USN | | | | | | | | | | |

Internal Assessment Test 1 – March 2018

Sub:	Software Architectures							Code:	10IS81
Date:	12-03-18	Duration:	90 mins	Max Marks:	50	Sem:	VIII	Branch:	CSE A, B

Note: Answer any 5 questions. All questions carry equal marks. Total marks: 50

		Marks	OBE	
			CO	RBT
1.	Define Software Architecture. Explain ABC (Architecture Business Cycle) with the help of a neat block diagram? List the software activities used in architecture development.	[10]	CO1	L2
2.	What are the factors which affect the influence of software architecture? Explain with neat diagram.	[10]	CO1	L2
3.	Enumerate and Explain in detail with supporting diagrams, the different categories of software architecture structures.	[10]	CO3	L2
4.	With the neat diagram, explain the relationships of Reference models, Architectural patterns, Reference architectures and Software architecture.	[10]	CO2	L2

USN | | | | | | | | | | |

Internal Assessment Test 1 – March 2018

Sub:	Software Architectures							Code:	10IS81
Date:	12-03-18	Duration:	90 mins	Max Marks:	50	Sem:	VIII	Branch:	CSE A, B

Note: Answer any 5 questions. All questions carry equal marks. Total marks: 50

		Marks	OBE	
			CO	RBT
1.	Define Software Architecture. Explain ABC (Architecture Business Cycle) with the help of a neat block diagram? List the software activities used in architecture development.	[10]	CO1	L2

2.	What are the factors which affect the influence of software architecture? Explain with neat diagram.	[10]	CO1	L2
3.	Enumerate and Explain in detail with supporting diagrams, the different categories of software architecture structures.	[10]	CO3	L2
5.	a. What are Functional and Non-functional requirements in Software architecture?	[04]	CO1	L3
	b. Explain Quality attribute scenario with a neat diagram.	[06]	CO2	L2
6.	Explain in brief summary of Availability tactic with neat diagram.	[10]	CO2	L3
7.	Distinguish between Security, testability and Usability quality attributes scenario.	[10]	CO2	L3
8.	Explain the following with respect to Tactics: i) Defer Binding Time ii) Resource Arbitration iii) Input/output (w.r.t. testability)	[3+4+3]	CO2	L3
4.	With the neat diagram, explain the relationships of Reference models, Architectural patterns, Reference architectures and Software architecture.	[10]	CO2	L2

5.	a. What are Functional and Non-functional requirements in Software architecture?	[04]	CO1	L3
	b. Explain Quality attribute scenario with a neat diagram.	[06]	CO2	L2
6.	Explain in brief summary of Availability tactic with neat diagram.	[10]	CO2	L3
7.	Distinguish between Security, testability and Usability quality attributes scenario.	[10]	CO2	L3
8.	Explain the following with respect to Tactics: i) Defer Binding Time ii) Resource Arbitration iii) Input/output (w.r.t. testability)	[3+4+3]	CO2	L3

Scheme and solution

Question #	Description	Marks Distribution		Max Marks
1	Define Software Architecture. Explain ABC (Architecture Business Cycle) block diagram List the software activities used in architecture development.	2M 4M 2M 2M	10M	10M
2	factors which affect the influence of software architecture Explanation Neat diagram.	3M 5M 2M	10M	10M
3	Categories of software architecture structures with diagram.	10M	10M	10M
4	Relationships of Reference models, Architectural patterns, Reference architectures and Software architecture.	2.5M 2.5M 2.5M 2.5M	10M	10M
5 a	What are Functional and Non-functional requirements in Software architecture?	2M 2M	4M	4M
5 b	Quality attribute scenario neat diagram.	4M 2M	6M	6M
6	Summary of Availability tactic neat diagram.	8 M 2 M	10M	10M
7	Distinguish between Security, testability and Usability quality attributes scenario.	4M 3M 3M	10M	10M

8	Explain the following with respect to Tactics: i) Defer Binding Time ii) Resource Arbitration iii) Input/output (w.r.t. testability)	3M 4M 3M	10M	10M

SOLUTION

1>

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 1.4 shows the feedback loops. Some of the feedback comes from the architecture itself, and some comes from the system built from it.

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.

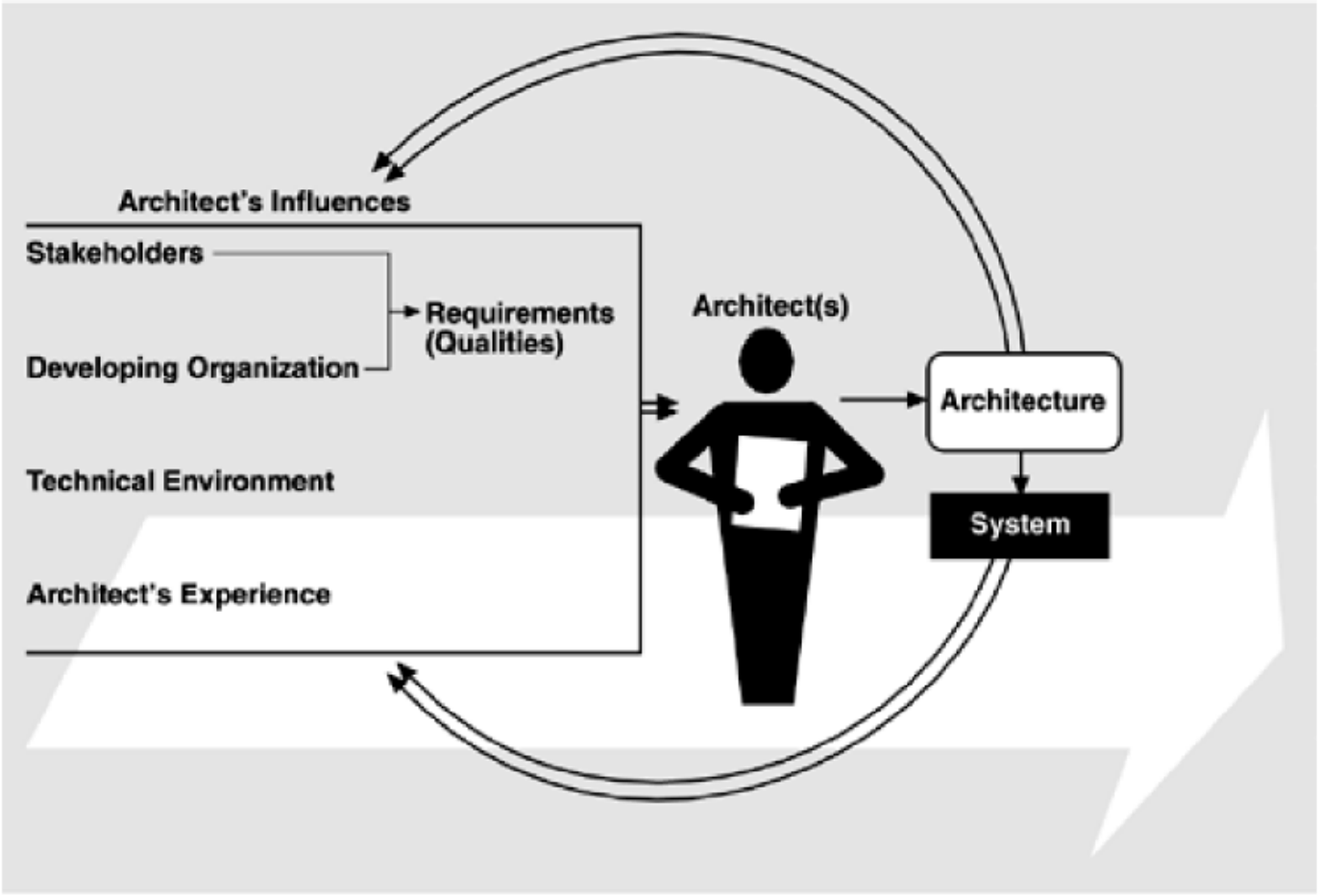
2) The architecture can affect the goals of the developing organization. A successful system built from it can enable a company to establish a foothold in a particular market area. The architecture can provide opportunities for the efficient production and deployment of the similar systems, and the organization 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.

Figure 1.4. The Architecture Business Cycle



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

 Creating the business case for the system

o It is an important step in creating and constraining any future requirements.
o How much should the product cost?
o What is its targeted market?
o What is its targeted time to market?
o Will it need to interface with other systems?
o Are there system limitations that it must work within?
o These are all the questions that must involve the system's architects.
o 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

o There are a variety of techniques for eliciting requirements from the stakeholders.
o 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.
o Another technique that helps us understand requirements is the creation of prototypes.
o 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

o 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

o 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.
o 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

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

 Implementing the system based on the architecture

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

 Ensuring that the implementation conforms to the architecture

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

2>**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".

 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 1.3.

 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.

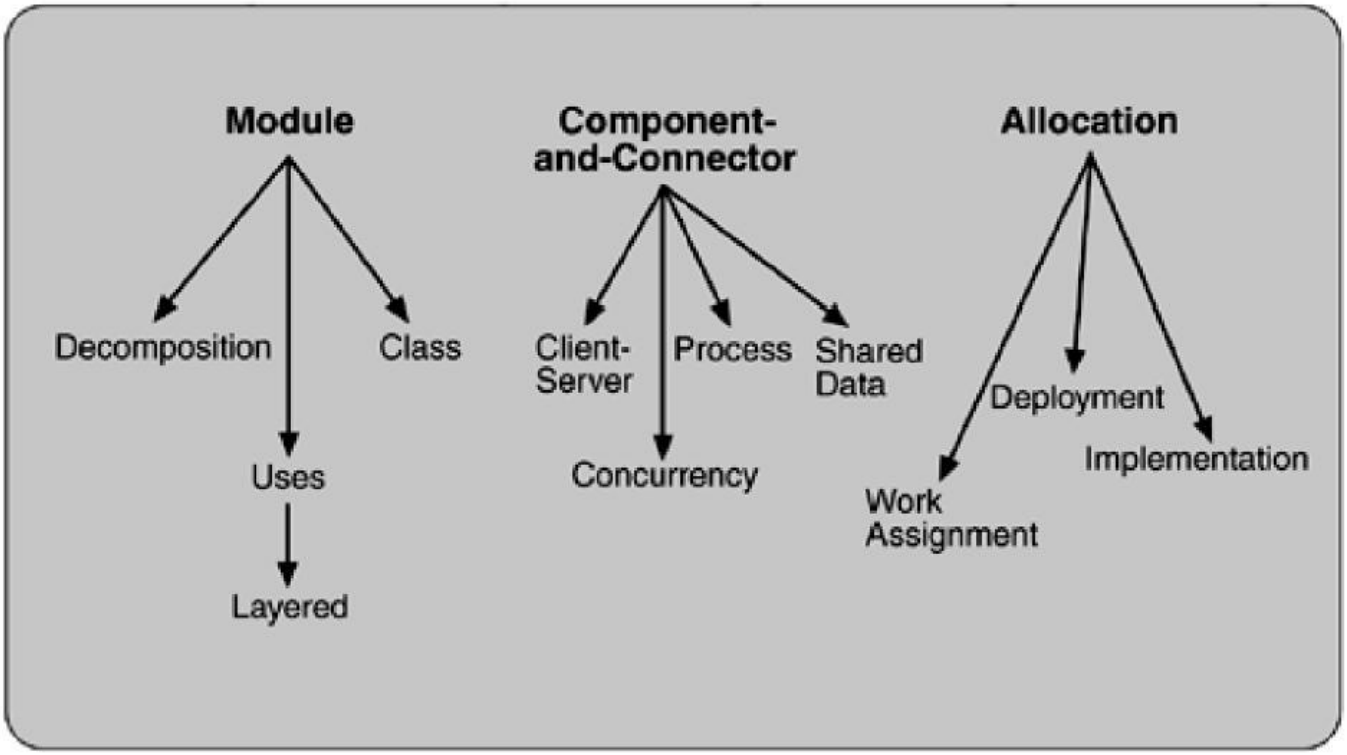
3> Architectural structures can by and large be divided into three groups, depending on the broad nature of the elements they show.
 Module structures.

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?
 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 What processor does each software element execute on? In what files is each element stored during development, testing, and system building? What is the assignment of software elements to development teams?

Figure 2-3. Common software architecture structures



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

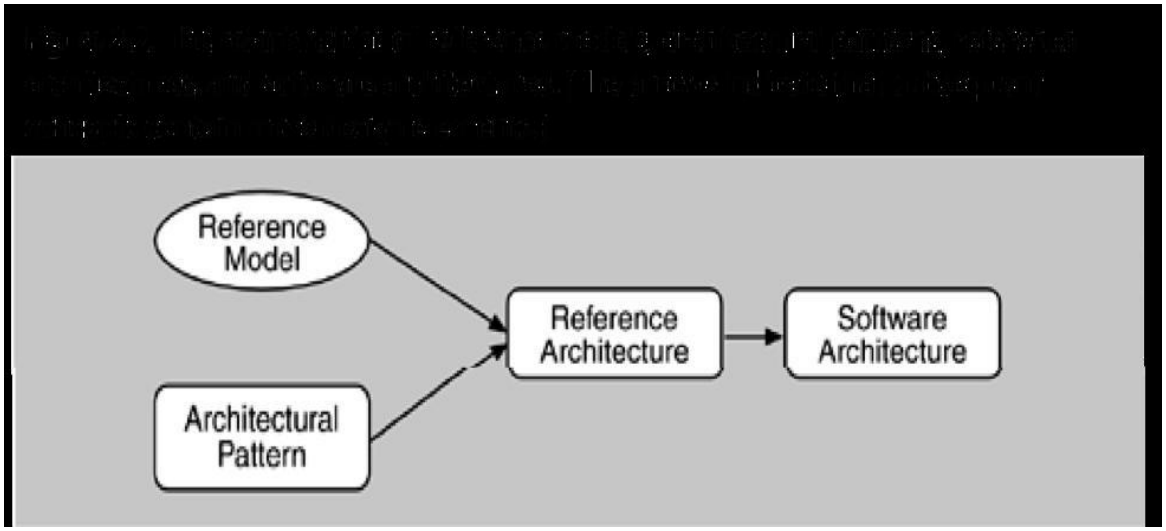
 Deployment: This view allows an engineer to reason about performance, data integrity, availability, and security

 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.

4>**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. **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. **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.

Reference models, architectural patterns, and reference architectures are not architectures; they are useful concepts that capture elements of an architecture. Each is the outcome of early design decisions. The relationship among these design elements is shown in Figure 2.2. 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.



5 a>

Functionality: It is the ability of the system to do the work for which it was intended. A task requires that many or most of the system's elements work in a coordinated manner to complete the job, just as framers, electricians, plumbers, drywall hangers, painters, and finish carpenters all come together to cooperatively build a house. **Software architecture** constrains its allocation to structure when *other* quality attributes are important.

5 b>

QUALITY ATTRIBUTE SCENARIOS

A quality attribute scenario is a quality-attribute-specific requirement. It consists of six parts.

1) ***Source of stimulus***. This is some entity (a human, a computer system, or any other actuator) that generated the stimulus.

2) ***Stimulus***. The stimulus is a condition that needs to be considered when it arrives at a system.

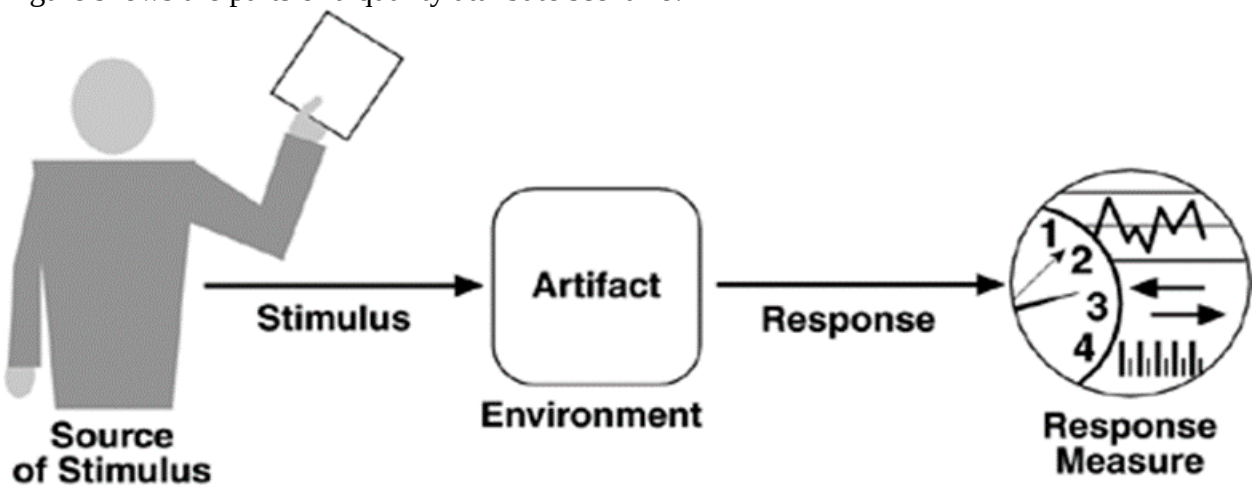
3) ***Environment.*** The stimulus occurs within certain conditions. The system may be in an overload condition or may be running when the stimulus occurs, or some other condition may be true.

4) ***Artifact***. Some artifact is stimulated. This may be the whole system or some pieces of it.

5) ***Response***. The response is the activity undertaken after the arrival of the stimulus.

6) ***Response measure.*** When the response occurs, it should be measurable in some fashion so that the requirement can be tested.

Figure shows the parts of a quality attribute scenario.



6>

QUALITY ATTRIBUTE SCENARIOS IN PRACTICE

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

Source of stimulus.

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

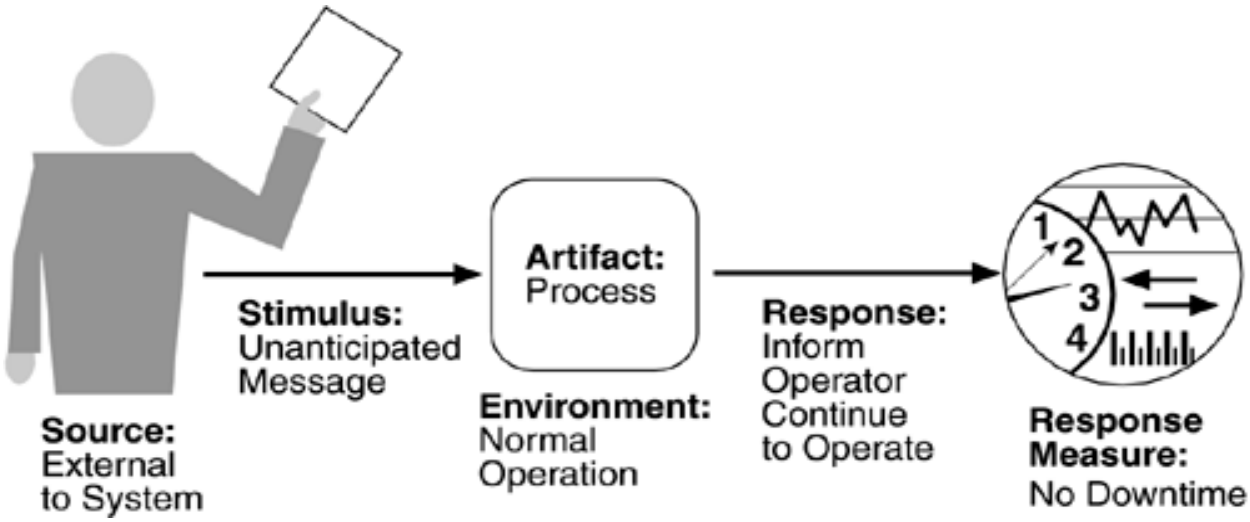
Stimulus. A fault of one of the following classes occurs. - *omission.* A component fails to respond to an input. - *crash.* The component repeatedly suffers omission faults. - *timing.* A component responds but the response is early or late. - *response.* A component responds with an incorrect value.

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 either less capacity or less function, shutting down external systems, or becoming unavailable during repair. In our 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



7>

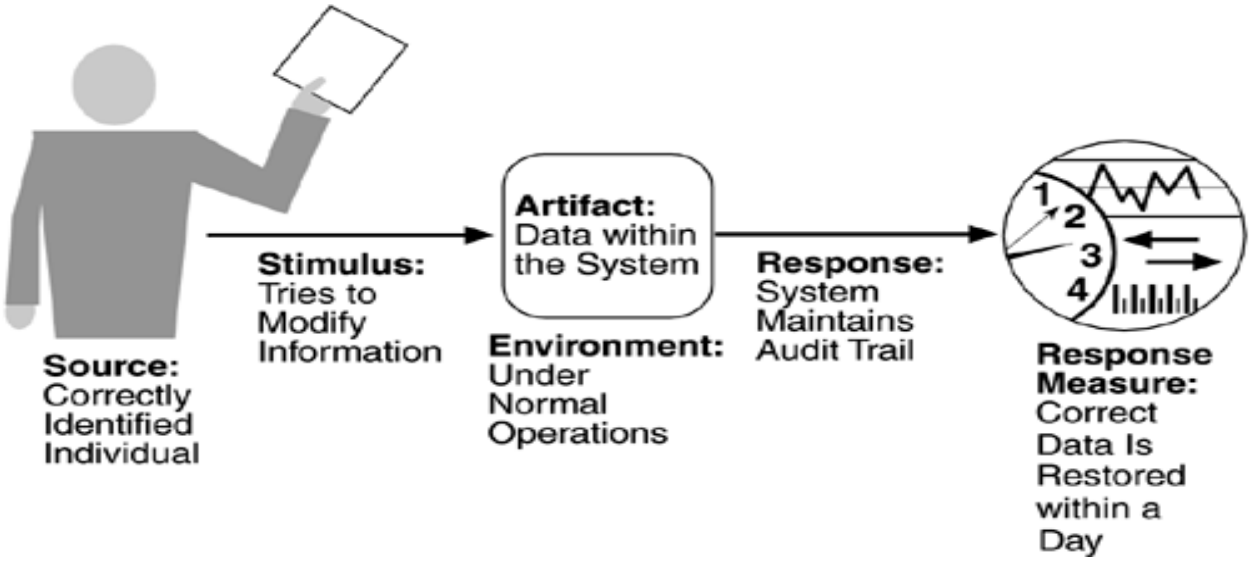
SECURITY SCENARIO

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. Security can be characterized as a system providing non-repudiation, confidentiality, integrity, assurance, availability, and auditing. For each term, we provide a definition and an example.

 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.

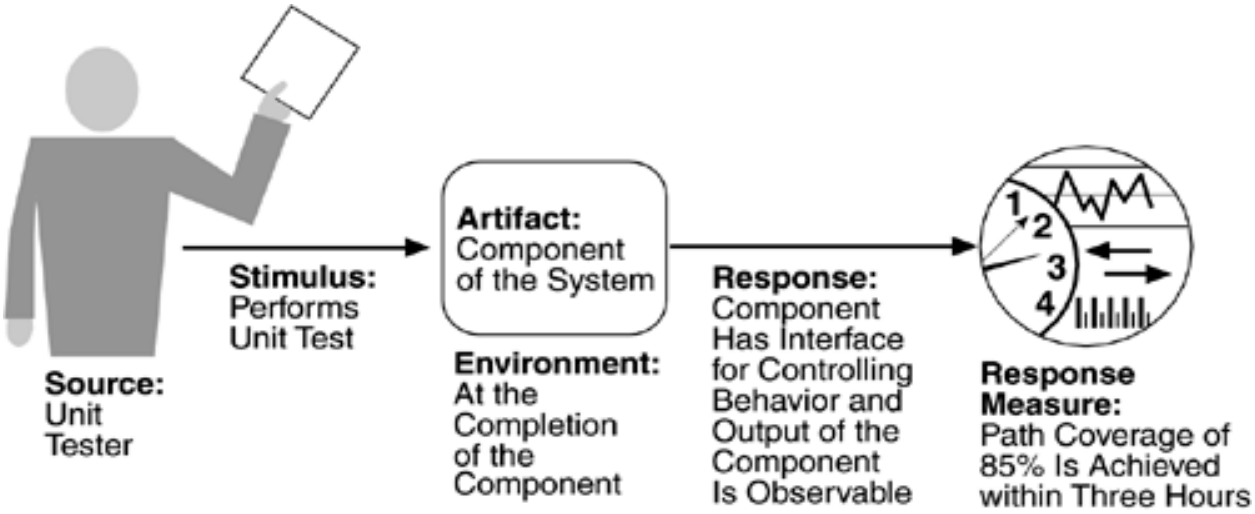
 Confidentiality is the property that data or services are protected from unauthorized access. This means that a hacker cannot access your income tax returns on a government computer.

 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.
 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.
 Availability is the property that the system will be available for legitimate use. This means that a denial-of-service attack won't prevent your ordering *this* book.
 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.



Each of these security categories gives rise to a collection of general scenarios.
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. ***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, 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
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.
TESTABILITY SCENARIO:
Software testability refers to the ease with which software can be made to demonstrate its faults through testing. In particular, testability refers to the probability, assuming that the software has at least one fault that it will fail on its *next* test execution. Testing is done by various developers, testers, verifiers, or users and is the last step of various parts of the software life cycle. Portions of the code, the design, or the complete system may be tested.



Source of stimulus. The testing is performed by unit testers, integration testers, system testers, or the client. A test of the design may be performed by other developers or by an external group. In our example, the testing is performed by a tester.

Stimulus. The stimulus for the testing is that a milestone in the development process is met. This might be the completion of an analysis or design increment, the completion of a coding increment such as a class, the completed integration of a subsystem, or the completion of the whole system. In our example, the testing is triggered by the completion of a unit of code.

Artifact. A design, a piece of code, or the whole system is the artifact being tested. In our example, a unit of code is to be tested.

Environment. The test can happen at design time, at development time, at compile time, or at deployment time. In Figure, the test occurs during development.

Response. Since testability is related to observability and controllability, the desired response is that the system can be controlled to perform the desired tests and that the response to each test can be observed. In our example, the unit can be controlled and its responses captured.

Response measure. Response measures are the percentage of statements that have been executed in some test, the length of the longest test chain (a measure of the difficulty of performing the tests), and estimates of the probability of finding additional faults. In Figure, the measurement is percentage coverage of executable statements.

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:

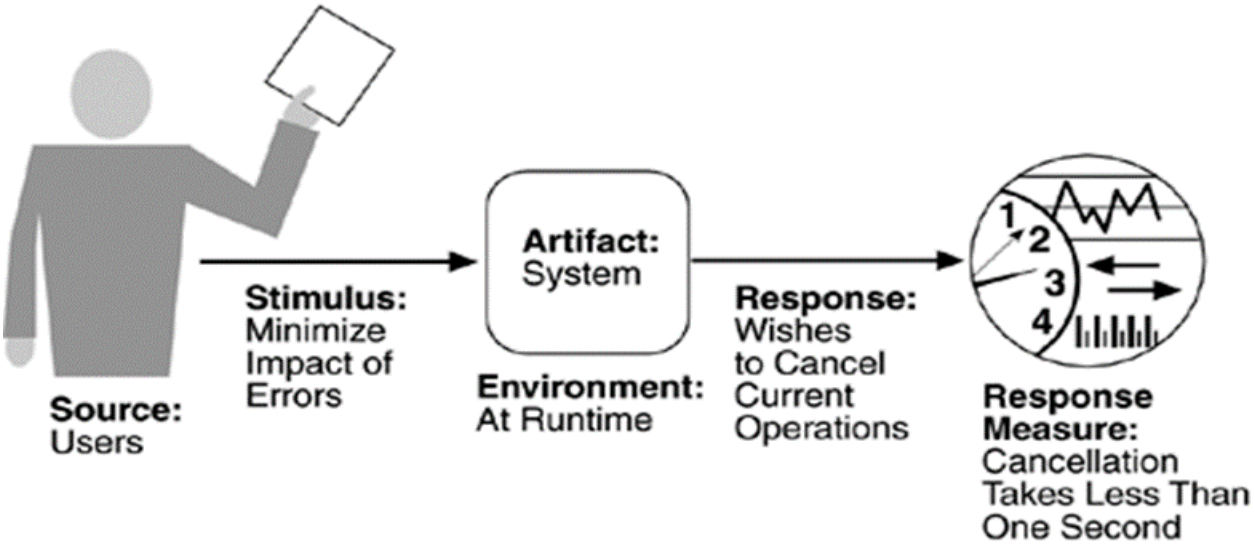
 Learning system features. If the user is unfamiliar with a particular system or a particular aspect of it, what can the system do to make the task of learning easier?

 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. 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.

8>
DEFER BINDING TIME Many tactics are intended to have impact at loadtime 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.

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

o *semantic importance.* Each stream is assigned a priority statically according to some domain characteristic of the task that generates it.
o *deadline monotonic.* Deadline monotonic is a static priority assignment that assigns higher priority to streams with shorter deadlines.
o *rate monotonic.* Rate monotonic is a static priority assignment for periodic streams that assigns higher priority to streams with shorter periods.
Dynamic priority scheduling:
o *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.
o *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.

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.