SOFTWARE ARCHITECTURE AND DESIGN PATTERNS

UNIT-I

Dr. T. K. RAO

VVIT

OBJECTIVES:

The course should enable the student to understand :

- interrelationships, principles and guidelines governing architecture and evolution over time
- various architectural styles of software systems
- design patterns and their underlying object oriented concepts
- implementation of design patterns and providing solutions to real world software design problems
- patterns with each other and understanding the consequences of combining patterns on the overall quality of a system

TEXT BOOKS:

- Software Architecture in Practice, second edition, Len Bass, Paul Clements & Rick Kazman, Pearson Education, 2003.
- Design Patterns, Erich Gamma, Pearson Education, 1995.

Contents

ENVISIONING ARCHITECTURE:

- **Architecture Business Cycle**
- **What is Software Architecture**
- **Architectural patterns**
- **Reference models**
- **Reference architectures**
- **Architectural structures and views**

CREATING AN ARCHITECTURE:

- **Quality Attributes**
- **Achieving qualities**
- **Architectural patterns & Styles**
- **Designing the Architecture**
- **Documenting s/w architectures**
- **Reconstructing Software Architecture**

The Architecture Business Cycle

- Definition:
  - The structure of the computing system which comprise s/w elements, the externally visible properties of those elements and the relationships among them
- s/w architecture is a result of technical, business and social influences
- Its existence in turn affects the technical, business and social environments that subsequently influences future architectures
- This cycle is **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 in this design
- Realization of the influences will change in which the architecture is required to perform
- Many people are connected with s/w development
- E.g. customer, end users, developers, project managers, maintainers, marketing agents, etc.
- Architect receives helpful suggestions from them
- An acceptable system involves the properties like:
  - performance, reliability, availability, platform compatibility, memory utilization, n/w usage, security, modifiability, usability, interoperability, etc.

- Architectures are influenced by:
  - The developing organization
    - Immediate business, long-term business, organizational structure
  - Background and experience of the architects
  - Technical environment
- Architectures affect the factors that influence them
- Architecture affects:
  - the structure of the developing organization
  - the goals of the developing organization
  - customer requirements for next system
- The process of system building will affect the architect’s experience with subsequent systems
- Few systems will influence and actually change the SE culture

Software processes and ABC

- s/w process is the term given to the organization, ritualization, and management of s/w development activities
- Following are the activities involved
  - Creating the business case for the system
  - Understanding the requirements
  - Creating or selecting the architecture
  - Documenting and communicating the architecture
  - Analyzing and evaluating the architecture
  - Implementing the system based on the architecture
  - Ensuring that the implementation conforms the architecture

What is Software Architecture

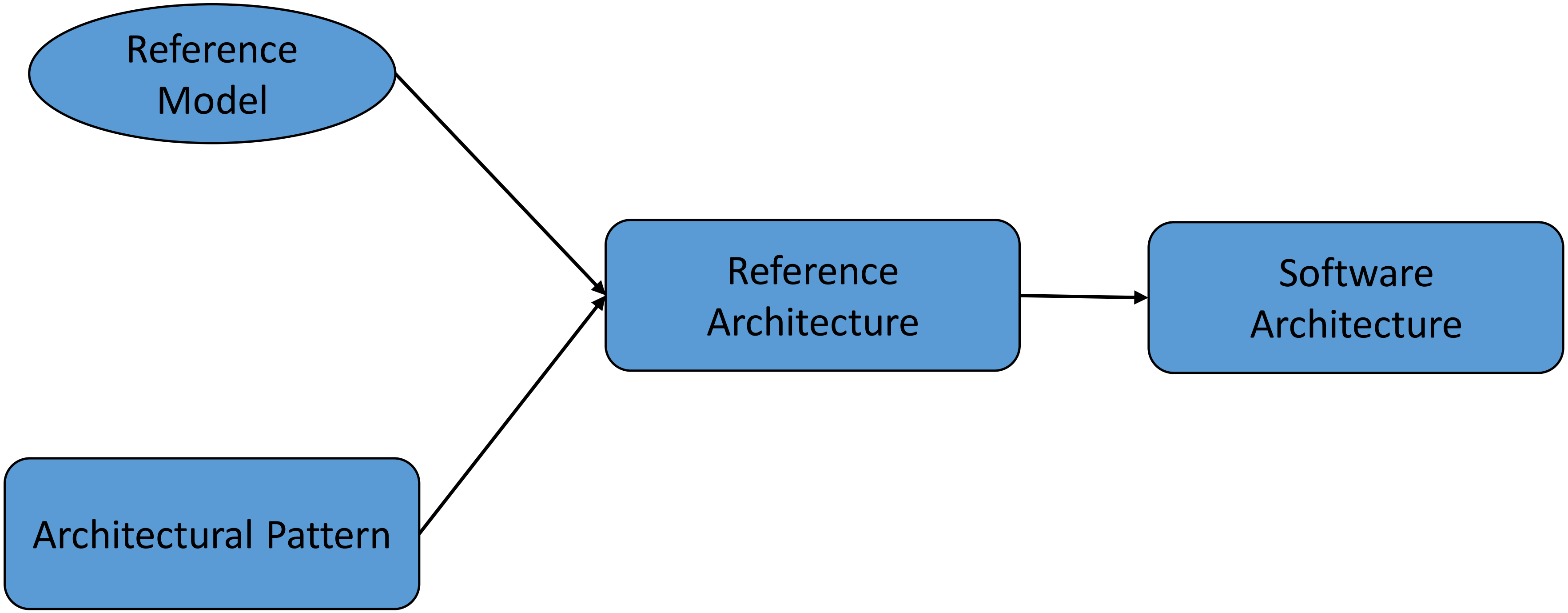
- First, architecture define s/w elements
  - It shapes info about how the elements relate with each other
  - It is an abstraction of a system that suppresses details of elements that do not effect how they use
- Second, systems can have more than one structure and no one structure can irrefutably claim to be the architecture
- Third, every system with s/w has a s/w architecture
- Fourth, behavior of each element is part of the architecture

Architectural patterns, reference models and reference architectures

VASIREDDY VENKATADRI INSTITUTE OF TECHNOLOGY

- Architectural pattern is a description of element & relation types together with a set of constraints on how they may be used
  - A pattern is a set of constraints on an architecture
  - E.g. Client-Server Architecture
- Reference model is a division of functionality together with data flow between the pieces
  - Reference model is a standard decomposition of a problem into parts that cooperatively solve the problem
  - E.g. DBMS, Compiler Design, etc.

- A reference architecture is a reference model mapped onto s/w elements
  - A reference model divides the functionality, a reference architecture is the mapping of that functionality onto a system decomposition
  - A s/w element may implement part of a function or several functions
  - Possible to define at many levels of detail or abstraction
  - Generally, will not completely specify all the technologies, components and their relationships in sufficient detail to enable direct implementation
- 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.


Reference Model
Architectural Pattern
Reference Architecture
Software Architecture

Why is s/w architecture important

There are three fundamental reasons

- **Communication among stakeholders:**
  - SA represents a common abstraction of a system
  - Stakeholders can use as basis for mutual understanding, communication
- **Early design decisions:**
  - SA manifests the earliest design decisions about a system
  - Design decisions governing the system to be built can be analyzed
- **Transferable abstraction of a system:**
  - SA constitutes a relatively small, intellectually graspable model for how a system is structured and works
  - Can promote large scale reuse

Communication among stakeholders

- User need: system is reliable and available when needed
- Customer need: implemented on schedule and budget
- Manager need: system should allow teams to work largely independently, interacting in disciplined and controlled ways
- Architect need: strategies to achieve all of those goals

Early design decisions

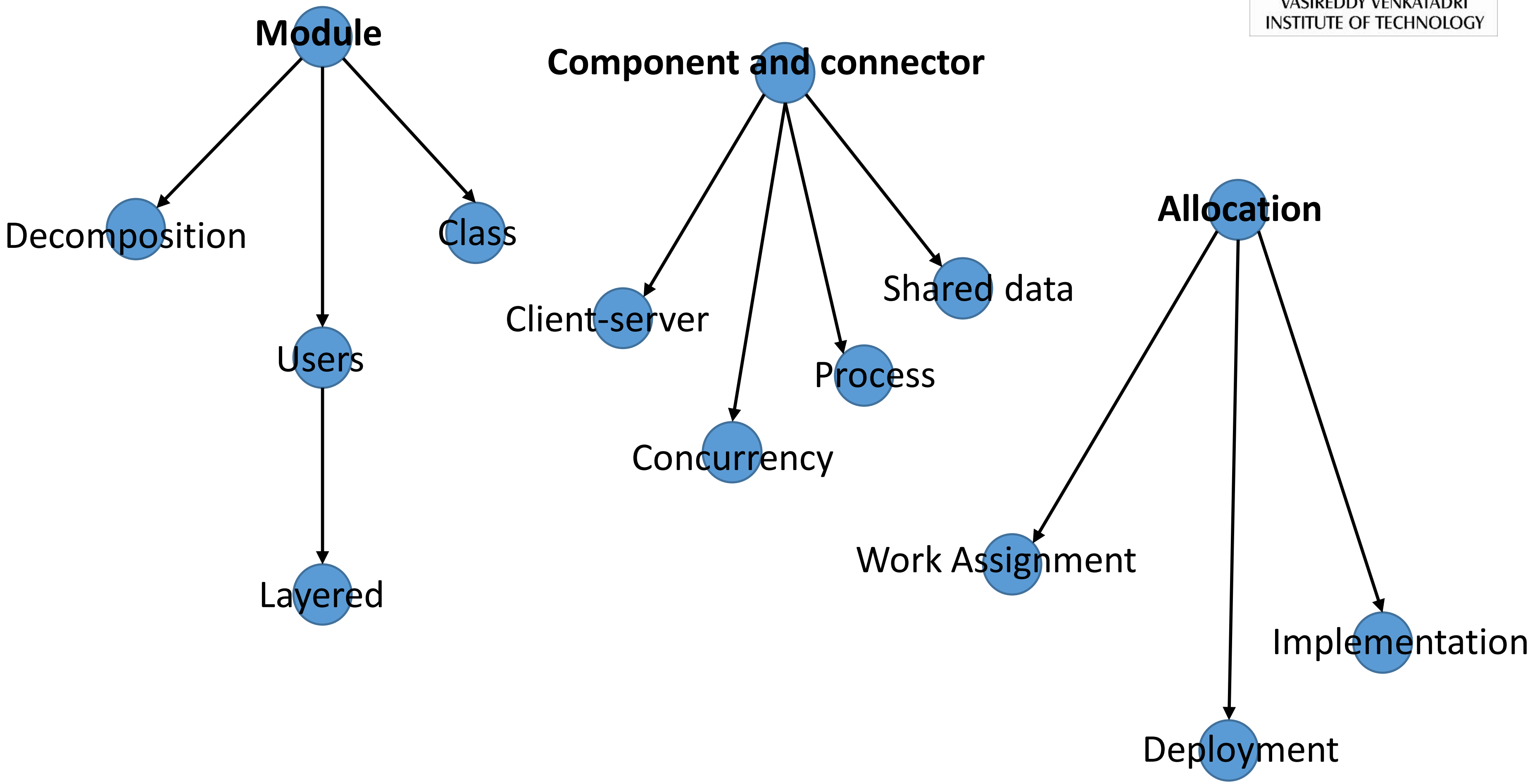
- Early decisions are most difficult to get correct and the hardest to change later
  - Architecture defines constraints on implementation
  - Architecture dictates organizational structure
  - Architecture inhibits or enables a system’s quality attributes
  - Predicting system qualities by studying the architecture
  - Architecture makes it easier to reason about and manage change
  - Architecture helps in evolutionary prototyping
  - Architecture enables more accurate cost and schedule estimates

A transferable and reusable model

- Reuse at the Architectural level provides tremendous leverage for systems with similar needs
  - s/w 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
  - Architecture permits template based development
  - Architecture can be the basis for training

Architectural structures & views

- Structure is the set of all elements
- View is a representation of elements & their relations
- These restrict our attention at any one moment to one (or a small number) of the software system's structures
- Broadly, Architectural Structures can be divided as:
  - **Module structures**
  - **Component and connector structures**
  - **Allocation structures**



Common S/w Architecture Structures

Module structures

- Elements are modules, which are units of implementation
  - What is the primary functional responsibility assigned to each module?
  - What other software elements is a module allowed to use?
  - What other software does it actually use?
- Module structures include:
  - **Decomposition**
  - **Uses**
  - **Layered**
  - **Class or generalization**

- **Decomposition:** the units are modules, related to each other by the ‘is a sub-module of’ relationship
- This shows how larger modules decomposed into smaller ones recursively until they are small enough
- Modules in this structure represent a common starting point for design
- **Uses:** the units are: modules procedures or resources on the interfaces of modules
- The units are related by the ‘uses’ relation
- Uses structure is used to engineer systems that can be easily extended to add functionality

- **Layered:** when the ‘uses’ relations are controlled in a particular way, a system of layers emerges
- In a strictly layered structure, layer ‘n’ may only use the services of layer n-1
- **Class or generalization:** the module units in this structure are called class
- Relation ‘inherits from’ or ‘is-an-instance-of’ is used
- This view supports reasoning about collections of similar behavior
- The class structure allows us to reason about re-use and the incremental addition of functionality

Component and connector structures

- Elements are runtime components and connectors
- The relation is attachment, showing how the components and connectors are hooked together
  * What are the major executing components and how do they interact?
  * What are the major shared data stores?
  * Which parts of the system are replicated?
  * How does data progress through the system?
  * What parts of the system can run in parallel?
- This structure includes:
  - Process or communicating processes
  - Concurrency
  - Shared data or repository
  - Client-server

- **Process or communicating processes:** this is orthogonal to the module-based structures
- The units are processes or threads
- The relation in this is ‘attachment’, showing how components and connectors are hooked together
- **Concurrency:** this structure allows the architect to determine opportunities for parallelism and the locations where resources conflictions may occur

- **Shared data or repository:** this structure comprises components and connectors that create, store, and access stable data
- It shows how data is produced and consumed by runtime s/w elements
- **Client-server:** if the system is built as a group of cooperating clients and servers, this is a good component-and-connector structure
- The components are the clients and servers and the connectors are protocols

Allocation structures

- Allocation structures include:
    - Deployment
    - Implementation
    - Work assignment
- **Deployment:** Deployment structure shows how s/w is assigned to h/w processing and communication elements
- Relations are:
    - ‘allocated-to’: showing on which physical units the s/w elements reside and
    - ‘migrates to’: if the allocation is dynamic

- **Implementation:** this structure shows how s/w elements (modules) are mapped to the file structure
- **Work assignment:** this structure assigns responsibility for implementing and integrating the modules
- The architect will know the expertise required for each team
- This means, calling out units of functional commonality and assigning them to a single team

Understanding Quality Attributes

- Understand how to express the qualities we want our architecture to provide to the system
- Discuss the relationship between quality attributes and s/w architecture
- Here are the few attributes:
  - Functionality and architecture
  - Architecture and quality attributes
  - System quality attributes
  - Quality attribute scenarios in practice
  - Other system quality attributes
  - Business qualities
  - Architecture qualities

Functionality and architecture

- Functionality is the ability of the system to do the work for which it was intended
- If the elements have not been assigned correct responsibility, resources, the system cannot perform
- E.g. systems are frequently divided so that several people can cooperatively build them

Architecture and quality attributes

- Achieving quality attributes must be considered through out the design, implementation and deployment
- **Usability** involves both architectural & non-architectural (e.g. user friendly GUI, radio button, etc.) aspects
- **Modifiability** is determined by how functionality is divided (archl.) and by coding techniques within a module (N-A)
- Performance involves both architectural & non-architectural dependencies
- It depends partially on:
  - how much communication is needed among components
  - what functionality has been allocated to each component (archl.)
  - how shared resources are allocated (archl.)
  - choice of algorithms to implement selected functionality (n-A)
  - how these algorithms are coded (n-A)

System quality attribute

- It consists of six parts:
  - **Source of stimulus:** an entity, e.g. human that generate stimulus
  - **Stimulus:** a condition, should be considered when it arrives at a system
  - **Environment:** the stimulus occurs under few conditions, e.g. overload
  - **Artifact:** this may be the whole system or some part of it
  - **Response:** the activity undertaken after the arrival of the stimulus
  - **Response measure:** when the response occurs, it should be measurable so that the requirement can be tested

Quality attribute scenarios in practice

There are six important systems quality attributes

- **Availability:** the availability of a system is the probability that it will be operational in need

$$\text{Def: } \alpha = \frac{(\text{mean time of failure})}{(\text{mean time to failure + mean time to repair})}$$

- **Modifiability:** is about cost of change, brings 2 concerns,
  - **what can change:** a change can occur to any aspect of a system, most commonly the functions that the system computes, the h/w, OS, protocols, etc.
  - **when and who makes change:** change was made to code by developer, but also by user, system admin

- **Performance:** Performance is about timing.
  - Events, e.g. interrupts, messages, requests, etc. occur, and the system must respond to them
  - One thing that make performance complex is, no.of event sources and arrivals (users, other sys., within)
- **Security:** is a measure of the system's ability to resist unauthorized usage
  - An attempt to breach security is called attack and can take no.of forms, e.g. accessing/modifying/deleting/ data theft etc.
  - Security means, a system providing approval, confidentiality, integrity, assurance, availability, and auditing

- **Testability:** ease with which s/w can be made to demonstrate its faults through testing
  - It must be possible to control each component’s internal state and i/ps and then to observe its o/ps
  - Testing is done by various developers, testers, or users and is the last step of various parts of SDLC
- **Usability:** how easy for the user to provide a desired task and the kind of user support it gives

  It can be classified into following areas:
  - Learning system features
  - Using a system efficiently
  - Minimizing the impact of errors
  - Adapting the system to user needs
  - Increasing confidence and satisfaction

Other system quality attributes

VASIREDDY VENKATADRI INSTITUTE OF TECHNOLOGY

- A number of other attributes can be found in the attribute taxonomies
- e.g. scalability (modifying system capacity-no.of users supported),
- portability (a platform modification),
- interoperability (create own general scenario)

Business qualities

- following are the qualities:
  - **Time to market:** if there is a competitive pressure for a product, development time becomes important
  - The ability to insert 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
  - An architecture that is highly flexible will typically be more costly to build
  - Projected lifetime of the system: if the system is intended to have a long lifetime, modifiability, scalability, and portability become important

- **Targeted market:** for general-purpose s/w, the platforms on which a system runs as well as its feature set will determine the size of the potential market
- Other qualities like performance, reliability, and usability play a key role
- **Rollout schedule:** if a product is to be introduced as base functionality with many features to be released later, the flexibility of the architecture is important
- Particularly the system must be developed with ease of expansion 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

- There are three qualities:
  - **Conceptual integrity:** the underlying theme or vision that unifies the design of the system at all levels
  - To evaluate the architecture, architect must be available
  - If none is available with that role, the conceptual integrity is lacking
  - **Correctness and completeness:** are essential for the architecture to allow for all of the system's requirements
  - A formal evaluation is the architect's best hope for a correct and complete architecture
  - **Buildability:** allows the system to be completed by the available team in a timely manner
  - Is open to certain changes as development progress

Achieving qualities

VASIREDDY VENKATADRI INSTITUTE OF TECHNOLOGY

- Our interest is in the tactics, used by the architect to create a design using DP, Architectural Patterns, etc.
- Architect should consider what combination of tactics of modifiability should be applied, as the tactics chosen will guide the architectural decisions
- A tactic is a design decision that influences the control of a quality attribute response
- A collection of tactics is known as architectural strategy

Availability of tactics

- A failure occurs when the system no longer delivers a service, which is observable by users
- A fault has the potential to cause a failure
- Repair is an important aspect of availability
- The tactics we discuss will keep faults from becoming failures
- Many tactics are available within standard execution environments such as OS, application servers, DBMS
- In some cases, monitoring or recovery is automatic and in others it is manual

Goal of availability tactics

we consider:

- **Fault detection**
- **Faulty recovery**
- **Fault prevention**

Fault detection

There are three widely used tactics

- **Ping/echo**: one component issues ping and expects to receive back an echo within a predefined time
- It is used in client server architecture to ensure communication path is in expected performance
- **Hearbeat (dead man timer)**: one component emits a heartbeat message periodically and another listens
- If the heartbeat fails, the originating component assumed to have failed and a fault is notified (e.g. ATM)
- **Exceptions**: this is raised when one of the fault classes is recognized

Faulty recovery

Some preparations and repair tactics are:

- **Voting**: equal i/p is given to redundant processors and compute a simple o/p value that is sent to voter
- If the voter detect deviant behavior from a single processor, it fails it
- **Active redundancy**: all redundant components (all in same state) respond to events in parallel
- Response from only one is used and rest are discarded
- When a fault occurs, downtime of system is milliseconds and the only time to recover is the switchingtime

- **Passive redundancy**: one component responds to events and informs to other components of state updates they must make
- if fault occurs, system must ensure that the backup state is sufficiently fresh before resuming services
- **Spare**: a standby computing platform is configured to replace many different failed components
- **Shadow operation**: a previously failed component may be run in shadow mode for a short time
- **Check-point/ rollback**: a check point is a recording of a consistent state created either periodically or in response to specific events
- If a system fails, the system should be restored using a previous check point

Fault prevention

Following are few fault prevention techniques:

- **Removal from service**: this tactic removes a component from operation to undergo some activities to prevent anticipated failures (e.g. rebooting a system)
- **Transactions**: the bundling of several sequential steps such that the entire bundle can be undone at once
- **Process monitor**: if a fault in a process is detected, a monitoring process can delete the non-performing process and create a new instance of it

Modifiability tactics

VASIREDDY VENKATADRI INSTITUTE OF TECHNOLOGY

- We organize the tactics for modifiability in sets according to their goals
- One set has as its goal reducing the no.of modules that are directly affected by a change
- These sets are:
  - **Localize modifications**
  - **Prevent ripple effects**

- **Localize modifications:** the goal of tactics in this set is to assign responsibilities to modules
- They are: maintain semantic coherence, anticipate expected changes, generalize the module, limit possible options
- **Prevent ripple effects:** a ripple effect is to modify a dependent module since other module was modified
- Various types of dependencies are: syntax & semantics of data & service, sequence of data & control, identity of an interface, location of, quality of service, resource behavior

Performance tactics

- After an event arrives, either system processes that or the processing is blocked for some reason
- This leads to resource consumption or blocked time
- With this background, there are three tactics:
  - **Resource demand**
  - **Resource management**
  - **Resource arbitration**

- **Resource demand:** one tactic for reducing latency is to reduce the resources required for processing
- To do this, increase computational efficiency and reduce computational overhead, manage event rate
- **Resource management:** though the demand for resources may not be controllable, management of the resources affects response times
- Few resource management tactics: introduce concurrency, maintain multiple copies data, increase resources
- **Resource arbitration:** when there is a contention for resource, it must be scheduled
- Few scheduling processes are: FIFO, Fixed Priority, Dynamic priority, Static scheduling

Security tactics

- Security tactics can be divided into: Resisting attacks, Detecting attacks, Recovering from attacks
- **Resisting attacks:** following attacks can be used to in combination to achieve these goals
  - Authenticate users
  - Authorize users
  - Maintain data confidentiality
  - Maintain integrity
  - Limit exposure
  - Limit access

- **Detecting attacks**: attacks are detected usually by an intrusion detection system
- It works by comparing n/w traffic patterns to a DB
- In case of misuse, the traffic pattern is compared to historic patterns of known attacks
- **Recovering from attacks**: it can be done using either with restoring state or attacker identification
- The tactics used in restoring the system or data to a correct state overlap with those used for availability
- One special attentions is paid to maintain redundant copies of system administrative data
- For identifying an attacker is to maintain an audit trail, which is a copy of each transaction applied to the data

Testability tactics

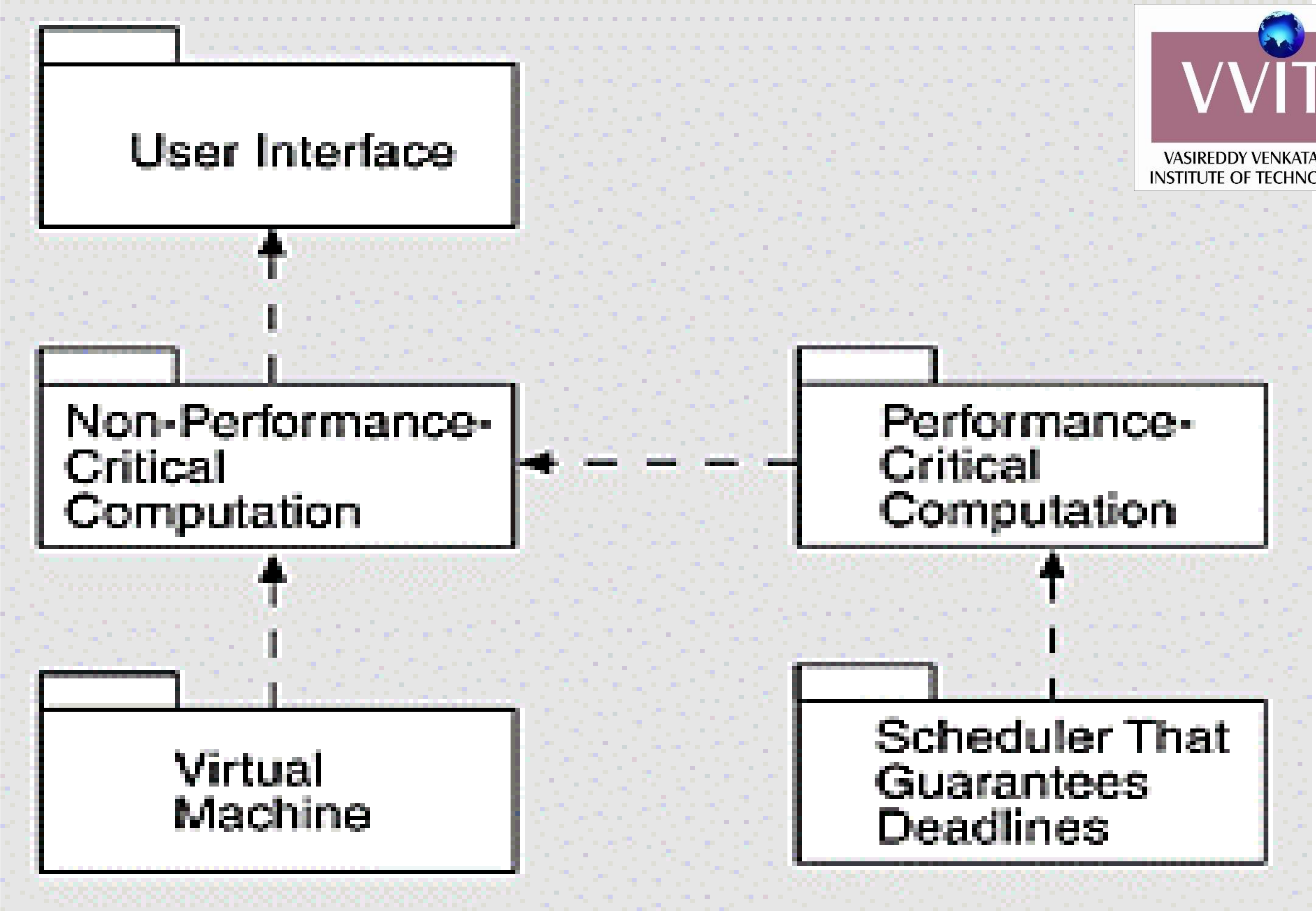
- Goal of this tactics is to allow for easier testing when an increment of s/w development is done
- There are two categories or tactics for testing: providing i/p & capturing o/p, internal monitoring
- **Input/output**: there are three tactics for this testing, viz. record/playback, separate interface from implementation and specialize access routes/interfaces
- **Internal monitoring**: built-in monitors can maintain state, performance, capacity, security or other information

Usability tactics

- Usability is concerned with how easy it is for the user to accomplish a task
- Two types of tactics support usability, viz. runtime tactics, design time tactics
- **Runtime tactics**: providing the user with the ability to issue usability-based commands like, cancel, undo, show multiple views, etc.
- **Design-time tactics**: user interfaces are typically revised frequently during the testing process
- Separate the user interface from the rest of the app.

Designing the Architecture - ADD

- A method called Attribute Driven Design (ADD) is used to satisfy quality and functional requirements
- ADD defines a SA that bases the decomposition process on the quality attributes the s/w has to fulfill
- o/p of ADD is the first several levels of a module decomposition view of an architecture & other views
- ADD depends on identification of the drivers and can start as soon as all of them are known
- Example taken is, ‘a garage door opener’ within a home information systems
- The opener is responsible for raising and lowering the door via switch/ remote/ home info-sys



Architectural pattern that utilizes tactics to achieve garage door drivers

ADD steps: these are the steps performed when designing an architecture using ADD method

1. **Choose the module to decompose:**
2. **Refine the module according to these steps:**
   a. **Choose the architectural drivers**
   b. **Choose an architectural pattern**
   c. **Instantiate modules and allocate functionality from the usecases and represent using multiple views**
   d. **Define interfaces of the child modules**
   e. **Verify and refine usecases and quality scenarios**
3. **Repeat the steps above for every module that needs further decomposition**

1. Choose the module to decompose:

- All modules are: system, subsystem, and sub-module
- The decomposition starts with system, which is then decomposed into subsystems, which are further decomposed into sub modules

2a. Choose the Architectural drivers:

- Architectural drivers are the combination of functional and quality requirements that shape the architecture or the particular module
- The drivers will be found among the top-priority requirements for the module

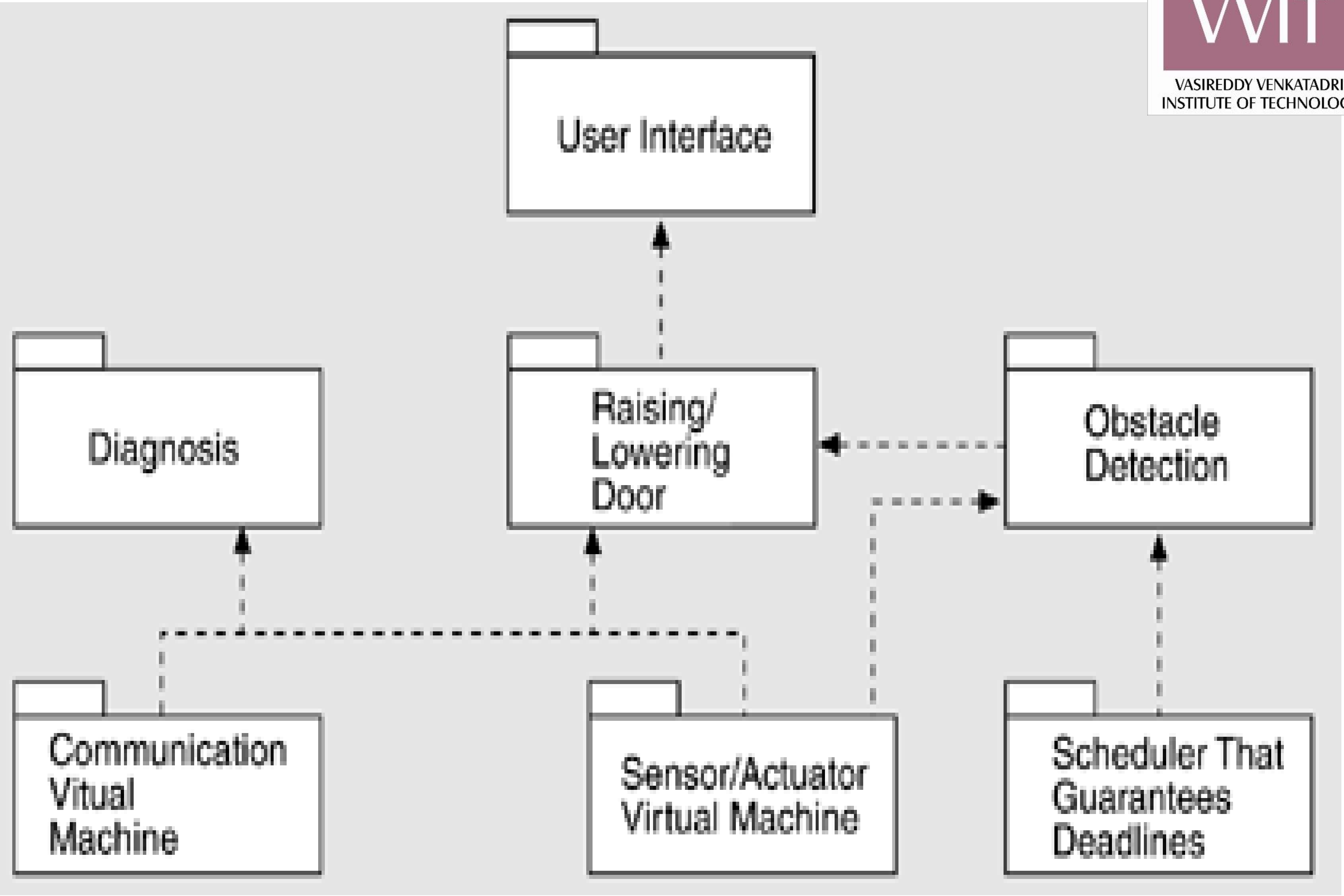
2b. Choose an Architectural Pattern

- For each quality there are identifiable tactics
- Each tactic is designed to realize one or more quality attributes and many such tactics can be used
- Two main factors guide tactic selection: drivers, and side effects that a pattern implementing a tactic has on other qualities

2c. Instantiate modules and allocate functionality

- **Instantiate modules**: E.g. we allocate responsibility for managing obstacle detection & halting the garage door to performance critical system since this functionality has a deadline
- The management of normal raising and lowering of door has no timing deadline, hence it is treated as non-performance critical section

- Diagnosis capabilities are also non-performance critical
- Thus the non-performance-critical module of previous figure becomes instantiated as diagnosis & raising/ lowering door modules in next fig
- **Allocate functionality**: Applying usecases that belongs to the parent module helps the architect gain a more detailed understanding of the distribution of functionality
- This also may lead to adding/ removing child modules to fulfill all the functionality required



First-level decomposition of garage door opener

- **Represent the architecture with views**: ADD uses three common views:
  - ***Module decomposition view***: this provides containers for holding responsibilities
  - Major data flow relations among the modules are also identified
  - ***Concurrency view***: dynamic aspects of a system such as parallel activities and synchronization can be modeled
  - This model identifies resource contention problems, deadlocks, data consistency issues, etc.
  - ***Deployment view***: if multiple processors or specialized h/w is used in a system, additional responsibilities may arise from deployment to the h/w
  - This view result in the virtual threads of the concurrency view being decomposed into virtual threads
  - Messages that travel between processors to initiate the next entry in sequence of actions

2d. Define interfaces of the child module:

- An interface of a module shows the services and properties provided & re
- Analyzing and documenting the decomposition in terms of structure (module decomposition view), dynamism (concurrency view), and runtime (deployment view)

2e. Verify and refine usecases and quality scenarios as constraints for the child modules:

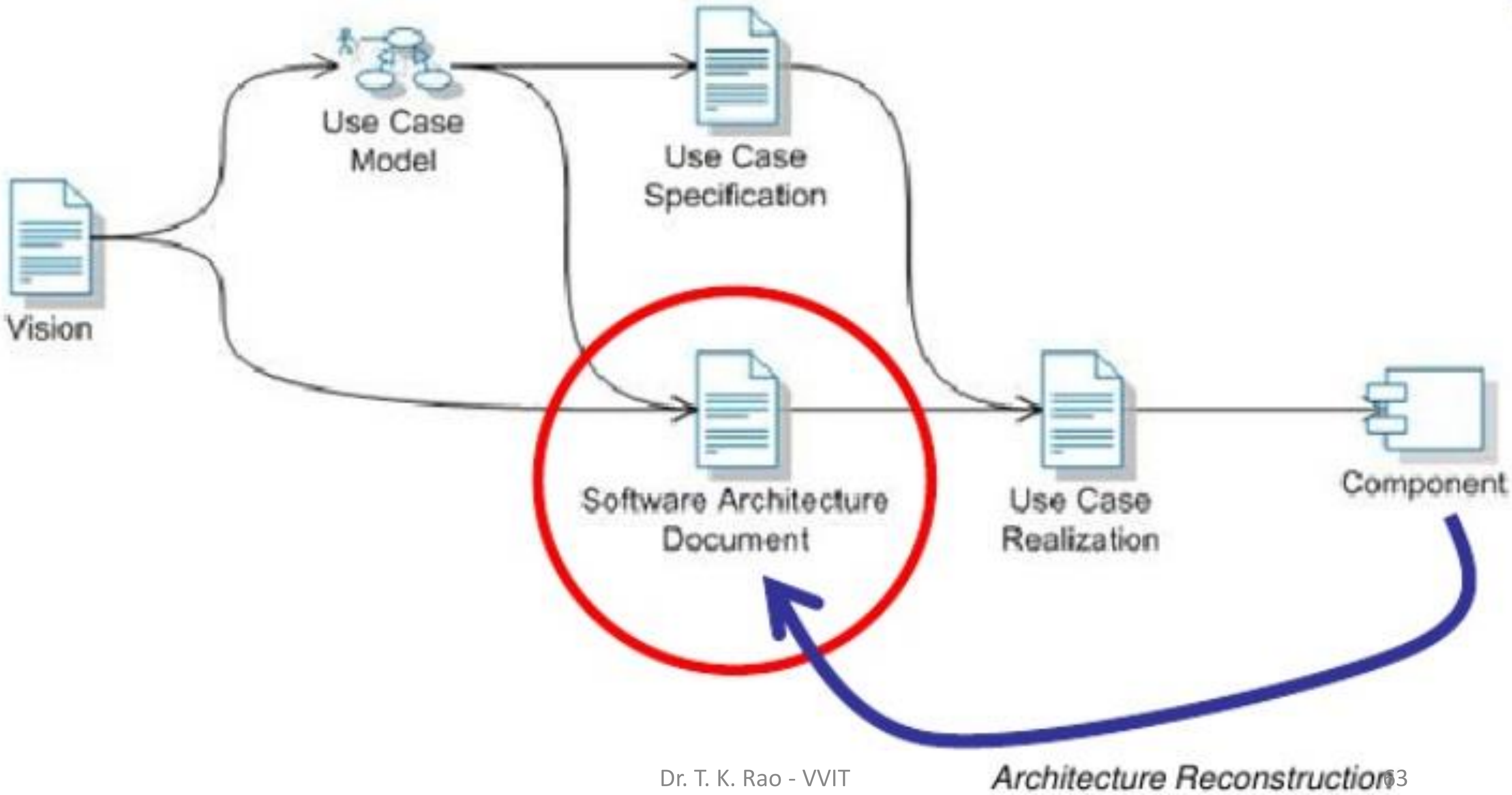
- **Functional requirements**: each child module has responsibilities that derive partially from considering decomposition of the functional requirements
- **Constraints**: constraints of the parent module can be satisfied in one of the ways, viz. the decomposition satisfies the constraint, the constrain is satisfied by a single child module, constraint is satisfied by multiple child modules
- **Quality scenarios**: these are also have to be refined and assigned to the child modules

Documenting SAs

Following points are important for discussion

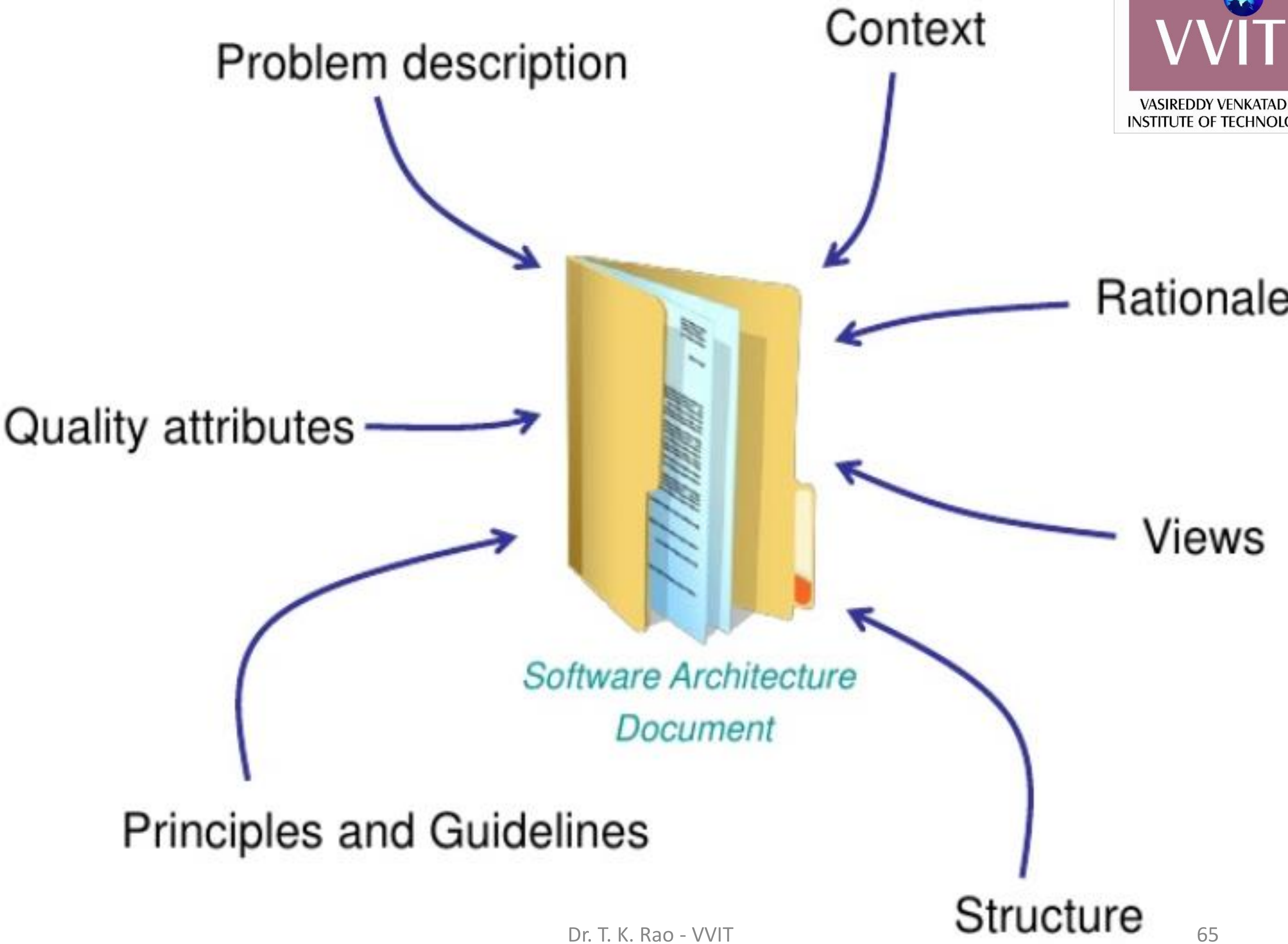
- **Uses of architectural documentation**
- **Views**
- **Choosing the relevant views**
- **Documenting a view**
- **Documentation across views**

Architecture in the software development process



Uses of architectural documentation

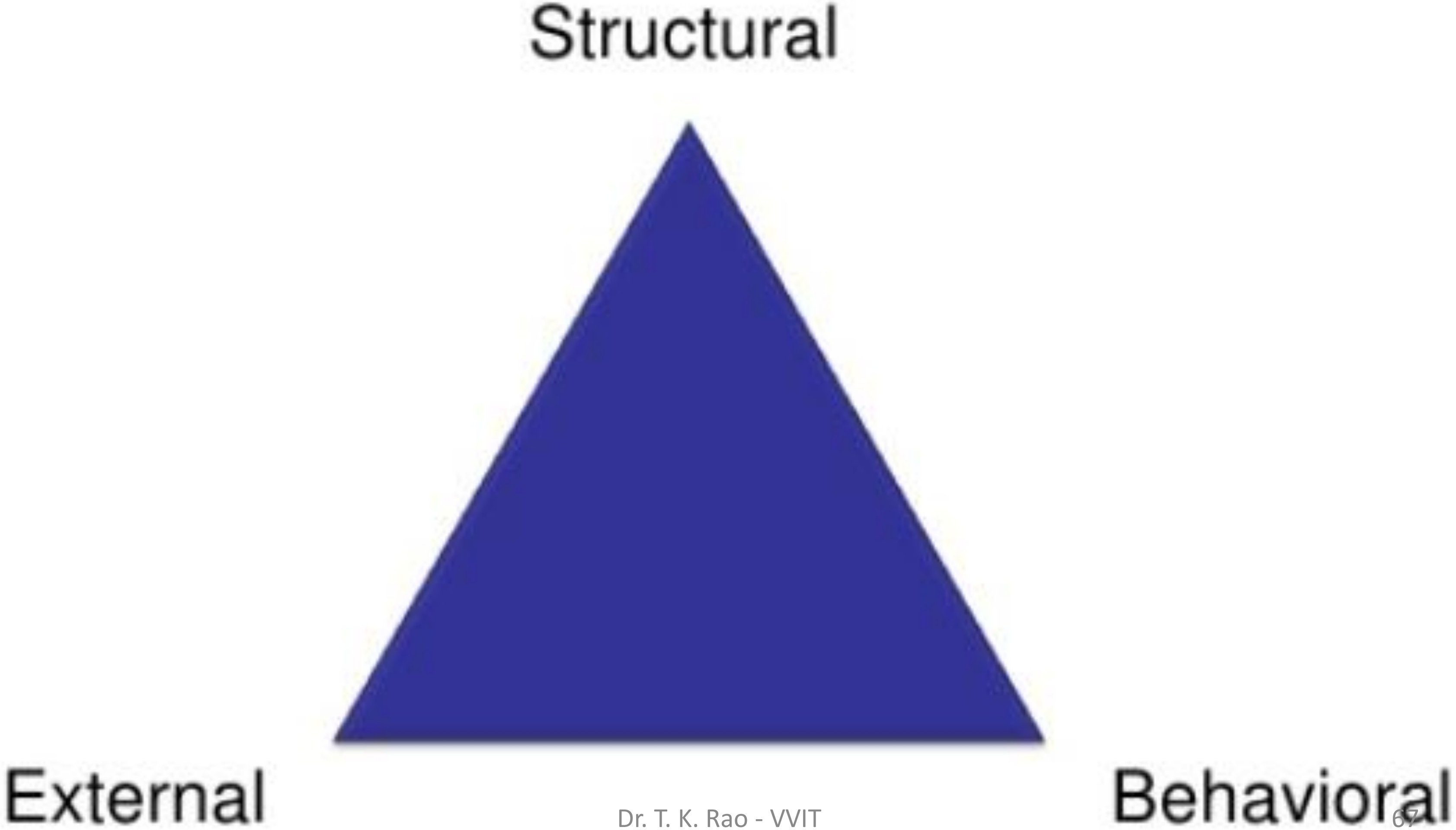
- Different stakeholders have different needs, i.e. different kinds of info., levels of info., and treatments of info.
- One fundamental rule for technical documentation is to write from readers' point of view
- Understanding who the stakeholders are & how they will want to use the documentation helps in organizing it and make it accessible to and usable for them
- Primary purpose of architecture was to serve as communication vehicle among stakeholders
- Documentation facilitates that communication
- Arch. Doc. is a key means for educating people who need an overview: new developers, sponsors, project visitors, etc.


Problem description
Context
Rationale
Quality attributes
Views
Software Architecture
Document
Principles and Guidelines
Structure

Choosing the relevant views

- Quality attributes of most concern to stakeholders will affect the choice of what views to document
- E.g. layered view tells system's portability, deployment view describes system's performance
- Views are divided into three: module, component and connector, and allocation
- With the three ways, architect needs to think at least in three ways as how the system is :
  - Structured as a set of implementation units
  - Structured as a set of elements that have runtime behavior and interactions
  - Related to non-s/w structures in its environment

Different kinds of views



SEI example Stakeholders' documentation needs

Stakeholder	Module				C&C	Allocation	
	Decomposition	**Uses**	**Class**	**Layer**	***Various***	**Deployment**	**Implem.**
Project Manager	s	s		s		d	
Member of Development Team	d	d	d	d	d	s	s
Testers and Integrators		d	d		s	s	s
Maintainers	d	d	d	d	d	s	s
Product Line Application Builder		d	s	o	s	s	s
Customer					s	o	
End User					s	s	
Analyst	d	d	s	d	s	d	
Infrastructure Support	s	s		s		s	d
New Stakeholder	x	x	x	x	x	x	x
Current and Future Architect	d	d	d	d	d	d	s

d = detailed information, s = some details, o = overview information, x = anything

A three-step procedure for selecting the view for project:

- Produce a candidate view list: build a stakeholder/view table for project
- Rows and columns are to be filled how much info a stakeholder requires from the view
- E.g. none, overview only, moderate detail, high detail
- Combine views: if the stakeholders could be equally well served by another view having strong constituency
- Look for views that are good candidates to be combined
- Prioritize: you should have an appropriate step of views to serve the stakeholders
- At this point you need to decide what to do first

Documenting a view

- There is no industry-standard template for documenting a view
- The seven-part standard organization has worked well in practice, they are:
  - Primary presentation
  - Element catalog details
  - Context diagram
  - Variability guide
  - Architecture background
  - Glossary of terms
  - Other information

- **Primary presentation:** shows elements & relations among them, it might not include all of them
- It is mostly graphical, tabular also
- **Element catalog:** details of elements & relations that are depicted in primary presentation, & others
- E.g. module decomposition view has elements that are modules, relations that are a form of ‘part of’ and properties that define responsibilities of each module
- **Context diagram:** shows how system depicted in view relates to its environment in vocabulary of the view
- E.g. component-&-connector view shows which component & connectors interact with external components & connectors via which interfaces & protocols

- **Variability guide:** should include about each point of variation in architecture as follows:
  - Options among which a choice is to be made
  - The binding time of the option, some choices are made at design time and some at build time and others at runtime
- **Architecture background:** explains why the design reflected in the view came to be
- 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

- **Glossary of terms:** terminology used in the views, with a brief description of each
- **Other information:** the precise contents of this section will vary according to the standard practices of your organization
- They include managements information such as authorship, configuration, control data, histories, etc.
- Architect might record references to specific sections of a requirements document to establish traceability

Documentation across views

- Cross view documentation consists of three major aspects, can be summarized as how-what-why
  - How the documentation is laid out and organized so that a stakeholder of the architecture can find information efficiently and reliably (consists of view catalog & template)
  - What the architecture is, here information that remains to be captured beyond the views themselves is a short system overview to ground any reader as to the purpose of system
  - 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