USN

CMRIT
CMR INSTITUTE OF TECHNOLOGY, BENGALURU.
ACCREDITED WITH A+ GRADE BY NAAC

Internal Assessment Test III– May 2018

Sub:	Software Architectures				Sub Code:	**10IS81**	Branch:	CSE	
Date:	21/05/18	Duration:	90 min’s	Max Marks:	50	Sem / Sec:	VIII/A,B,C		OBE

	Answer any FIVE FULL Questions	MARKS	CO	RBT
1	Enumerate the implementation of microkernel architectural pattern.	[10]	CO3	L3
2	List out and explain the components of Micro kernel architectural pattern.	[10]	CO3	L3
3	Discuss in detail the structure of whole part design pattern and its implementation.	[10]	CO4	L2
4	Explain in detail the components and six phases of dynamic scenarios of master slave design pattern.	[10]	CO4	L3
5	Define proxy design pattern. Discuss the implementation steps to carry out the proxy pattern.	[10]	CO4	L2

USN

CMRIT
CMR INSTITUTE OF TECHNOLOGY, BENGALURU.
ACCREDITED WITH A+ GRADE BY NAAC

Internal Assessment Test II – April 2018

Sub:	Software Architectures				Sub Code:	**10IS81**	Branch:	CSE	
Date:	21/05/18	Duration:	90 min’s	Max Marks:	50	Sem / Sec:	VIII/A,B,C		OBE

	Answer any FIVE FULL Questions	MARKS	CO	RBT
1	Enumerate the implementation of microkernel architectural pattern.	[10]	CO3	L3
2	List out and explain the components of Micro kernel architectural pattern.	[10]	CO3	L3
3	Discuss in detail the structure of whole part design pattern and its implementation.	[10]	CO4	L2
4	Explain in detail the components and six phases of dynamic scenarios of master slave design pattern.	[10]	CO4	L3
5	Define proxy design pattern. Discuss the implementation steps to carry out the proxy pattern.	[10]	CO4	L2

6	Explain the steps involved in designing an architecture using the ADD method.	[10]	CO5	L3
7	List and explain the steps in choosing and documenting a view.	[10]	CO5	L3

6	Explain the steps involved in designing an architecture using the ADD method.	[10]	CO5	L3
7	List and explain the steps in choosing and documenting a view.	[10]	CO5	L3

Scheme and solution

Question #	**Description**	**Marks Distribution**		**Max Marks**
1	Implementation of microkernel architectural pattern.	10M	10M	10M
2	List & explanation of components of Micro kernel architectural pattern.	10M	10M	10M
3	The structure of whole part design pattern and its implementation	5M 5M	10M	10M
4	The components Six phases of dynamic scenarios of master slave design pattern.	5M 5M	10M	10M
5	Define proxy design pattern. The implementation steps to carry out the proxy pattern	5M 5M	10M	10M
6	The steps involved in designing an architecture using the ADD method	10M	10M	10M
7	List and explanation of the steps in choosing and documenting a view.	4 M 6 M	10M	10M

1>

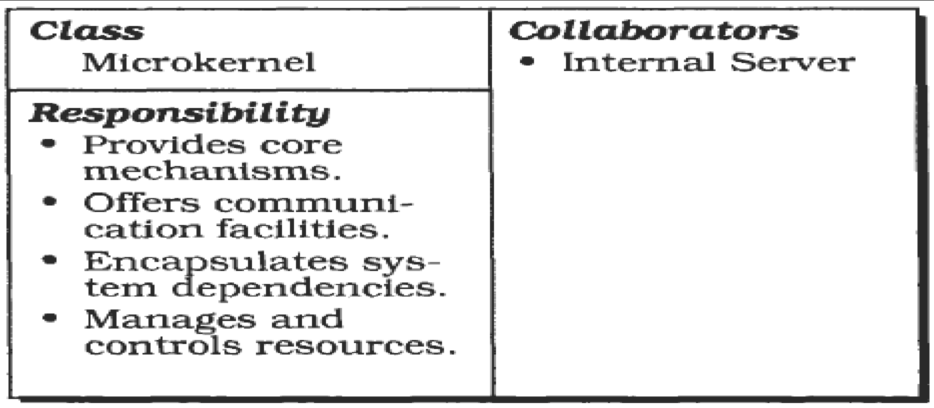
Implementation:

1. Analyze the application domain:
Perform a domain analysis and identify the core functionality necessary for implementing ext servers. **2. Analyze external servers**
That is polices ext servers are going to provide **3. Categorize the services**
Group all functionality into semantically independent categories. **4. Partition the categories**
Separate the categories into services that should be part of the microkernel, and those that should be available as internal servers.
5. **Find a consistent and complete set of operations and abstractions** for every category you identified in step 1.
6. Determine the strategies for request transmission and retrieval.
Specify the facilities the microkernel should provide for communication b/w components.
Communication strategies you integrate depend on the requirements of the application domain. **7. Structure the microkernel component**
Design the microkernel using the layers pattern to separate system-specific parts from system-independent parts of the kernel.
8. Specify the programming interfaces of microkernel
To do so, you need to decide how these interfaces should be accessible externally.
You must take into an account whether the microkernel is implemented as a separate process or as a module that is physically shared by other component in the latter case, you can use conventional method calls to invoke the methods of the microkernel.
9. The microkernel is responsible for **managing all system resources** such as memory blocks, devices or **device contexts -** a handle to an output area in a GUI implementation.
10. Design and implement the internal servers as separate processes or shared libraries
Perform this in parallel with steps 7-9, because some of the microkernel services need to access internal servers.
It is helpful to distinguish b/w active and passive servers ✓ Active servers are implemented as processes ✓ Passive servers as shared libraries.
Passive servers are always invoked by directly calling their interface methods, where as active server process waits in an event loop for incoming requests.
11 Implement the external servers
Each external server is implemented as a separate process that provide its own service interface The internal architecture of an external server depends on the policies it comprises
Specify how external servers dispatch requests to their internal procedures. **12. Implement the adapters**
Primary task of an adapter is to provide operations to its clients that are forwarded to an external server.
You can design the adapter either as a conventional library that is statically linked to client during complication or as a shared library dynamically linked to the client on demand.

2>

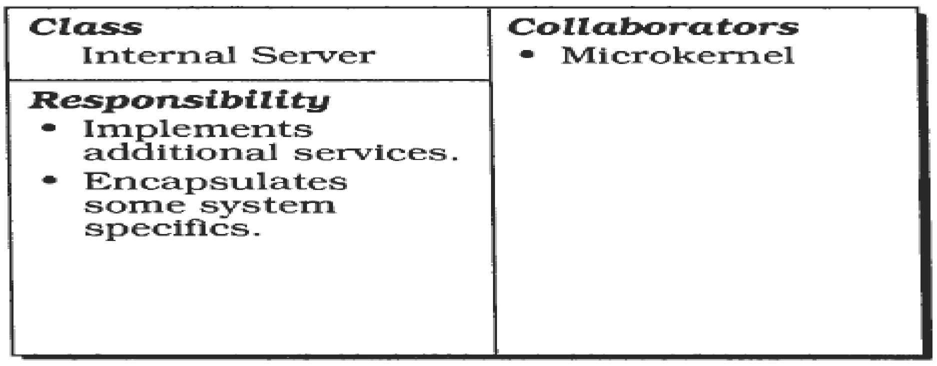
Microkernel pattern defines 5 kinds of participating components. Internal servers
External servers
Adapters
Clients
Microkernel

o The microkernel represents the main component of the pattern.
o It implements central services such as communication facilities or resource handling.
o The microkernel is also responsible for maintaining system resources such as processes or files. o It controls and coordinates the access to these resources.
o A microkernel implements atomic services, which we refer to as mechanisms.
o These mechanisms serve as a fundamental base on which more complex functionality called policies are constructed.



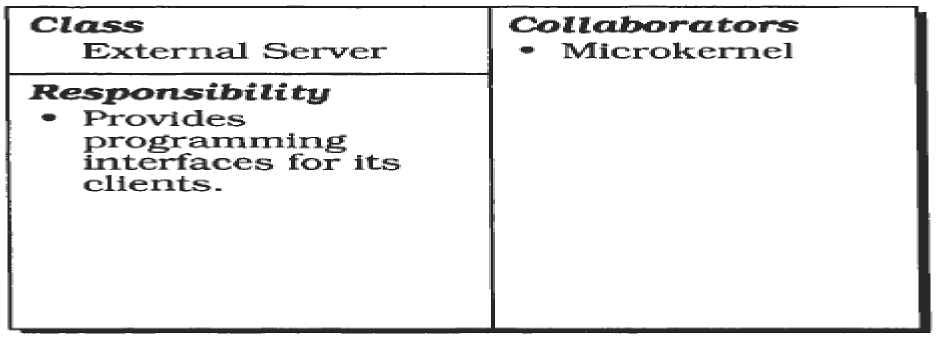
Class Microkernel	***Collaborators***
Responsibility • Provides core mechanisms. • Offers communication facilities. • Encapsulates system dependencies. • Manages and controls resources.	• Internal Server

❖An internal server (subsystem)
o Extends the functionality provided by microkernel.
o It represents a separate component that offers additional functionality.
o Microkernel invokes the functionality of internal services via service requests.
o Therefore internal servers can encapsulates some dependencies on the underlying hardware or software system.



Class Internal Server	***Collaborators***
Responsibility • Implements additional services. • Encapsulates some system specifics.	• Microkernel

An external server (personality)
o Uses the microkernel for implementing its own view of the underlying application domain. o Each external server runs in separate process.
o It receives service requests from client applications using the communication facilities provided by the microkernel, interprets these requests, executes the appropriate services, and returns its results to clients.
o Different external servers implement different policies for specific application domains.



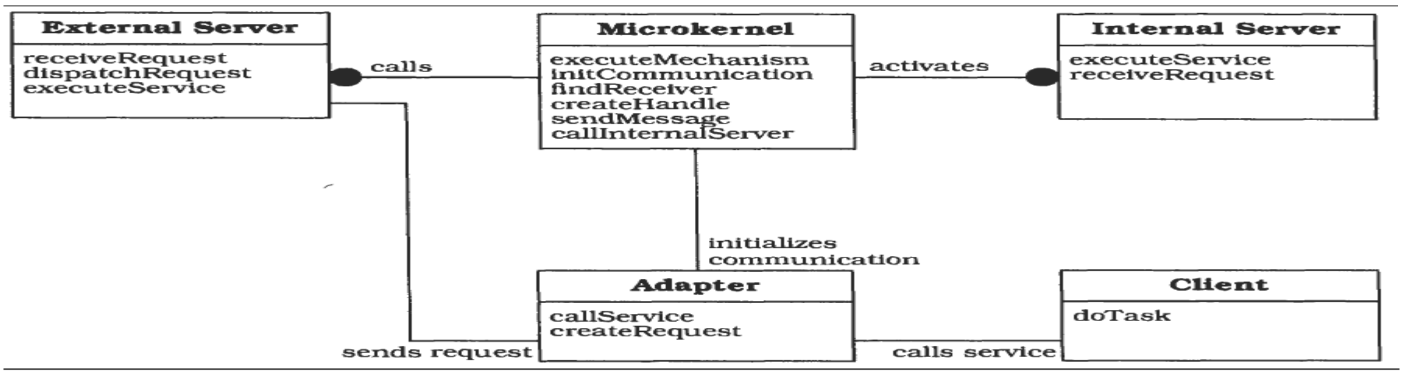
Class External Server	Collaborators
Responsibility • Provides programming interfaces for its clients.	• Microkernel

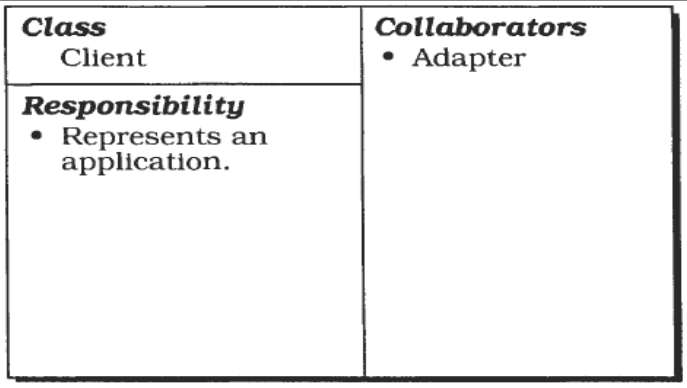
❖ **Client:**

o It is an application that is associated with exactly one external server. It only accesses the programming interfaces provided by the external server.
o Problem arises if a client accesses the interfaces of its external server directly (direct dependency) ✓ Such a system does not support changeability
✓ If ext servers emulate existing application platforms clients will not run without modifications.

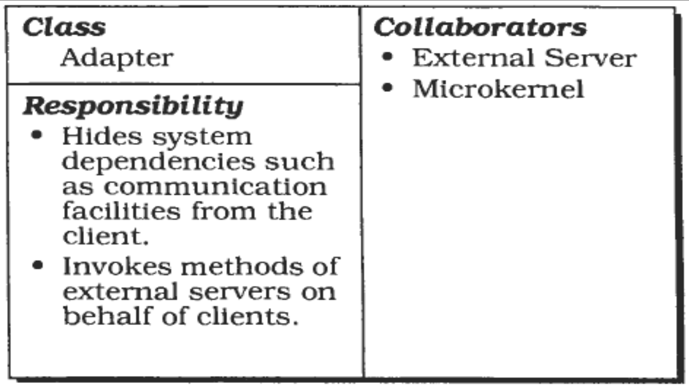
❖**Adapter** (emulator)

o Represents the interfaces b/w clients and their external servers and allow clients to access the services of their external server in a portable way.
o They are part of the clients address space.
o The following OMT diagram shows the static structure of a microkernel system.





Class Client	Collaborators
Responsibility • Represents an application.	• Adapter



Class Adapter	Collaborators
Responsibility • Hides system dependencies such as communication facilities from the client. • Invokes methods of external servers on behalf of clients.	• External Server • Microkernel

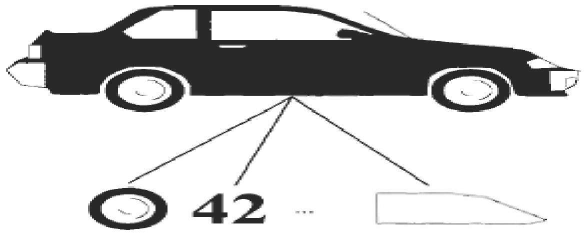
3>

WHOLE-PART

Whole-part design pattern helps with the aggregation of components that together form a semantic unit. An aggregate component, the whole, encapsulates its constituent components, the parts, organizes their collaboration, and provides a common interface to its functionality. Direct access to the parts is not possible.

Example:

A computer-aided design (CAD) system for 2-D and 3-D modelling allows engineers to design graphical objects interactively. For example, a car object aggregates several smaller objects such as wheels and windows, which



The Whole-Part pattern introduces two types of participant: ❖**Whole**

Whole object represents an aggregation of smaller objects, which we call parts.

It forms a semantic grouping of its parts in that it co ordinates and organizes their collaboration.

Some methods of whole may be just place holder for specific part services when such a method is invoked the whole only calls the relevant part services, and returns the result to the client.

❖part

Each part object is embedded in exactly one whole. Two or more parts cannot share the same part. Each part is created and destroyed within the life span of the whole.

Implementation:

1. Design the public interface of the whole

➢ Analyze the functionality the whole must offer to its clients. ➢ Only consider the clients view point in this step.

➢ Think of the whole as an atomic component that is not structured into parts. **2. Separate the whole into parts, or synthesize it from existing ones.**

➢ There are two approaches to assembling the parts either assemble a whole 'bottom-up' from existing parts, or decompose it 'top-down' into smaller parts.

➢ Mixtures of both approaches is often applied

3. **If you follow a bottom up approach**, use existing parts from component libraries or class libraries and specify their collaboration.

4. **If you follow a top down approach, partition the Wholes services into smaller collaborating services** and map these collaborating services to separate parts.

5. Specify the services of the whole in terms of services of the parts.

Decide whether all part services are called only by their whole, or if parts may also call each other. Two are two possible ways to call a Part service:

@ If a client request is forwarded to a Part service, the Part does not use any knowledge about the execution context of the Whole, relying on its own environment instead.

@ A delegation approach requires the Whole to pass its own context information to the Part. **6. Implement the parts**

If parts are whole-part structures themselves, design them recursively starting with step1 . if not reuse existing parts from a library.

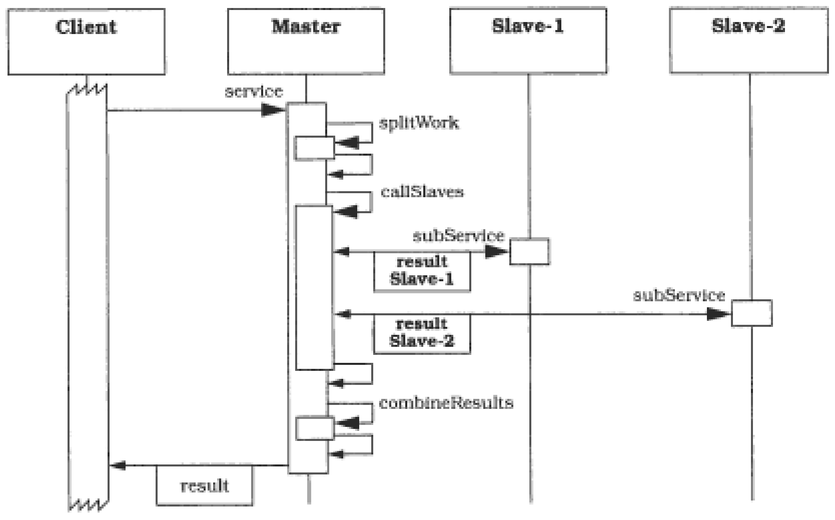
7. Implement the whole

Implement services that depend on part objects by invoking their services from the whole.

4>

Dynamics:

The scenario comprises six phases:
A client requests a service from the master.
The master partitions the task into several equal sub-tasks.
The master delegates the execution of these sub-tasks to several slave instances, starts their execution and waits for the results they return.
The slaves perform the processing of the sub-tasks and return the results of their computation back to the master.
The master computes a final result for the whole task from the partial results received from the slaves.
The master returns this result to the client.

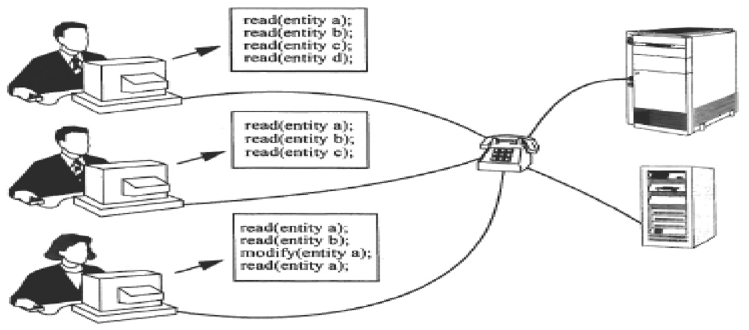


5>

Proxy design pattern makes the clients of a component communicate with a representative rather than to the component itself. Introducing such a place holder can serve many purposes, including enhanced efficiency, easier access and protection from unauthorized access.

Example:

Company engineering staff regularly consults databases for information about material providers, available parts, blueprints, and so on. Every remote access may be costly, while many accesses are similar or identical and are repeated often. This situation clearly offers scope for optimization of access time and cost.



```
read(entity a);
read(entity b);
read(entity c);
read(entity d);
```

```
read(entity a);
read(entity b);
read(entity c);
```

```
read(entity a);
read(entity b);
modify(entity a);
read(entity a);
```

Implementation:

1. Identify all responsibilities for dealing with access control to a component Attach these responsibilities to a separate component the proxy.

2. If possible introduce an abstract base class that specifies the common parts of the interfaces of both the proxy and the original.
Derive the proxy and the original from this abstract base. **3. Implement the proxy's functions**
To this end check the roles specified in the first step
4. **Free the original and its client** from responsibilities that have migrated into the proxy.
5. **Associate the proxy and the original** by giving the proxy a handle to the original. This handle may be a pointer a reference an address an identifier, a socket, a port, and so on.
6. Remove all direct relationships between the original and its client Replace them by analogous relationships to the proxy.

6>
ADD Steps:
Steps involved in attribute driven design (ADD) *1. Choose the module to decompose*
○ Start with entire system
○ Inputs for this module need to be available
○ Constraints, functional and quality requirements *2. Refine the module*
a) Choose architectural drivers relevant to this decomposition b) Choose architectural pattern that satisfies these drivers
c) Instantiate modules and allocate functionality from use cases representing using multiple views d) Define interfaces of child modules
e) Verify and refine use cases and quality scenarios *3. Repeat for every module that needs further decomposition*

Discussion of the above steps in more detail:
1. Choose The Module To Decompose
○ the following are the modules: system->subsystem->submodule
○ Decomposition typically starts with system, which then decompose into subsystem and then into sub-modules.
○ In our Example, the garage door opener is a system
○ Opener must interoperate with the home information system

2. **Refine the module**
1. Choose Architectural Drivers:
○ choose the architectural drivers from the quality scenarios and functional requirements ○ The drivers will be among the top priority requirements for the module.
○ In the garage system, the 4 scenarios were architectural drivers, ○ By examining them, we see
▪ Real-time performance requirement
▪ Modifiability requirement to support product line ○ Requirements are not treated as equals
○ Less important requirements are satisfied within constraints obtained by satisfying more important requirements
○ This is a difference of ADD from other architecture design methods
2. Choose Architectural Pattern
○ For each quality requirement there are identifiable tactics and then identifiable patterns that implement these tactics.
○ The goal of this step is to establish an overall architectural pattern for the module
○ The pattern needs to satisfy the architectural pattern for the module tactics selected to satisfy the drivers
○ Two factors involved in selecting tactics: ✓ Architectural drivers themselves
✓ Side effects of the pattern implementing the tactic on other requirements ○ This yields the following tactics:
Semantic coherence and *information hiding.* Separate responsibilities dealing with the user interface, communication, and sensors into their own modules.
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.

3. Instantiate Modules And Allocate Functionality Using Multiple Views *Instantiate modules*
The non-performance-critical module of Figure 7.2 becomes instantiated as diagnosis and raising/lowering door modules in Figure 7.3. We also identify several responsibilities of the virtual machine: communication and sensor reading and actuator control. This yields two instances of the virtual machine that are also shown in Figure 7.3.
Allocate functionality
Assigning responsibilities to the children in a decomposition also leads to the discovery of necessary information exchange. At this point in the design, it is not important to define how the information is exchanged. Is the information pushed or pulled? Is it passed as a message or a call parameter? These are all questions that need to be answered later in the design process. At this point only the information itself and the producer and consumer roles are of interest

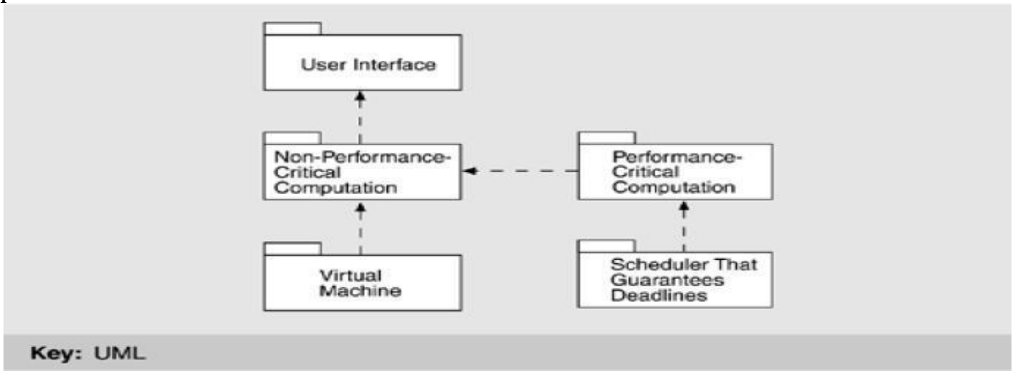

User Interface
Non-Performance-Critical Computation
Performance-Critical Computation
Virtual Machine
Scheduler That Guarantees Deadlines
Key: UML

Figure 7.3.

- *Represent the architecture with multiple views*

Module decomposition view
Concurrency view
Two users doing similar things at the same time
One user performing multiple activities simultaneously Starting up the system
Shutting down the system *Deployment view*

4. Define Interfaces Of Child Modules

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

5. Verify And Refine Use Cases And Quality Scenarios As Constraints For The Child Modules

o *Functional requirements*
Using functional requirements to verify and refine
➢ Decomposing functional requirements assigns responsibilities to child modules ➢ We can use these responsibilities to generate use cases for the child module
✓ User interface:
▪ Handle user requests
Translate for raising/lowering module Display responses
✓ *Raising/lowering door module*
Control actuators to raise/lower door
Stop when completed opening or closing ✓ *Obstacle detection:*
Recognize when object is detected
Stop or reverse the closing of the door ✓ *Communication virtual machine*
Manage communication with house information system(HIS)
✓ *Scheduler*
▪ Guarantee that deadlines are met when obstacle is detected ✓ *Sensor/actuator virtual machine*
▪ Manage interactions with sensors/actuators ✓ *Diagnosis:*
▪ Manage diagnosis interaction with HIS

- *Constraints:*

✓ The decomposition satisfies the constraint

- OS constraint-> satisfied if child module is OS The constraint is satisfied by a single module
- Constraint is inherited by the child module
- The constraint is satisfied by a collection of child modules
- E.g., using client and server modules to satisfy a communication constraint

- *Quality scenarios:*

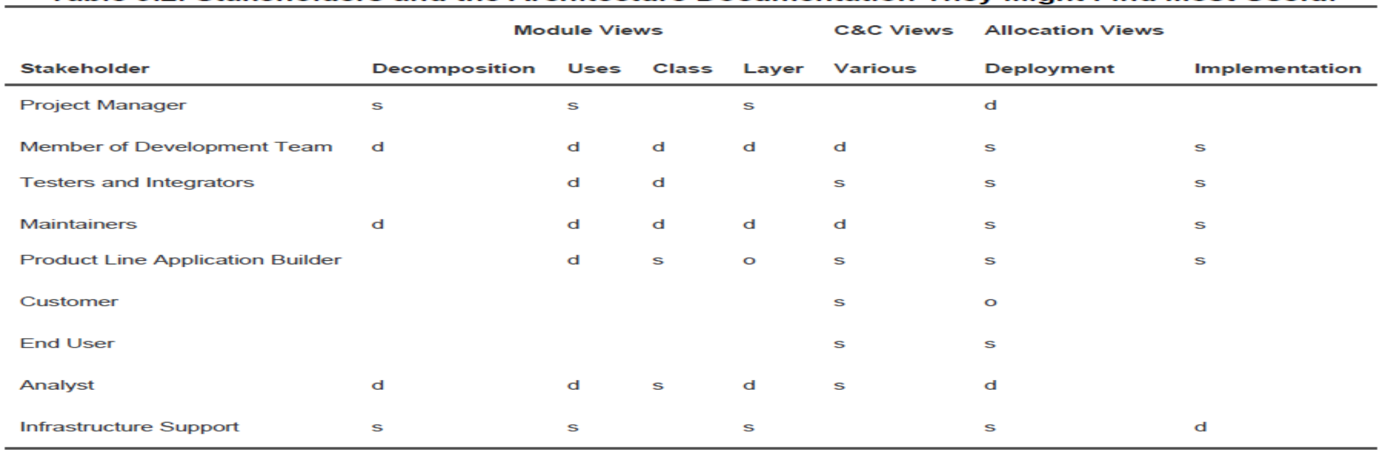
- Quality scenarios also need to be verified and assigned to child modules
- A quality scenario may be satisfied by the decomposition itself, i.e, no additional impact on child modules
- A quality scenario may be satisfied by the decomposition but generating constraints for the children
- The decomposition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

7>
A view simply represents a set of system elements and relationships among them, so whatever elements and relationships you deem useful to a segment of the stakeholder community constitute a valid view.
Here is a simple 3 step procedure for choosing the views for your project.

1. Produce a candidate view list:

Begin by building a stakeholder/view table. Your stakeholder list is likely to be different from the one in the table as shown below, but be as comprehensive as you can. For the columns, enumerate the views that apply to your system. Some views apply to every system, while others only apply to systems designed that way. Once you have rows and columns defined, fill in each cell to describe how much information the stakeholder requires from the view: none, overview only, moderate detail, or high detail.

Table 9.2. Stakeholders and the Architecture Documentation They Might Find Most Useful



	Module Views				C&C Views	Allocation Views	
Stakeholder	Decomposition	Uses	Class	Layer	Various	Deployment	Implementation
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

2. Combine views:

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

3. Prioritize:

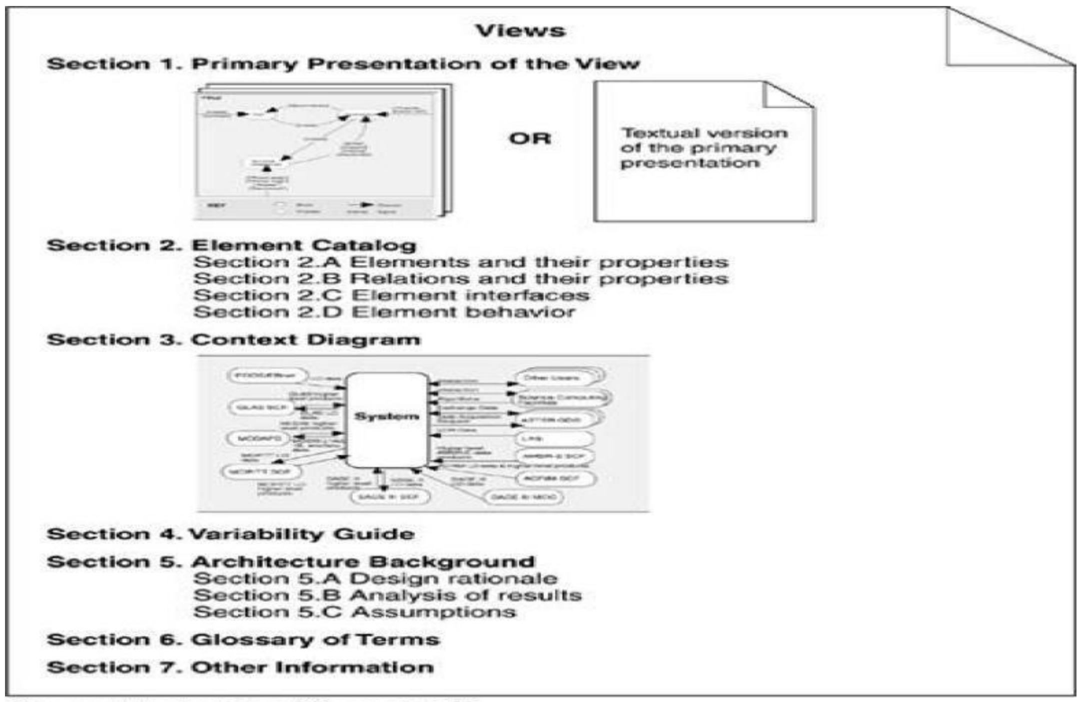
After step 2 you should have an appropriate set of views to serve your stakeholder community. At this point you need to decide what to do first. How you decide depends on the details specific project. But, remember that you don't have to complete one view before starting another. People can make progress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

DOCUMENTING A VIEW

Primary presentation- elements and their relationships, contains main information about these system , usually graphical or tabular.
Element catalog- details of those elements and relations in the picture, **Context diagram-** how the system relates to its environment
Variability guide- how to exercise any variation points a variability guide should include documentation about each point of variation in the architecture, including
o The options among which a choice is to be made
o The binding time of the option. Some choices are made at design time, some at build time, and others at runtime.
Architecture background –why the design reflected in the view came to be? an architecture background includes
o rationale, explaining why the decisions reflected in the view were made and why alternatives were rejected
o analysis results, which justify the design or explain what would have to change in the face of a modification
o assumptions reflected in the design
Glossary of terms used in the views, with a brief description of each.
Other information includes 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

DOCUMENTING BEHAVIOR

★Views present structural information about the system. However, structural information is not sufficient to allow reasoning about some system properties .behavior description add information that reveals the ordering of interactions among the elements, opportunities for concurrency, and time dependencies of interactions.
★Behavior can be documented either about an ensemble of elements working in concert. Exactly what to model will depend on the type of system being designed.
★Different modeling techniques and notations are used depending on the type of analysis to be performed. In UML, sequence diagrams and state charts are examples of behavioral descriptions. These notations are widely used.



Views

Section 1. Primary Presentation of the View

OR

Textual version of the primary presentation

Section 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

Section 3. Context Diagram

Section 4. Variability Guide

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

Section 6. Glossary of Terms

Section 7. Other Information

Source: Adapted from [Clements 03].

DOCUMENTING INTERFACES

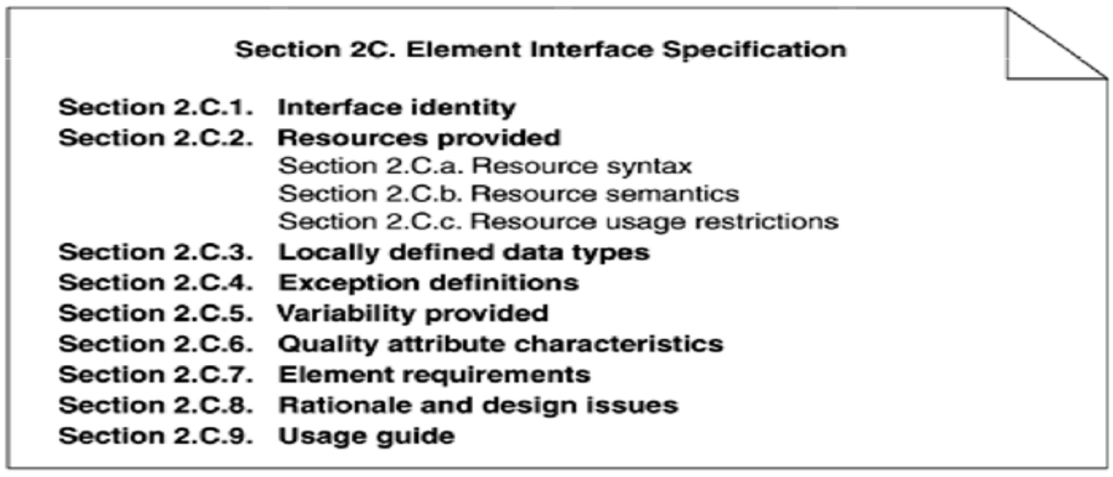
An interface is a boundary across which two independent entities meet and interact or communicate with each other.

1. **Interface identify**

When an element has multiple interfaces, identify the individual interfaces to distinguish them. This usually means naming them. You may also need to provide a version number.
2. Resources provided:

The heart of an interface document is the resources that the element provides. ★*Resource syntax* – this is the resource's signature
★*Resource Semantics:*
Assignment of values of data Changes in state
Events signaled or message sent
how other resources will behave differently in future humanly observable results
★*Resource Usage Restrictions*
initialization requirements
limit on number of actors using resource **3. Data type definitions:**
If used if any interface resources employ a data type other than one provided by the underlying programming language, the architect needs to communicate the definition of that type. If it is defined by another element, then reference to the definition in that element's documentation is sufficient.
4. Exception definitions:
These describe exceptions that can be raised by the resources on the interface. Since the same exception might be raised by more than one resource, if it is convenient to simply list each resource's exceptions but define them in a dictionary collected separately.
5. Variability provided by the interface.
Does the interface allow the element to be configured in some way? These configuration parameters and how they affect the semantics of the interface must be documented.
6. Quality attribute characteristics:
The architect needs to document what quality attribute characteristics (such as performance or reliability) the interface makes known to the element's users
7. Element requirements:
What the element requires may be specific, named resources provided by other elements. The documentation obligation is the same as for resources provided: syntax, semantics, and any usage restrictions.
8. Rationale and design issues:
Why these choices the architect should record the reasons for an elements interface design. The rationale should explain the motivation behind the design, constraints and compromises, what alternatives designs were considered.
9. Usage guide:

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



Section 2C. Element Interface Specification

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