Internal Assessment Test 2 – May 2017

Sub:	**Software Architectures**							**Code:**	**10IS81**
Date:	08/05/2017	Duration:	90 mins	Max Marks:	50	**Sem:**	VIII A & B	**Branch:**	**ISE**

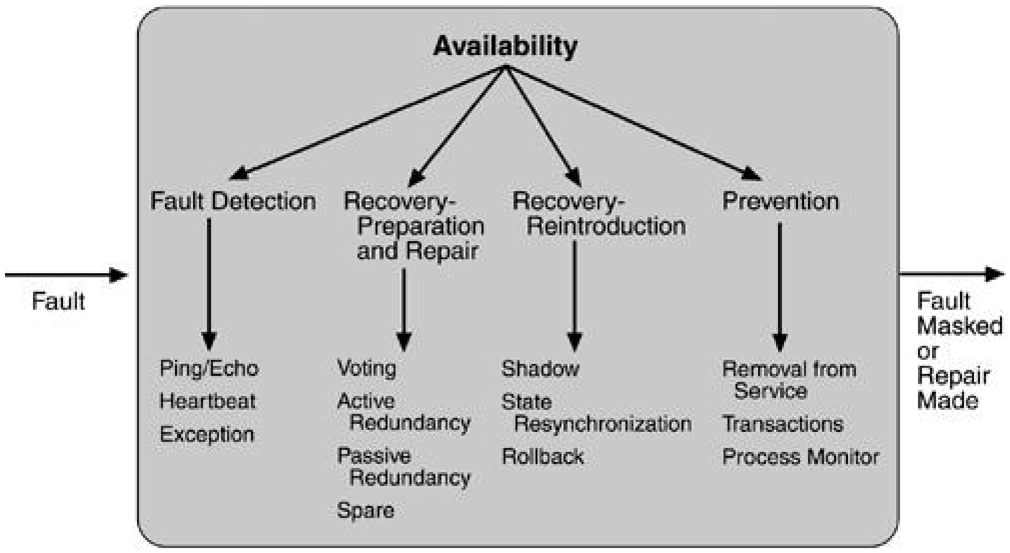
Answer any 5 questions. All questions carry equal marks.

Marks

1. a. What are the key concepts of availability tactics? Explain the same with an example. [10]

Soln.

- A failure occurs when the system no longer delivers a service that is consistent with its specification; this failure is observable by the system's users. A fault (or combination of faults) has the potential to cause a failure.



- ***Fault detection***-Three widely used tactics for recognizing faults are ping/echo, heartbeat, and exceptions.
- **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 another component listens for it. If the heartbeat fails, the originating component is assumed to have failed and a fault correction component is notified.
- **Exceptions.** One method for recognizing faults is to encounter an exception, which is raised when one of the fault classesis recognized. The exception handler typically executes in the same process that introduced the exception.
- ***Fault recovery***-Fault recovery consists of preparing for recovery and making the system repair. Some preparation and repair tactics follow.
- **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. The voting algorithm can be "majority

rules" or "preferred component" or some other algorithm. This method is used to correct faulty operation of algorithms or failure of a processor and is often used in control systems.

- **Active redundancy** (hot restart). All redundant components respond to events in parallel. Consequently, they are all in the same state. The response from only one component is used (usually the first to respond), and the rest are discarded. When a fault occurs, the downtime of systems using this tactic is usually milliseconds since the backup is current and the only time to recover is the switching time. 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. In a highly available distributed system, the redundancy may be in the communication paths.
- **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. This approach is also used in control systems, often when the inputs come over communication channels or from sensors and have to be switched from the primary to the backup on failure.
- **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. Making a checkpoint of the system state to a persistent device periodically and logging all state changes to a persistent device allows for the spare to be set to the appropriate state.

- **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. The updating approach will depend on the downtime that can be sustained, the size of the update, and the number of messages required for the update

- **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.
- ***Fault prevention-***
- **Removal 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.

2. a. Explain business qualities. [5]

Soln. In addition to the qualities that apply directly to a system, a number of business quality goals frequently shape a system's architecture. These goals center on cost, schedule, market, and marketing considerations. Each suffers from the same ambiguity that system qualities have, and they need to be made specific with scenarios in order to make them suitable for influencing the design process and to be made testable.

- **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. Time to market is often reduced by using prebuilt elements such as commercial off-the-shelf (COTS) products or elements re-used from previous projects. The ability to insert or deploy a subset of the system depends on the decomposition of the system into elements.

- **Cost and benefit**. The development effort will naturally have a budget that must not be exceeded. Different architectures will yield different development costs. An architecture that is highly flexible will typically be costlier to build than one that is rigid (although it will be less costly to maintain and modify).

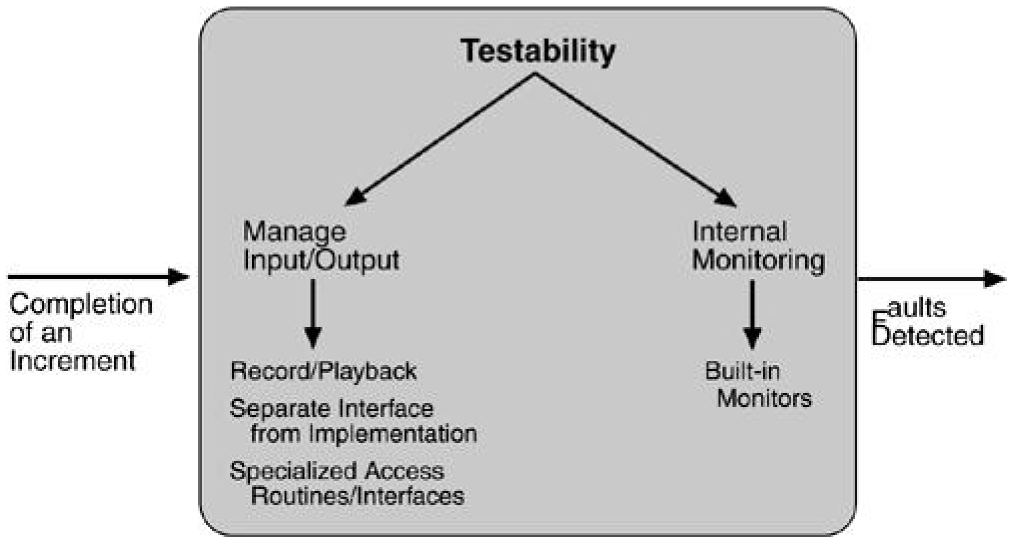
- **Projected lifetime of the system**. If the system is intended to have a long lifetime, modifiability, scalability, and portability become important. But building in the additional infrastructure (such as a layer to support portability) will usually compromise time to market. On the other hand, a modifiable, extensible product is more likely to survive longer in the marketplace, extending its lifetime.

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

- **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.

b. Explain Testability tactics [5]

Soln. The goal of tactics for testability is to allow for easier testing when an increment of software development is completed. .
two categories of tactics for testing: providing input and capturing output, and internal monitoring.


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

INPUT/OUTPUT

There are three tactics for managing input and output for testing.

- **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 and represents output from one component and input to another. 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. Substituting a specialized component allows the component being replaced to act as a test harness for the remainder of the system.
- **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. For example, metadata might be made available through a specialized interface that a test harness would use to drive its activities. Having a hierarchy of test interfaces in the architecture means that test cases can be applied at any level in the architecture and that the testing functionality is in place to observe the response.

INTERNAL MONITORING

A component can implement tactics based on internal state to support the testing process.

- 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 via an instrumentation technique such as aspect-oriented programming or preprocessor macros. 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.

3. a. What is Quality attribute scenario? List the parts of such a scenario. [2]

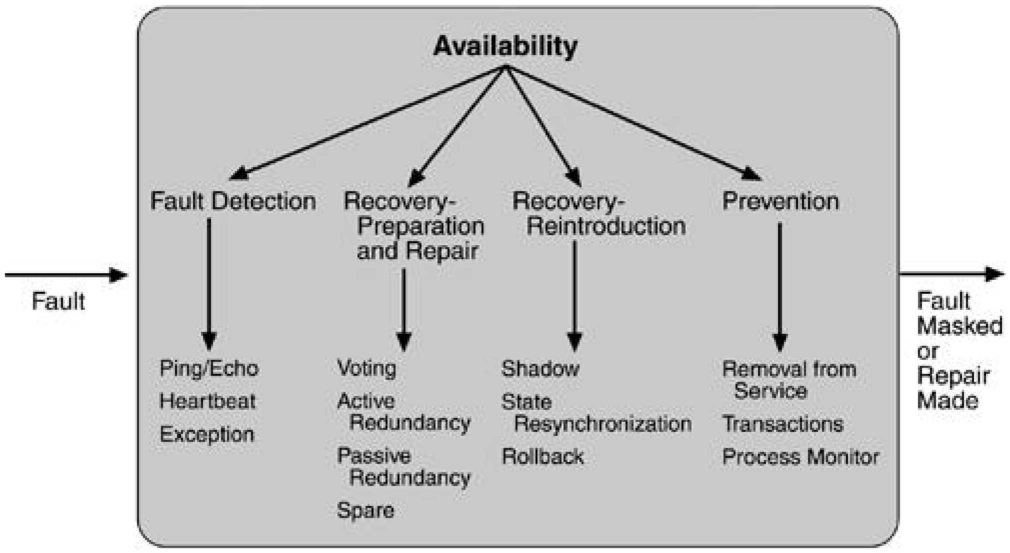
Soln. General scenarios provide a framework for generating a large number of generic, system-independent, quality-attribute-specific scenarios. Each is potentially but not necessarily relevant to the system you are concerned with.Making a general scenario system specific means translating it into concrete terms for the particular system.

- Source of stimulus. This is some entity (a human, a computer system, or any other actuator) that generated the stimulus.
- Stimulus. The stimulus is a condition that needs to be considered when it arrives at a system.
- Environment. The stimulus occurs within certain conditions. The system may be in an overload condition or may be running when the stimulus occurs, or some other condition may be true.
- Artifact. Some artifact is stimulated. This may be the whole system or some pieces of it.
- Response. The response is the activity undertaken after the arrival of the stimulus.
- Response measure. When the response occurs, it should be measurable in some fashion so that the requirement can be tested.

b. Explain how faults are detected, recovered and prevented [8]

Soln.

- A failure occurs when the system no longer delivers a service that is consistent with its specification; this failure is observable by the system's users. A fault (or combination of faults) has the potential to cause a failure.



- ***Fault detection***- Three widely used tactics for recognizing faults are ping/echo, heartbeat, and exceptions.
- **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 another component listens for it. If the heartbeat fails, the originating component is assumed to have failed and a fault correction component is notified.
- **Exceptions.** One method for recognizing faults is to encounter an exception, which is raised when one of the fault classes is recognized. The exception handler typically executes in the same process that introduced the exception.
- ***Fault recovery-*** Fault recovery consists of preparing for recovery and making the system repair. Some preparation and repair tactics follow.
- **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. The voting algorithm can be "majority rules" or "preferred component" or some other algorithm. This method is used to correct faulty operation of algorithms or failure of a processor and is often used in control systems.
- **Active redundancy** (hot restart). All redundant components respond to events in parallel. Consequently, they are all in the same state. The response from only one component is used (usually the first to respond), and the rest are discarded. When a fault occurs, the downtime of systems using this tactic is usually milliseconds since the backup is current and the only time to recover is the switching time. 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. In a highly available distributed system, the redundancy may be in the communication paths.
- **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. This approach is also used in control systems, often when the inputs come over communication channels or from sensors and have to be switched from the primary to the backup on failure.
- **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. Making a checkpoint of the system state to a persistent device periodically and logging all state changes to a persistent device allows for the spare to be set to the appropriate state.
- **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. The updating approach will depend on the downtime that can be sustained, the size of the

update, and the number of messages required for the update

- **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.
- ***Fault prevention-***
- **Removal 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.

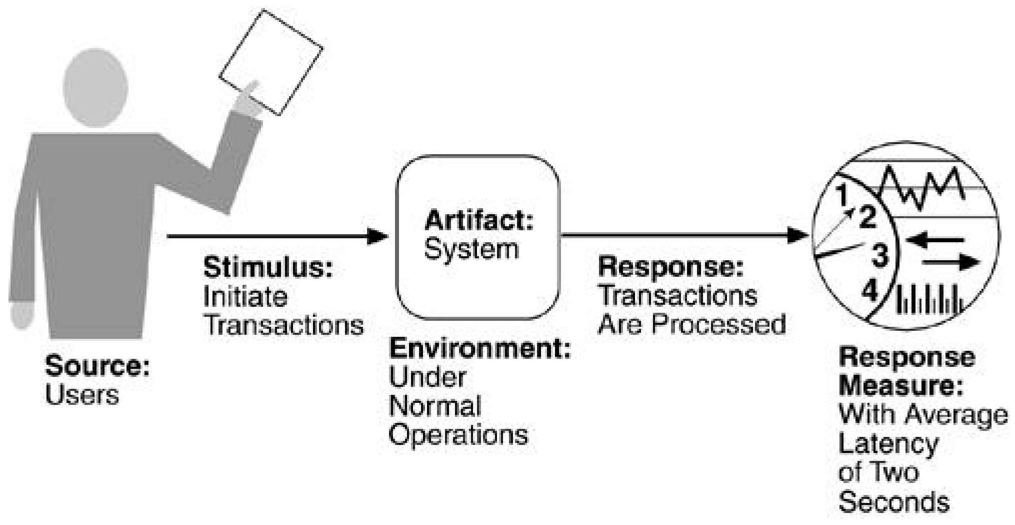
4. a. Explain Performance and Security scenarios with a neat block diagram [10]

Soln. Performance scenarios

- Performance is about timing. Events (interrupts, messages, requests from users, or the passage of time) occur, and the system must respond to them. There are a variety of characterizations of event arrival and the response but basically performance is concerned with how long it takes the system to respond when an event occurs.
- One of the things that make performance complicated is the number of event sources and arrival patterns. An arrival pattern for events may be characterized as either periodic or stochastic. For example, a periodic event may arrive every 10 milliseconds. Periodic event arrival is most often seen in real-time systems. Stochastic arrival means that events arrive according to some probabilistic distribution. Events can also arrive sporadically, that is, according to a pattern not capturable by either periodic or stochastic characterizations.

- A performance scenario begins with a request for some service arriving at the system. Satisfying the request requires resources to be consumed. While this is happening the system may be simultaneously servicing other requests.

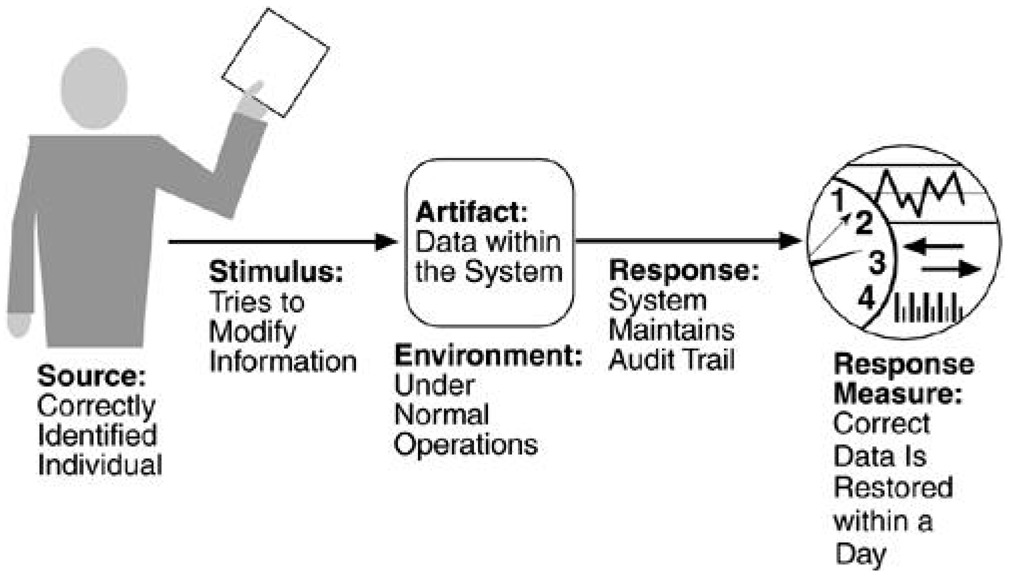
- The response of the system to a stimulus can be characterized by latency, deadlines in processing, the throughput of the system, the number of events not processed because the system was too busy to respond, and the data that was lost because the system was too busy.


Source: Users
Stimulus: Initiate Transactions
Artifact: System
Environment: Under Normal Operations
Response: Transactions Are Processed
Response Measure: With Average Latency of Two Seconds
1
2
3
4

- Source of stimulus. The stimuli arrive either from external (possibly multiple) or internal sources. In our example, the source of the stimulus is a collection of users.
- Stimulus. The stimuli are the event arrivals. The arrival pattern can be characterized as periodic, stochastic, or sporadic. In our example, the stimulus is the stochastic initiation of 1,000 transactions per minute.
- Artifact. The artifact is always the system's services, as it is in our example.
- Environment. The system can be in various operational modes, such as normal, emergency, or overload. In our example, the system is in normal mode.
- Response. The system must process the arriving events. This may cause a change in the system environment (e.g., from normal to overload mode). In our example, the transactions are processed.
- Response measure. The response measures are the time it takes to process the arriving events (latency or a deadline by which the event must be processed), the variation in this time (jitter), the number of events that can be processed within a particular time interval (throughput), or a characterization of the events that cannot be processed (miss rate, data loss). In our example, the transactions should be processed with an average latency of two seconds.

Security scenarios

- Security is a measure of the system's ability to resist unauthorized usage while still providing its services to legitimate users. An attempt to breach security is called an attack and can take a number of forms. It may be an unauthorized attempt to access data or services or to modify data, or it may be intended to deny services to legitimate users.
- Attacks, often occasions for wide media coverage, may range from theft of money by electronic transfer to modification of sensitive data, from theft of credit card numbers to destruction of files on computer systems, or to denial-of-service attacks carried out by worms or viruses.

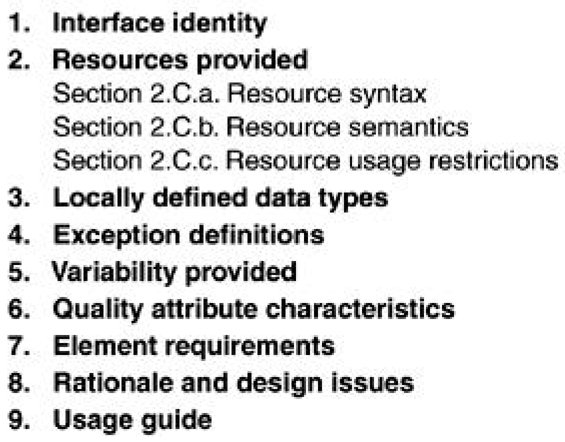

Source: Correctly Identified Individual
Stimulus: Tries to Modify Information
Artifact: Data within the System
Environment: Under Normal Operations
Response: System Maintains Audit Trail
Response Measure: Correct Data Is Restored within a Day
1
2
3
4

- Source of stimulus. The source of the attack may be either a human or another system. It may have been previously identified (either correctly or incorrectly) or may be currently unknown. If the source of the attack is highly motivated (say politically motivated), then defensive measures such as "We know who you are and will prosecute you" are not likely to be effective; in such cases the motivation of the user may be important. If the source has access to vast resources (such as a government), then defensive measures are very difficult. The attack itself is unauthorized access, modification, or denial of service.
- The difficulty with security is allowing access to legitimate users and determining legitimacy. If the only goal were to prevent access to a system, disallowing all access would be an effective defensive measure.
- Stimulus. The stimulus is an attack or an attempt to break security. We characterize this as an unauthorized person or system trying to display information, change and/or delete information, access services of the system, or reduce availability of system services. In Figure 4.6, the stimulus is an attempt to modify data.
- Artifact. The target of the attack can be either the services of the system or the data within it. In our example, the target is data within the system.
- Environment. The attack can come when the system is either online or offline, either connected to or disconnected from a network, either behind a firewall or open to the network.
- Response. Using services without authorization or preventing legitimate users from using services is a different goal from seeing sensitive data or modifying it. Thus, the system must authorize legitimate users and grant them access to data and services, at the same time rejecting unauthorized users, denying them access, and reporting unauthorized access. Not only does the system need to provide access to legitimate users, but it needs to support the granting or withdrawing of access. One technique to prevent attacks is to cause fear of punishment by maintaining an audit trail of modifications or attempted accesses. An audit trail is also useful in correcting from a successful attack. In Figure 4.6, an audit trail is maintained.

- Response measure. Measures of a system's response include the difficulty of mounting various attacks and the difficulty of recovering from and surviving attacks. In our example, the audit trail allows the accounts from which money was embezzled to be restored to their original state. Of course, the embezzler still has the money, and he must be tracked down and the money regained, but this is outside of the realm of the computer system.

5. a. Explain different templates for documentation interfaces. [10]

Soln. An interface is a boundary across which two independent entities meet and interact or communicate with each other.



1. Interface identity
2. Resources provided
Section 2.C.a. Resource syntax
Section 2.C.b. Resource semantics
Section 2.C.c. Resource usage restrictions
3. Locally defined data types
4. Exception definitions
5. Variability provided
6. Quality attribute characteristics
7. Element requirements
8. Rationale and design issues
9. Usage guide

1. **Interface identity**. 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. Define them by giving their syntax, their semantics, and any restrictions on their usage.At a minimum, the interface is named; the architect can also specify signature information.
    - **Resource syntax**. This is the resource's signature. The signature includes any information another program will need to write a syntactically correct program that uses the resource. The signature includes the resource name, names and logical data types of arguments (if any), and so forth.

- **Resource semantics**. This describes the result of invoking the resource. It might include
- - **assignment of values to data that the actor invoking the resource can access**. It might be as simple as setting the value of a return argument or as far-reaching as updating a central database.

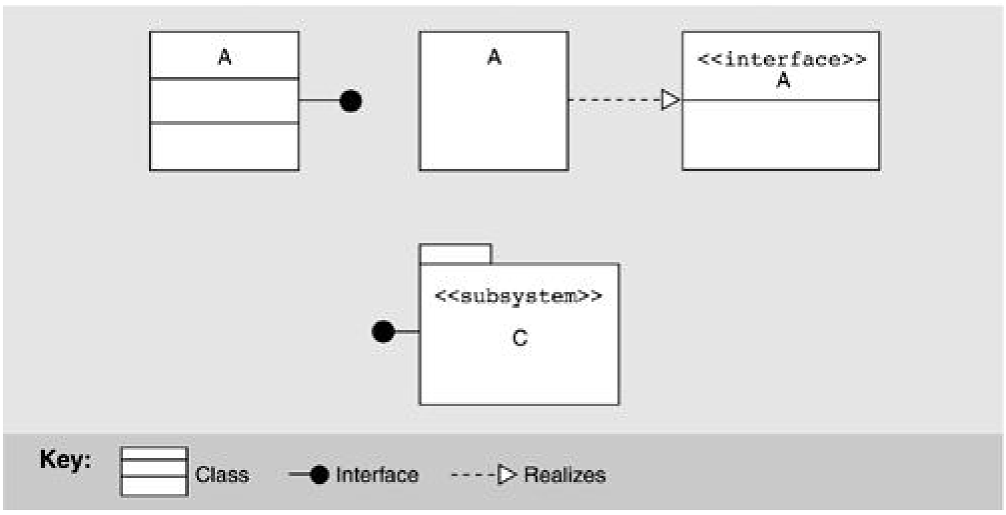
  - **events that will be signaled** or messages that will be sent as a result of using the resource.

  - **how other resources will behave** in the future as the result of using this resource. For example, if you ask a resource to destroy an object, trying to access that object in the future through other resources will produce quite a different outcome (an error).

- **humanly observable results**. These are prevalent in embedded systems; for example, calling a program that turns on a display in a cockpit has a very observable effect: The display comes on.

- Resource usage restrictions. Under what circumstances may this resource be used? Perhaps data must be initialized before it can be read, or a particular method cannot be invoked unless another is invoked first.

Figure Interfaces in UML



3. **Data type definitions**. 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 data type. If it is defined by another element, then a 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, it is often convenient to simply list each resource's exceptions but define them in a dictionary collected separately. This section is that dictionary. Common exception-handling behavior can also be defined here.

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 of the interface.** The architect needs to document what quality attribute characteristics the interface makes known to the element's users. This information may be in the form of constraints on implementations of elements that will realize the interface. Which qualities you choose to concentrate on and make promises about will depend on context.

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**. As with rationale for the architecture at large, the

architect should record the reasons for an element's interface design. The rationale should explain the motivation behind the design, constraints and compromises, what alternative designs were considered and rejected, and any insight the architect has about how to change the interface in the future.

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. Essentially, a protocol is involved that is documented by considering a sequence of interactions.

6. a. List the steps of ADD. [2]

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

2. Refine the module according to these steps:

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

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

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

   d. Define interfaces of the child modules. The decomposition provides modules and constraints on the types of module interactions. Document this information in the interface document for each module.

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

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

b. Explain all the steps involved in refining the module. [8]

Soln. 2.a Choose the Architectural Drivers

- 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.
- The determination of architectural drivers is not always a top-down process. Sometimes detailed investigation is required to understand the ramifications of particular requirements. For example, to determine if performance is an issue for a particular system configuration, a prototypical implementation of a piece of the system may be required. In our example, determining that the performance

requirement is an architectural driver requires examining the mechanics of a garage door and the speed of the potential processors.

2.b Choose an Architectural Pattern

- for each quality there are identifiable tactics (and patterns that implement these tactics) that can be used in an architecture design to achieve a specific quality. Each tactic is designed to realize one or more quality attributes, but the patterns in which they are embedded have an impact on other quality attributes
- The goal of step 2b is to establish an overall architectural pattern consisting of module types. The pattern satisfies the architectural drivers and is constructed by composing selected tactics. Two main factors guide tactic selection. The first is the drivers themselves. The second is the side effects that a pattern implementing a tactic has on other qualities.
- we see performance and modifiability as the critical quality attributes for available tactics. The modifiability tactics are "localize changes," "prevent the ripple effect," and "defer binding time." Moreover, since our modifiability scenarios are concerned primarily with changes that will occur during system design, the primary tactic is "localize changes." We choose semantic coherence and information hiding as our tactics and combine them to define virtual machines for the affected areas. The performance tactics are "resource demand" and "resource arbitration." We choose one example of each: "increase computational efficiency" and "choose scheduling policy." This yields the following tactics:

- **Semantic coherence and information hiding**. Separate responsibilities dealing with the user interface, communication, and sensors into their own modules. We call these modules virtual machines and we expect all three to vary because of the differing products that will be derived from the architecture. Separate the responsibilities associated with diagnosis as well.

- **Increase computational efficiency**. The performance-critical computations should be made as efficient as possible.

- **Schedule wisely.** The performance-critical computations should be scheduled to ensure the achievement of the timing deadline.

2.c Instantiate Modules and Allocate Functionality Using Multiple Views

- we identified a non-performance-critical computation running on top of a virtual machine that manages communication and sensor interactions. The software running on top of the virtual machine is typically an application. In a concrete system we will normally have more than one module. There will be one for each "group" of functionality; these will be instances of the types shown in the pattern. Our criterion for allocating functionality is similar to that used in functionality-based design methods, such as most object-oriented design methods.
- **Allocate functionality-**Applying use cases that pertain to the parent module helps the architect gain a more detailed understanding of the distribution of functionality. This also may lead to adding or removing child modules to fulfill all the functionality required.

- Assigning responsibilities to the children in a decomposition also leads to the discovery of necessary information exchange. This creates a producer/consumer relationship between those modules, which needs to be recorded
- **Represent the architecture with views-** ADD uses three common views.
- **Module decomposition view**. module decomposition view provides containers for holding responsibilities as they are discovered. Major data flow relationships among the modules are also identified through this view.
- **Concurrency view**. In the concurrency view dynamic aspects of a system such as parallel activities and synchronization can be modeled. This modeling helps to identify resource contention problems, possible deadlock situations, data consistency issues, and so forth. Modeling the concurrency in a system likely leads to discovery of new responsibilities of the modules, which are recorded in the module view. It can also lead to discovery of new modules, such as a resource manager, in order to solve issues of concurrent access to a scarce resource and the like.

  To understand the concurrency in a system, the following use cases are illuminating:

  - **Two users doing similar things at the same time.** This helps in recognizing resource contention or data integrity problems. In our garage door example, one user may be closing the door remotely while another is opening the door from a switch.

  - **One user performing multiple activities simultaneously**. This helps to uncover data exchange and activity control problems. In our example, a user may be performing diagnostics while simultaneously opening the door.

  - Starting up the system. This gives a good overview of permanent running activities in the system and how to initialize them. It also helps in deciding on an initialization strategy, such as everything in parallel or everything in sequence or any other model. In our example, does the startup of the garage door opener system depend on the availability of the home information system? Is the garage door opener system always working, waiting for a signal, or is it started and stopped with every door opening and closing?
  - **Shutting down the system**. This helps to uncover issues of cleaning up, such as achieving and saving a consistent system state
- **Deployment view**. If multiple processors or specialized hardware is used in a system, additional responsibilities may arise from deployment to the hardware. Using a deployment view helps to determine and design a deployment that supports achieving the desired qualities. The deployment view results in the virtual threads of the concurrency view being decomposed into virtual threads within a particular processor and messages that travel between processors to initiate the next entry in the sequence of actions. Thus, it is the basis for analyzing the network traffic and for determining potential congestion.

2.d Define Interfaces of the Child Modules

- For purposes of ADD, an interface of a module shows the services and properties provided and required. This is different from a signature. It documents what others can use and on what they can depend.

- Analyzing and documenting the decomposition in terms of structure (module decomposition view), dynamism (concurrency view), and runtime (deployment view) uncovers the interaction assumptions for the child modules, which should be documented in their interfaces. The module view documents
- producers/consumers of information.
- patterns of interaction that require modules to provide services and to use them.

The concurrency view documents

- interactions among threads that lead to the interface of a module providing or using a service.
- the information that a component is active?for example, has its own thread running.
- the information that a component synchronizes, sequentializes, and perhaps blocks calls.

The deployment view documents

- the hardware requirements, such as special-purpose hardware.
- some timing requirements, such as that the computation speed of a processor has to be at least 10 MIPS.
- communication requirements, such as that information should not be updated more than once every second.

2.e Verify and Refine Use Cases and Quality Scenarios as Constraints for the Child Modules

- The steps enumerated thus far amount to a proposal for a module decomposition. This decomposition must be verified and the child modules must be prepared for their own decomposition.
- Functional requirements
- Each child module has responsibilities that derive partially from considering decomposition of the functional requirements. Those responsibilities can be translated into use cases for the module. Another way of defining use cases is to split and refine the parent use cases
- In our example, the initial responsibilities for the garage door opener were to open and close the door on request, either locally or remotely; to stop the door within 0.1 second when an obstacle is detected; and to interact with the home information system and support remote diagnostics. The responsibilities are decomposed into the following functional groups corresponding to the modules:
- User interface. Recognize user requests and translate them into the form expected by the raising/lowering door module.
- Raising/lowering door module. Control actuators to raise or lower the door. Stop the

door when it reaches either fully open or fully closed.

- Obstacle detection. Recognize when an obstacle is detected and either stop the descent of the door or reverse it.
- Communication virtual machine. Manage all communication with the home information system.
- Sensor/actuator virtual machine. Manage all interactions with the sensors and actuators.
- Scheduler. Guarantee that the obstacle detector will meet its deadlines.
- Diagnosis. Manage the interactions with the home information system devoted to diagnosis.

Constraints- Constraints of the parent module can be satisfied in one of the following ways:

- **The decomposition satisfies the constraint**. For example, the constraint of using a certain operating system can be satisfied by defining the operating system as a child module. The constraint has been satisfied and nothing more needs to be done.
- **The constraint is satisfied by a single child module**. For example, the constraint of using a special protocol can be satisfied by defining an encapsulation child module for the protocol. The constraint has been designated a child. Whether it is satisfied or not depends on what happens with the decomposition of the child.
- **The constraint is satisfied by multiple child modules**. For example, using the Web requires two modules (client and server) to implement the necessary protocols. Whether the constraint is satisfied depends on the decomposition and coordination of the children to which the constraint has been assigned.

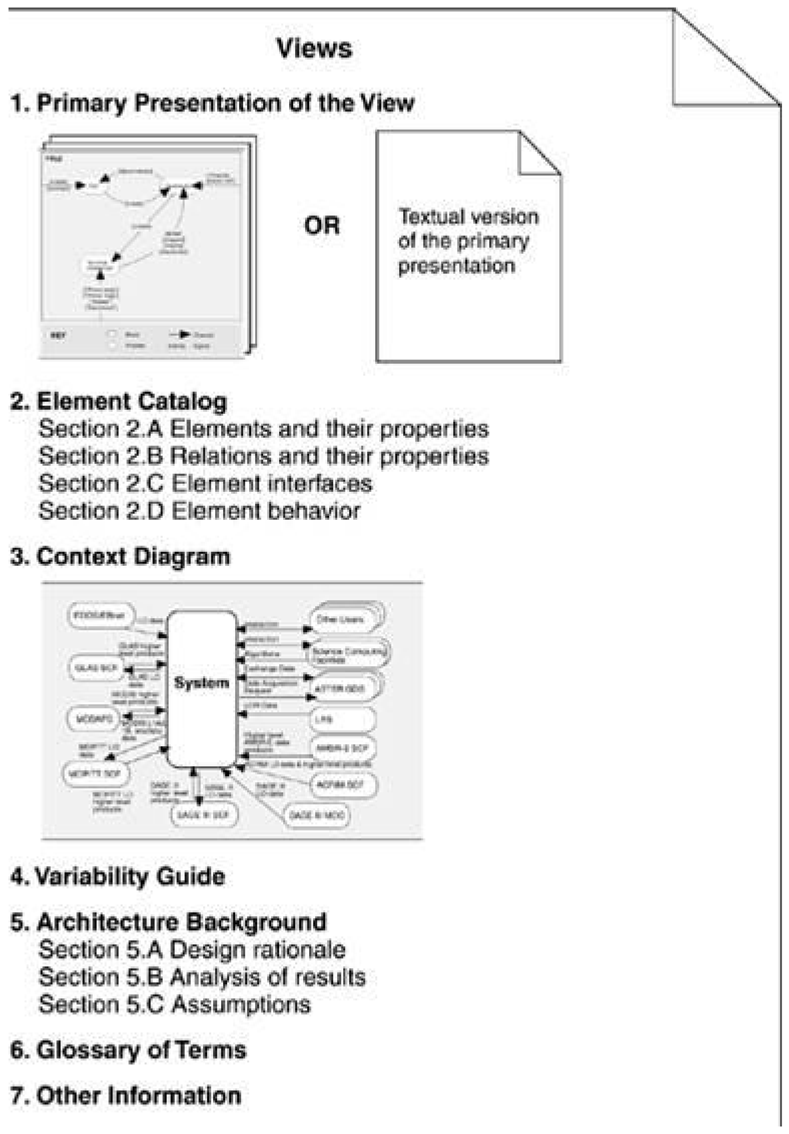
Quality scenarios- Quality scenarios also have to be refined and assigned to the child modules.

- A quality scenario may be completely satisfied by the decomposition without any additional impact. It can then be marked as satisfied.
- A quality scenario may be satisfied by the current decomposition with constraints on child modules.
- The decomposition may be neutral with respect to a quality scenario.
- A quality scenario may not be satisfiable with the current decomposition. If it is an important one, then the decomposition should be reconsidered. Otherwise, the rationale for the decomposition not supporting this scenario must be recorded.

At the end of this step we have a decomposition of a module into its children, where each child module has a collection of responsibilities; a set of use cases, an interface, quality scenarios, and a collection of constraints. This is sufficient to start the next iteration of decomposition.

7. a. Briefly explain the concept of documenting a view. [10]

Soln.



Views

1. Primary Presentation of the View

OR

Textual version of the primary presentation

2. Element Catalog
   Section 2.A Elements and their properties
   Section 2.B Relations and their properties
   Section 2.C Element interfaces
   Section 2.D Element behavior

3. Context Diagram

4. Variability Guide

5. Architecture Background
   Section 5.A Design rationale
   Section 5.B Analysis of results
   Section 5.C Assumptions

6. Glossary of Terms

7. Other Information

1. **Primary presentation** shows the elements and the relationships among them that populate the view. The primary presentation should contain the information you wish to convey about the system (in the vocabulary of that view) first. It should certainly include the primary elements and relations of the view, but under some circumstances it might not include all of them. For example, you may wish to show the elements and relations that come into play during normal operation, but relegate error handling or exceptional processing to the supporting documentation.

- The primary presentation is usually graphical. In fact, most graphical notations make their contributions in the form of the primary presentation and little else. If the primary presentation is graphical, it must be accompanied by a key that explains, or that points to an explanation of, the notation or symbology used.

- Sometimes the primary presentation can be tabular; tables are often a superb way to convey a large amount of information compactly. An example of a textual primary presentation is the A-7E module decomposition view illustrated in Chapter 3. A textual presentation still carries the obligation to present a terse summary of the most important information in the view. In Section 9.6 we will discuss using UML for the primary presentation.

2. **Element catalog** details at least those elements and relations depicted in the primary presentation, and perhaps others. Producing the primary presentation is often what architects concentrate on, but without backup information that explains the picture, it is of little value For instance, if a diagram shows elements A, B, and C, there had better be documentation that explains in sufficient detail what A, B, and C are, and their purposes or the roles they play, rendered in the vocabulary of the view. For example, a module decomposition view has elements that are modules, relations that are a form of "is part of," and properties that define the responsibilities of each module. A process view has elements that are processes, relations that define synchronization or other process-related interaction, and properties that include timing parameters.

3. **Context diagram** shows how the system depicted in the view relates to its environment in the vocabulary of the view. For example, in a component-and-connector view you show which component and connectors interact with external components and connectors, via which interfaces and protocols.

4. **Variability guide** shows how to exercise any variation points that are a part of the architecture shown in this view. In some architectures, decisions are left unbound until a later stage of the development process, and yet the architecture must still be documented.

- the options among which a choice is to be made. In a module view, the options are the various versions or parameterizations of modules. In a component-and-connector view, they might include constraints on replication, scheduling, or choice of protocol. In an allocation view, they might include the conditions under which a software element would be allocated to a particular processor.

- the binding time of the option. Some choices are made at design time, some at build time, and others at runtime.

5. **Architecture background** explains why the design reflected in the view came to be. The goal of this section is to explain to someone why the design is as it is and to provide a convincing argument that it is sound. An architecture background includes

- rationale, explaining why the decisions reflected in the view were made and why alternatives were rejected.

- analysis results, which justify the design or explain what would have to change in the face of a modification.

- assumptions reflected in the design.

6. **Glossary of terms** used in the views, with a brief description of each.

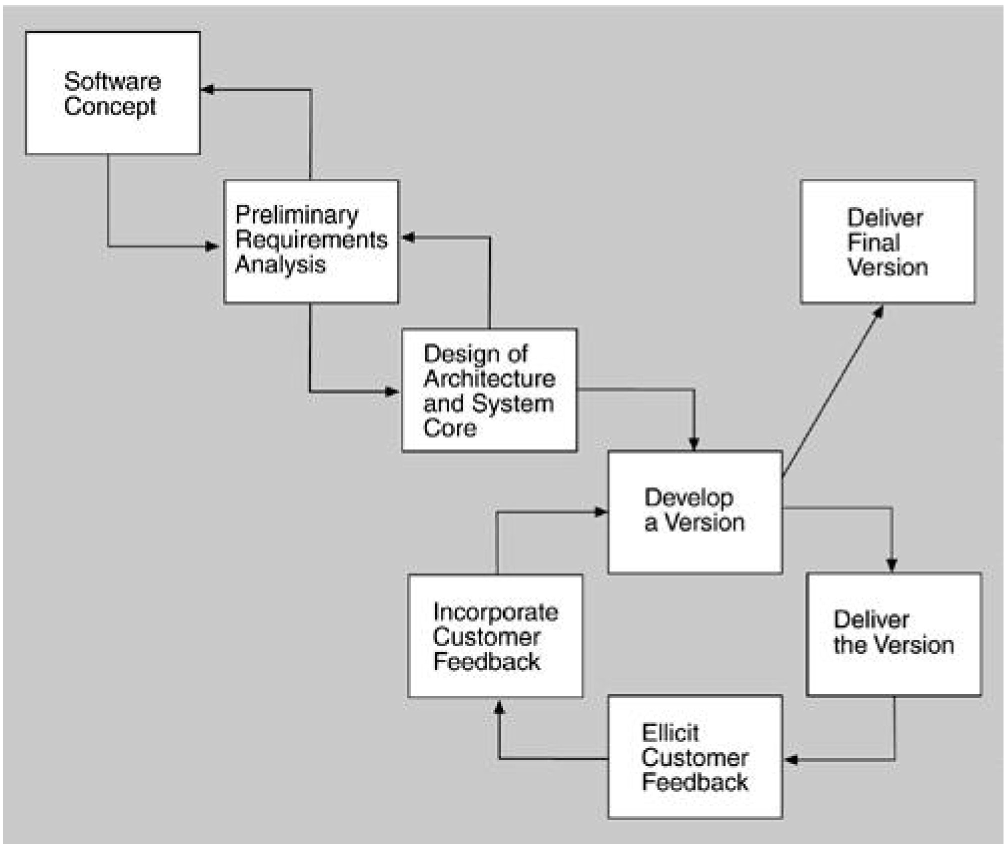
7. **Other information**. The precise contents of this section will vary according to the standard practices of your organization. They might include management information such as authorship, configuration control data, and change histories. Or the architect might record references to specific sections of a requirements document to establish traceability. Strictly speaking, information such as this is not architectural. Nevertheless,

it is convenient to record it alongside the architecture, and this section is provided for that purpose. In any case, the first part of this section must detail its specific contents.

8. a. Explain with a neat diagram, the evolutionary delivery life cycle model. [5]

Soln.

- 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
- 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.
- The life-cycle model shows the design of the architecture as iterating with preliminary requirements analysis. 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
- 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. From this list, choose the ones that will have the most impact on the architecture. These are the architectural drivers, and there should be fewer than ten.
- Once the architectural drivers are known, the architectural design can begin. The requirements analysis process will then be influenced by the questions generated during architectural design?one of the reverse-direction arrows shown in figure


Software Concept
Preliminary Requirements Analysis
Deliver Final Version
Design of Architecture and System Core
Develop a Version
Incorporate Customer Feedback
Deliver the Version
Ellicit Customer Feedback

b. Write a note on view catalog. [5]

Soln.

- A view catalog is the reader's introduction to the views that the architect has chosen to include in the suite of documentation.
- When using the documentation suite as a basis for communication, it is necessary for a new reader to determine where particular information can be found. A catalog contains this information. When using the documentation suite as a basis for analysis, it is necessary to know which views contain the information necessary for a particular analysis. In a performance analysis, for example, resource consumption is an important piece of information, A catalog enables the analyst to determine which views contain properties relevant to resource consumption.
- There is one entry in the view catalog for each view given in the documentation suite. Each entry should give the following:
  1. The name of the view and what style it instantiates
  2. A description of the view's element types, relation types, and properties
  3. A description of what the view is for
  4. Management information about the view document, such as the latest version, the location of the view document, and the owner of the view document
- The view catalog is intended to describe the documentation suite, not the system being documented. Specifics of the system belong in the individual views, not in the view catalog. For instance, the actual elements contained in a view are listed in the view's element catalog.

*********************************All the best**