a basis for analysis.Architecture documentation is both prescriptive and descriptive. – For some audiences, it prescribes what should be true, placing constraints on decisions yet to be made.

15. What do you mean by Views?

Views let us divide software architecture into a number of interesting and manageable representations of the system. Documenting architecture is a matter of documenting the relevant views and then adding documentation that applies to more than one view Auditing is the property that the system tracks activities within it at levels sufficient to reconstruct them.

16. Define SOA.

SOA is a form of technology architecture that adheres to the principles of service-orientation. When realized through web services technology platform, SOA establishes the potential to support and promote these principles throughout the business process and Automation domains of an enterprise.

17. List any 4 characteristics of SOAs.

a. SOA supports vendor diversity.
b. SOA is fundamentally autonomous.
c. SOA promotes discovery.
d. SOA fosters intrinsic interoperability.

18. List the components SOA.

Messages- contains data for operation
Operations-holds the logic
Services- group the operation together
Processes-large units of work with business logic

19. What is WSDL?

Web services Definition Language is the focal point of service design as it is used to design the abstract and concrete definition of service interfaces. WSDL definition hosts multiple child constructs associated with abstract and concrete parts of the service description.

20. Define service. How do services communicate?

A service represents a logically grouped set of operations capable of performing the related units of work.Services communicate via SOAP messages.

21. Define Cloud Computing.

Cloud computing is a model for enabling convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction

22. Define Cloud Computing Architecture.

The Cloud Computing Architecture of a cloud solution is the structure of the system, which comprise on-premise and cloud resources, services, middleware, and software components, geo-location, the externally visible properties of those, and the relationships between them.

23. What is meant by documentation of a system's cloud computing.

Documenting facilitates communication between stakeholders, documents early decisions about highlevel design, and allows reuse of design components and patterns between projects.

24. Name the major building block of cloud computing architecture

- Reference Architecture
- Technical architecture
- Deployment Operation Architecture

25. Define Adaptive structures.

Software that evaluates its own performance and changes behavior when the evaluation indicates that it is not accomplishing what the software is intended to do..."

PART- B

1. Explain the three steps for choosing views for a project?

Three-step procedure for choosing the views for your project.

1. Produce a candidate view list. Begin by building a stakeholder/view table, for your project. Your stakeholder list is likely to be different from the one in the table, but be as comprehensive as you can. For the columns, enumerate the views that apply to your system. Some views (such as decomposition or uses) apply to every system, while others (the layered view, most component-and-connector views such as client-server or shared data) only apply to systems designed that way. Once you have the rows and columns defined, fill in each cell to describe how much information the stakeholder requires from the view: none, overview only, moderate detail, or high detail.

2. Combine views. The candidate view list from step 1 is likely to yield an impractically large number of views. To reduce the list to a manageable size, first look for views in the table that require only overview depth or that serve very few stakeholders. See if the stakeholders could be equally well served by another view having a stronger constituency. Next, look for views that are good candidates to be combined?that is, a view that gives information from two or more views at once. For small and medium projects, the implementation view is often easily overlaid with the module decomposition view. The module decomposition view also pairs well with uses or layered views. Finally, the deployment view usually combines well with whatever component-and-connector view shows the components that are allocated to hardware elements?

3. Prioritize. After step 2 you should have an appropriate set of views to serve your stakeholder community. At this point you need to decide what to do first. How you decide depends on the details specific to your project, but remember that you don't have to complete one view before starting another. People can make progress with overview-level information, so a breadth-first approach is often the best. Also, some stakeholders' interests supersede others. A project manager or the management of a company with which yours is partnering demands attention and information early and often.

2. Explain the options for representing connectors and systems in UML?

Connectors types as associations and connector instances as links
Connector types as association classes.
Connectors types as classes and connector instances as objects

In addition
System as UML subsystem
System as contained objects
System as collaboration

What is a Software Connector?

- Architectural element that models
  - Interactions among components

    - Rules that govern those interactions
- Simple interactions
    - Procedure calls
    - Shared variable access
- Complex & semantically rich interactions
    - Client-server protocols
    - Database access protocols
    - Asynchronous event multicast
- Each connector provides
    - Interaction duct(s)

Transfer of control and/or data

Software Connector Roles

- Locus of interaction among set of components
- Protocol specification (sometimes implicit) that defines its properties
    - Types of interfaces it is able to mediate
    - Assurances about interaction properties
    - Rules about interaction ordering
    - Interaction commitments (e.g., performance)
- Roles
    - Communication
    - Coordination
    - Conversion
    - Facilitation

Connector Types

- Procedure call
- Data access
- Event
- Stream
- Linkage
- Distributor
- Arbitrator
- Adaptor

3. Explain with a neat diagram, the evolutionary delivery life cycle model?

ARCHITECTURE IN THE LIFE CYCLE

Any organization that embraces architecture as a foundation for its software development processes needs to understand its place in the life cycle. Several life-cycle models exist in the literature, but one that puts architecture squarely in the middle of things is the evolutionary delivery life cycle model Evolutionary Delivery Life Cycle

The intent of this model is to get user and customer feedback and iterate through several releases before the final release. The model also allows the adding of functionality with each iteration and the delivery of a limited version once a sufficient set of features has been developed.

WHEN CAN I BEGIN DESIGNING?

- The life-cycle model shows the design of the architecture as iterating with preliminary requirements analysis. Clearly, you cannot begin the design until you have some idea of the system requirements. On the other hand, it does not take many requirements in order for design to begin.

- An architecture is "shaped" by some collection of functional, quality, and business requirements. We call these shaping requirements *architectural drivers* and we see examples of them in our case studies like modifiability, performance requirements availability requirements and so on.

- To determine the architectural drivers, identify the highest priority business goals. There should be relatively few of these. Turn these business goals into quality scenarios or use cases.

4. List the steps of ADD and explain the uses of architectural documentation?

USES OF ARCHITECTURAL DOCUMENTATION

- Architecture documentation is both prescriptive and descriptive.
- That is, for some audiences it prescribes what should be true by placing constraints on decisions to be made.
- For other audiences it describes what is true by recounting decisions already made about a system's design.

- All of this tells us that different stakeholders for the documentation have different needs—different kinds of information, different levels of information, and different treatments of information.

- One of the most fundamental rules for technical documentation in general, and software architecture documentation in particular, is to write from the point of view of the reader.

- Documentation that was easy to write but is not easy to read will not be used, and "easy to read" is in the eye of the beholder—or in this case, the stakeholder.

- Documentation facilitates that communication.

- In addition, each stakeholders come in two varieties: seasoned and new.

- A new stakeholder will want information similar in content to what his seasoned counterpart wants, but in smaller and more introductory doses.

- Architecture documentation is a key means for educating people who need an overview: new developers, funding sponsors, visitors to the project, and so forth.

5. Bring out the concept of view as applied to architectural documentation.

A*view* is a coherent set of architectural elements or constructs along with the relationships among them, as written by and read by system stakeholders.

A *functional* view of the architecture enlists the various architectural building blocks, the relationships among them, and how they are allocated to different layers in the architecture.

The *operational* view (also called the technology view) enlists the various infrastructure and middleware software components that provide the run time platform for the functional architecture components to be deployed upon.

To an application architect, the functional view assumes primary importance. To the infrastructure architect, the operational view is the one to focus on.

View	Description
Logical view	Addresses the static design model
Process view	Addresses the design's dynamic view
Physical view	Addresses how the software components are mapped onto the hardware infrastructure
Development view	Represents the static organization of the software components in the development time environment

The fifth view is more of a litmus test view. It takes a set of architecturally significant use cases (business scenarios) and shows how the set of architectural elements in each of the four views, together with the architectural constraints and decisions around those elements, can be used to realize the use cases.

View	Description
Conceptual architecture view	Describes the system in terms of its major design elements and the relationships among them
Module interconnection architecture view	Describes the functional decomposition and how the software modules are arranged in different architectural layers
Execution architecture view	Describes the dynamic structure of the systems
Code architecture view	Describes how the source code, binaries, and libraries are arranged in a development environment

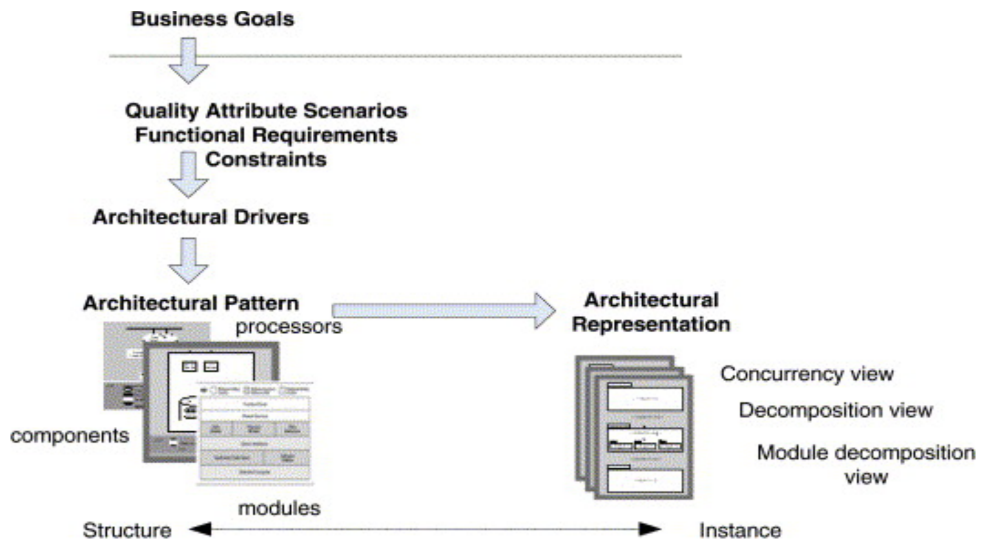
6.Briefly explain the different steps performed while designing an architecture using the ADD method.

Attribute-Driven Design(ADD)

- It was developed at the SEI.
- Approach to defining software architectures by basing the design process on the architecture's quality attribute requirements.

- In ADD, architectural design follows a recursive decomposition process where, at each stage in the decomposition, architectural tactics and patterns are chosen to satisfy a set of quality attribute scenarios.
- The architecture designed represents the high-level design choices documented as containers for functionality and interactions among the containers using views.
- The views that are most commonly used are one or more of the following:
    - module decomposition view
    - concurrency view, and
    - deployment view.

Steps when designing an architecture using the ADD method:
1. Choose the module to decompose.
   The module to start with is usually the whole system.
2. Refine the modules according to these steps:
   a. Choose the architectural drivers from the set of concrete quality scenarios and functional requirements.
   b. Choose an architectural pattern that satisfies the drivers.
   c. Instantiate modules and allocate functionality from use cases, and represent the results using multiple views.
   d. Define interfaces of the child modules.
   e. Verify and refine the use cases and quality scenarios and make them constraints for the child modules.
3. Repeat the steps above for every module that needs further decomposition.



7.Explain the roots of SOA

IT departments started to recognize the need for a stadndardized definition of a baseline application that could act as a template for all others

This definition was abstract in nature but sprciafically explained the technology ,boundarires rules,limitations and desin characterstics that apply to all solutions based on this template

This was the birth of application software.

Types of archeitecture
Application archetercture
Enterprise archeitecture
Service oriented architecture

Applications architecture is the science and art of ensuring the suite of applications being used by an organization to create the composite architecture is scalable, reliable, available and manageable.

One not only needs to understand and manage the dynamics of the functionalities the composite architecture is implementing but also help formulate the deployment strategy and keep an eye out for technological risks that could jeopardize the growth and/or operations of the organization

An **enterprise architecture** (EA) is a conceptual blueprint that defines the structure and operation of an organization.
The intent of an **enterprise architecture** is to determine how an organization can most effectively achieve its current and future objectives.

A **service-oriented architecture** (**SOA**) is an architectural pattern in computer software design in which application components provide services to other components via a communications protocol, typically over a network.
The principles of **service-orientation** are independent of any vendor, product or technology.

8. Briefly explain in detail about WSDL and SOAP basics in service oriented design?
WSDL is an XML format for describing network services as a set of endpoints operating on messages containing either document-oriented or procedure-oriented information.

The operations and messages are described abstractly, and then bound to a concrete network protocol and message format to define an endpoint

WSDL is often used in combination with SOAP and XML Schema.

WSDL stands for Web Services Description Language. It is the standard format for describing a web service. WSDL was developed jointly by Microsoft and IBM.

Features of WSDL

- WSDL is an XML-based protocol for information exchange in decentralized and distributed environments.
- WSDL definitions describe how to access a web service and what operations it will perform.
- WSDL is a language for describing how to interface with XML-based services.
- WSDL is an integral part of Universal Description, Discovery, and Integration (UDDI), an XML-based worldwide business registry.
- WSDL is the language that UDDI uses.
- WSDL is pronounced as 'wiz-dull' and spelled out as 'W-S-D-L'.

SOAP stands for Simple Object Access Protocol.

SOAP is an application communication protocol.

SOAP is a format for sending and receiving messages.

SOAP is platform independent.

SOAP is based on XML.

SOAP is an open-standard, XML-based messaging protocol for exchanging information among computers.

The nature of SOAP −

- SOAP is a communication protocol designed to communicate via Internet.
- SOAP can extend HTTP for XML messaging.
- SOAP provides data transport for Web services.
- SOAP can exchange complete documents or call a remote procedure.
- SOAP can be used for broadcasting a message.
- SOAP is platform- and language-independent.
- SOAP is the XML way of defining what information is sent and how.
- SOAP enables client applications to easily connect to remote services and invoke remote methods.

The SOAP architecture consists of several layers of specifications for:

- message format
- Message Exchange Patterns (MEP)
- underlying transport protocol bindings
- message processing models
- protocol extensibility

9. Write short notes on:

i) Forming team structures
FORMING THE TEAM STRUCTURES
Once the architecture is accepted we assign teams to work on different portions of the design and development.

- Once architecture for the system under construction has been agreed on, teams are allocated to work on the major modules and a work breakdown structure is created that reflects those teams.
- Each team then creates its own internal work practices.
- For large systems, the teams may belong to different subcontractors.
  - Teams adopt “work practices’ including
  - Team communication via website/bulletin boards
  - Naming conventions for files
  - Configuration/revision control system
  - Quality assurance and testing procedure

The teams within an organization work on modules, and thus within team high level of communication is necessary

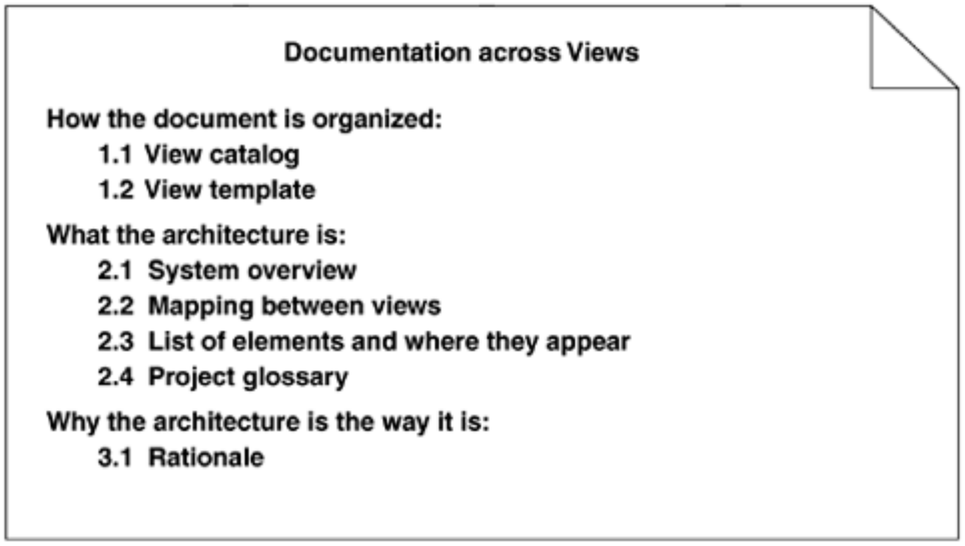
ii)Documenting across views

Cross-view documentation consists of just three major aspects, which we can summarize as how-what-why:

1. How the documentation is laid out and organized so that a stakeholder of the architecture can find the information he or she needs efficiently and reliably. This part consists of a view catalog and a view template.

2. What the architecture is. Here, the information that remains to be captured beyond the views themselves is a short system overview to ground any reader as to the purpose of the system; the way the views are related to each other; a list of elements and where they appear; and a glossary that applies to the entire architecture.

3. Why the architecture is the way it is: the context for the system, external constraints that have been imposed to shape the architecture in certain ways, and the rationale for coarse-grained large-scale decisions.



Documentation across Views

How the document is organized:
- **1.1 View catalog**
- **1.2 View template**

What the architecture is:
- **2.1 System overview**
- **2.2 Mapping between views**
- **2.3 List of elements and where they appear**
- **2.4 Project glossary**

Why the architecture is the way it is:
- **3.1 Rationale**

Source: Adapted from [Clements 03].

iii) Documenting interfaces

An interface is a boundary across which two independent entities meet and interact or communicate with each other.
When an element has multiple interfaces, identify the individual interfaces to distinguish them. This usually means naming them. You may also need to provide a version number.

2. Resources provided:

The heart of an interface document is the resources that the element provides.
- ❖ *Resource syntax* – this is the resource's signature
- ❖ *Resource Semantics:*
  - Assignment of values of data
  - Changes in state
  - Events signaled or message sent
  - how other resources will behave differently in future
  - humanly observable results
- ❖ *Resource Usage Restrictions*

- initialization requirements
- limit on number of actors using resource

3. Data type definitions:

If used if any interface resources employ a data type other than one provided by the underlying programming language, the architect needs to communicate the definition of that type. If it is defined by another element, then reference to the definition in that element's documentation is sufficient.

4. Exception definitions:

These describe exceptions that can be raised by the resources on the interface. Since the same exception might be raised by more than one resource, if it is convenient to simply list each resource's exceptions but define them in a dictionary collected separately.

5. Variability provided by the interface.

Does the interface allow the element to be configured in some way? These configuration parameters and how they affect the semantics of the interface must be documented.

6. Quality attribute characteristics:

The architect needs to document what quality attribute characteristics (such as performance or reliability) the interface makes known to the element's users

7. Element requirements:

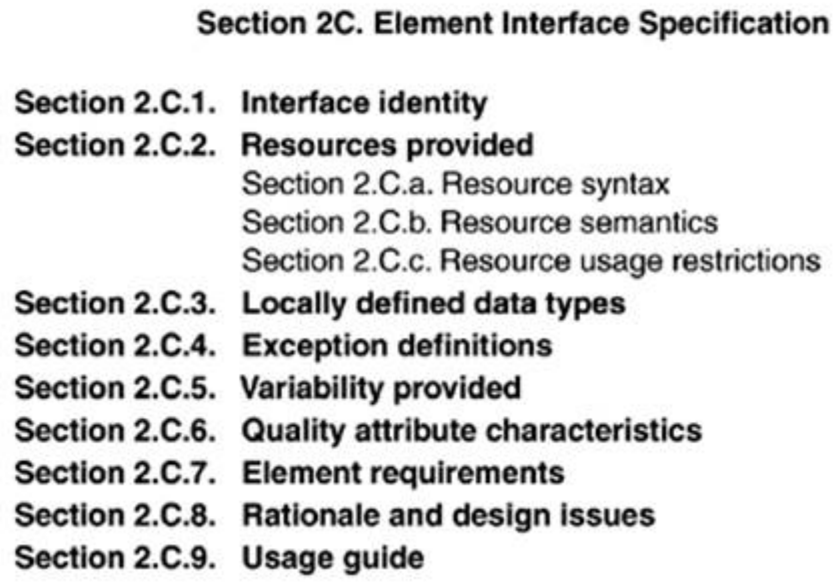
What the element requires may be specific, named resources provided by other elements. The documentation obligation is the same as for resources provided: syntax, semantics, and any usage restrictions.

8. Rationale and design issues:

Why these choices the architect should record the reasons for an elements interface design. The rationale should explain the motivation behind the design, constraints and compromises, what alternatives designs were considered.

9. Usage guide:

Item 2 and item 7 document an element's semantic information on a per resource basis. This sometimes falls short of what is needed. In some cases semantics need to be reasoned about in terms of how a broad number of individual interactions interrelate.



Section 2C. Element Interface Specification

Section 2.C.1. **Interface identity**
Section 2.C.2. **Resources provided**
- Section 2.C.a. Resource syntax
- Section 2.C.b. Resource semantics
- Section 2.C.c. Resource usage restrictions

Section 2.C.3. **Locally defined data types**
Section 2.C.4. **Exception definitions**
Section 2.C.5. **Variability provided**
Section 2.C.6. **Quality attribute characteristics**
Section 2.C.7. **Element requirements**
Section 2.C.8. **Rationale and design issues**
Section 2.C.9. **Usage guide**