CMR INSTITUTE OF TECHNOLOGY

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

Improvement Test – May 2017

Sub:	Software Architectures			Code:	10IS81
Date: 25-05-17	Duration: 90 mins	Max Marks: 50	Sem: VIII	Branch:	CSE

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

		Marks	OBE CO	OBE RBT
1.	Define software architecture. Explain the factors that influence architecture.	[10]	CO1	L2
2.	Explain the activities of software architecture development.	[10]	CO1	L3
3.	What makes a "Good" architecture?	[10]	CO1	L3
4. a.	Explain different implications of software architecture.	[06]	CO1	L5
b.	Explain the importance of software architecture.	[04]	CO1	L2
5.	Explain the structure and implementation steps and consequences of access control	[10]	CO4	L4
6. a.	Explain implementation of master slave	[06]	CO4	L4
b.	Explain consequences of master slave.	[04]	CO5	L3
7.	Explain Whole part in detail.	[10]	CO5	L4
8. a.	Explain structure of broker pattern.	[06]	CO5	L3
b.	Explain dynamics of broker pattern.	[04]		

CMR INSTITUTE OF TECHNOLOGY

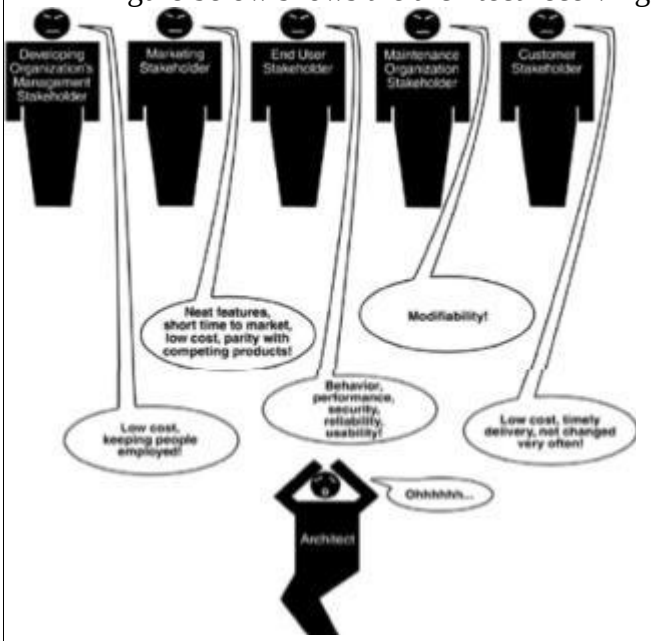
SCHEME OF EVALUATION

Improvement Test – March 2017

Sub:	Software Architectures					**Code:**	10IS81
Date:	28/3/2017	Duration:	90 mins	Max Marks:	50	**Sem:** VIII	**Branch:** CSE

Note: Answer any 5 questions. Total marks: 50

Question No		Description	Distribution of Marks		Total Marks
1.		Definition of Software Architecture. Diagram Stakeholders Developing Organization Technical Experience Architects Background	2M 2M 2M 2M 1M 1M	10M	10M
2.		Definition of Software process List of Activities Explanation	2M 1M 7M	7M	10M
3.		Process Recommendation Product Recommendation	5M 5M	10M	10M
4.	a.	Definition Implications	2M 4M	6M	10M
	b.	Importance of SA	4M	4M	
5.		Structure of Access Control Implementation Steps Consequences	4M 4M 2M	10M	10M
6.	a.	Implementation steps of Master Slave	6M	6M	10M
	b.	Benefits of master slave Liabilities of master slave	2M 2M	4M	
7.		Whole part Structure Implementation Dynamics	3M 4M 3M	10M	10M
8.	a.	Structure of broker(6 Cards 1 M each)	6M	6M	10M
	b.	Any 1 dynamic diagram Explanation	2M 2M	4M	

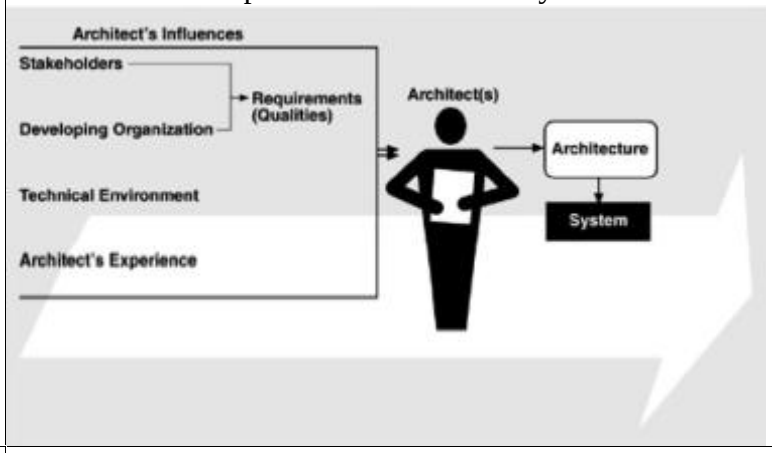
		Marks	OBE CO	OBE RBT
1.	**Define software architecture. Explain the factors that influence architecture.** The software architecture of a program or computing system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them. An architecture is the result of a set of business and technical decisions. There are many influences at work in its design, and the realization of these influences will change depending on the environment in which the architecture is required to perform. Even with the same requirements, hardware, support software, and human resources available, an architect designing a system today is likely to design a different system than might have been designed five years ago. **ARCHITECTURES ARE INFLUENCED BY SYSTEM STAKEHOLDERS** • 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".  **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	[10]	CO 1	L2

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.



2.		**Explain the activities of software architecture development.**	[10]	CO 1	L3

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

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

Understanding the requirements

- There are a variety of techniques for eliciting requirements from the stakeholders. 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.
- Another technique that helps us understand requirements is the creation of prototypes.
- Regardless of the technique used to elicit the requirements, the desired qualities of the system to be constructed determine the shape of its structure.

Creating or selecting the architecture

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

Documenting and communicating the architecture

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

Analyzing or evaluating the architecture

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

Implementing the system based on the architecture

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

Ensuring that the implementation conforms to the architecture

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

3.	**What makes a "Good" architecture?**	[10]	CO 1	L3

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

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

Process recommendations are as follows:

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

Product (structural) recommendations are as follows:

- The architecture should feature well-defined modules whose functional responsibilities are allocated on the principles of information hiding and separation of concerns.
- Each module should have a well-defined interface that encapsulates or "hides" changeable aspects from other software that uses its facilities. These interfaces

		should allow their respective development teams to work largely independent of each other. • Quality attributes should be achieved using well-known architectural tactics specific to each attribute. • The architecture should never depend on a particular version of a commercial product or tool. • Modules that produce data should be separate from modules that consume data. This tends to increase modifiability. • For parallel processing systems, the architecture should feature well-defined processors or tasks that do not necessarily mirror the module decomposition structure. • Every task or process should be written so that its assignment to a specific processor can be easily changed, perhaps even at runtime. • The architecture should feature a small number of simple interaction patterns.			
4.	a.	**Explain different implications of software architecture.** The software architecture of a program or computing system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them. Let's look at some of the implications of this definition in more detail. Architecture defines software elements • The definition makes clear that systems can and do comprise more than one structure and that no one structure can irrefutably claim to be the architecture. • The definition implies that every computing system with software has a software architecture because every system can be shown to comprise elements and the relations among them. • The behavior of each element is part of the architecture insofar as that behavior can be observed or discerned from the point of view of another element. Such behavior is what allows elements to interact with each other, which is clearly part of the architecture. • The definition is indifferent as to whether the architecture for a system is a good one or a bad one, meaning that it will allow or prevent the system from meeting its behavioral, performance, and life-cycle requirements.	[06]	CO 1	L5
	b.	**Explain the importance of software architecture.** There are fundamentally three reasons for software architecture's importance from a technical perspective. • **Communication among stakeholders**: software architecture represents a common abstraction of a system that most if not all of the system's stakeholders can use as a basis for mutual understanding, negotiation, consensus and communication. • **Early design decisions**: Software architecture manifests the earliest design decisions about a system with respect to the system's remaining development, its deployment, and its maintenance life. It is the earliest point at which design decisions governing the system to be built can be analyzed. • **Transferable abstraction of a system**: software architecture model is transferable across systems. It can be applied to other systems exhibiting similar quality attribute and functional attribute and functional requirements and can promote large-scale re-use.	[04]	CO 1	L2
5.		**Explain the structure and implementation steps and consequences of access control.** **Structure:** **Original** Implements a particular service Client Responsible for specific task ,To do this, it involves the functionality of the original in an indirect way by accessing the proxy. Proxy	[10]	CO 4	L4

Offers same interface as the original, and ensures correct access to the original. To achieve this, the proxy maintains a reference to the original it represents. Usually there is one-to-one relationship b/w the proxy and the original.

Abstract original

Provides the interface implemented by the proxy and the original. i.e, serves as abstract base class for the proxy and the original.

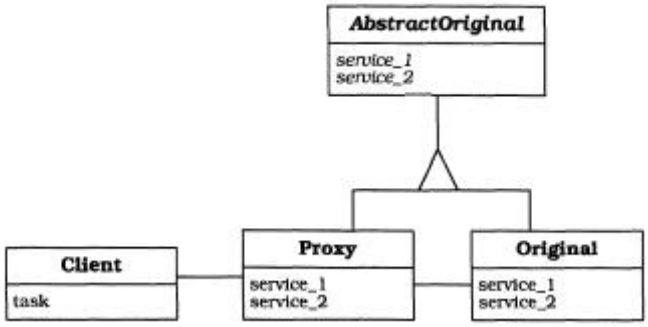
Class Client	Collaborators • Proxy
Responsibilities • Uses the interface provided by the proxy to request a particular service. • Fulfills its own task.	

Class *AbstractOriginal*	Collaborators -
Responsibilities • Serves as an abstract base class for the proxy and the original.	

Class Proxy	Collaborator • Original
Responsibilities • Provides the interface of the original to clients. • Ensures a safe, efficient and correct access to the original.	

Class Original	Collaborators -
Responsibilities • Implements a particular service.	

The OMT Diagram is as shown below



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.

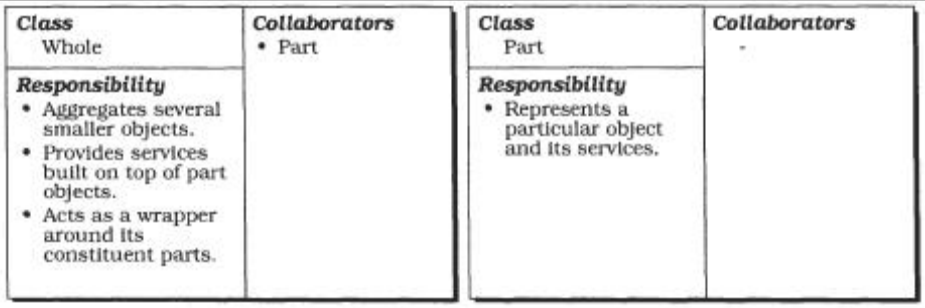
Consequences:

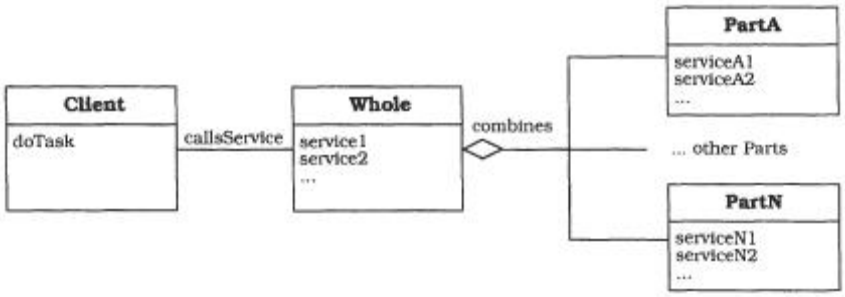
The Proxy pattern provides the following Benefits:

- **Enhanced efficiency and lower cost**
  The Virtual Proxy variant helps to implement a 'load-on-demand' strategy. This allows you to avoid unnecessary loads from disk and usually speeds up your application
- **Decoupling clients from the location of server components**
  By putting all location information and addressing functionality into a Remote Proxy

		variant, clients are not affected by migration of servers or changes in the networking infrastructure. This allows client code to become more stable and reusable. • **Separation of housekeeping code from functionality.** A proxy relieves the client of burdens that do not inherently belong to the task the client is to perform. The Proxy pattern has the following Liabilities: • **Less efficiency due to indirection** All proxies introduce an additional layer of indirection. • **Over kill via sophisticated strategies** Be careful with intricate strategies for caching or loading on demand they do not always pay.			
6.	**a.**	**Explain implementation of master slave** **1. Divide work:** Specify how the computation of the task can be split into a set equal sub tasks. Identify the sub services that are necessary to process a subtask. **2. Combine sub-task results** Specify how the final result of the whole service can be computed with the help of the results obtained from processing individual sub-tasks. **3. Specify co operation between master and slaves** • Define an interface for the subservice identified in step1 it will be implemented by the slave and used by the master to delegate the processing of individual subtask. • One option for passing subtasks from the master to the slaves is to include them as a parameter when invoking the subservice. • Another option is to define a repository where the master puts sub tasks and the slaves fetch them. 4. **Implement the slave component**s according to the specifications developed in previous step. 5. **Implement the master** according to the specifications developed in step 1 to 3 • There are two options for dividing a task into subtasks. The first is to split work into a fixed number of subtasks. The second option is to define as many subtasks as necessary or possible. • Use strategy pattern to support dynamic exchange and variations of algorithms for subdividing a task.	[06]	CO 4	L4
	b.	**Explain consequences of master slave.** The Master-Slave design pattern provides several Benefits: • **Exchangeability and extensibility** By providing an abstract slave class, it is possible to exchange existing slave implementations or add new ones without major changes to the master. • **Separation of concerns** The introduction of the master separates slave and client code from the code for partitioning work, delegating work to slaves, collecting the results from the slaves, computing the final result and handling slave failure or inaccurate slave results. • **Efficiency** The Master-Slave pattern for parallel computation enables you to speed up the performance of computing a particular service when implemented carefully The Master-Slave design pattern has certain Liabilities: • **Feasibility** It is not always feasible • **Machine dependency** It depends on the architecture of the m/c on which the program runs. • **Hard to implement** Implementing Master-Slave is not easy, especially for parallel computation. • **Portability** Master-Slave structures are difficult or impossible to transfer to other machines.	[04]	CO 5	L3

7.		**Explain Whole part in detail.** 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. **Structure:** 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.	[10]	CO 5	L4





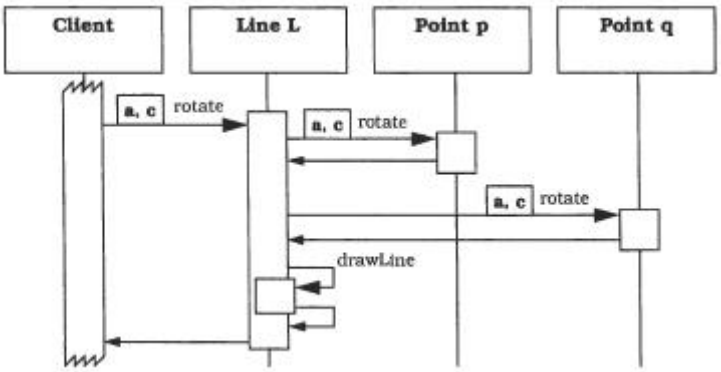
Dynamics:

The following scenario illustrates the behavior of a Whole-Part structure. We use the two-dimensional rotation of a line within a CAD system as an example. The line acts as a Whole object that contains two points p and **q** as Parts. A client asks the line object to rotate around the point c and passes the rotation angle as an argument.

$$\dot{p}' = \begin{bmatrix} \cos a & -\sin a \\ \sin a & \cos a \end{bmatrix} \cdot (\dot{p} - \dot{c}) + \dot{c}$$

The scenario consists of four phases:

- A client invokes the rotate method of the line L and passes the angle a and the rotation center c as arguments.
- The line L calls the rotate method of the point p.
- The line L calls the rotate method of the point q.
- The line L redraws itself using the new positions of p **I** and q **I** as endpoints.



Implementation:

1. Design the public interface of the whole

- Analyze the functionality the whole must offer to its clients. Only consider th 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.

8a. Explain structure of broker pattern. [06] CO 5 L3

The broker architectural pattern comprises six types of participating components.

Server

- **Client**
- **Client side proxy**
- **Server side proxy**
- **Broker**
- **Bridge**

Class Client	***Collaborators***
Responsibility • Implements user functionality. • Sends requests to servers through a client-side proxy.	• Client-side Proxy • Broker

Class Server	***Collaborators***
Responsibility • Implements services. • Registers itself with the local broker. • Sends responses and exceptions back to the client through a server-side proxy.	• Server-side Proxy • Broker

Class: Broker	Collaborators
Responsibility • (Un-)registers servers. • Offers APIs. • Transfers messages. • Error recovery. • Interoperates with other brokers through bridges. • Locates servers.	• Client • Server • Client-side Proxy • Server-side Proxy • Bridge

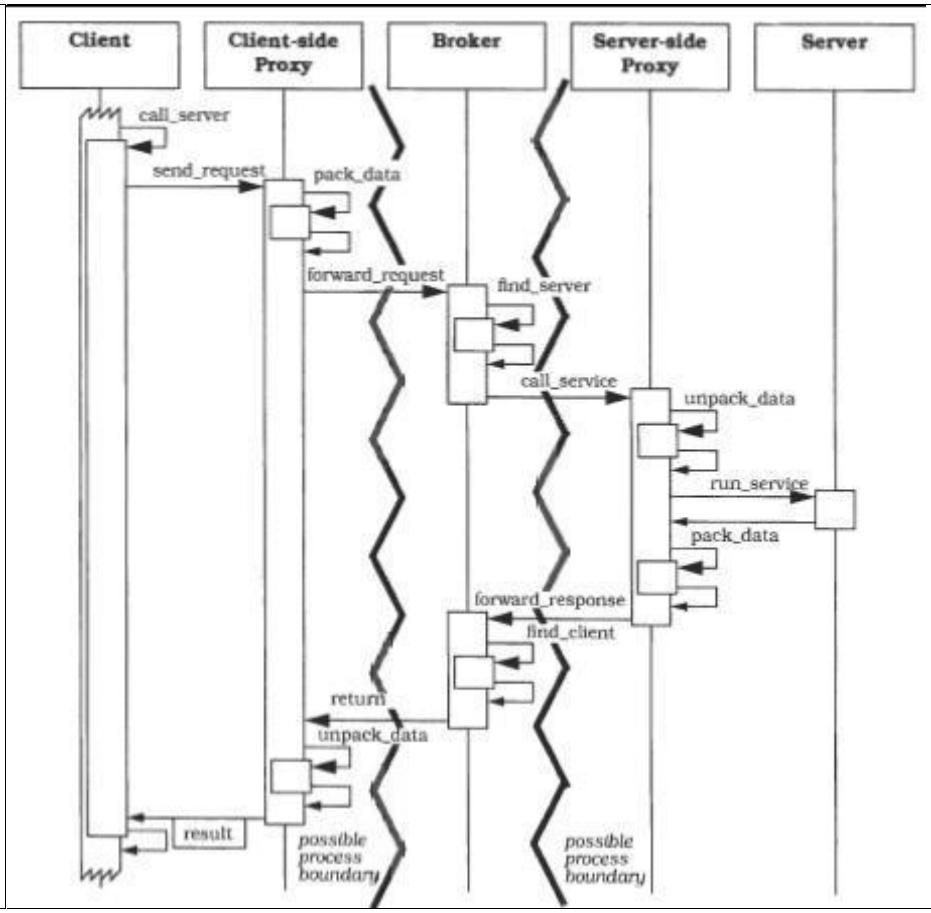
Class: Bridge	Collaborators
Responsibility • Encapsulates network-specific functionality. • Mediates between the local broker and the bridge of a remote broker.	• Broker • Bridge

Class: Client-side Proxy	Collaborators
Responsibility • Encapsulates system-specific functionality. • Mediates between the client and the broker.	• Client • Broker

Class: Server-side Proxy	Collaborators
Responsibility • Calls services within the server. • Encapsulates system-specific functionality. • Mediates between the server and the broker.	• Server • Broker

b. Explain dynamics of broker pattern. [04]

Scenario illustrates the behavior when a client sends a request to a local server. In this scenario we describe a synchronous invocation, in which the client blocks until it gets a response from the server. The broker may also support asynchronous invocations, allowing clients to execute further tasks without having to wait for a response.


Client
Client-side Proxy
Broker
Server-side Proxy
Server
call_server
send_request
pack_data
forward_request
find_server
call_service
unpack_data
run_service
pack_data
forward_response
find_client
return
unpack_data
result
possible process boundary
possible process boundary