CMR
INSTITUTE OF
TECHNOLOGY

USN | 1 | C | R | | | | | | | |

Improvement Test – May 2017

Sub:	Software Architectures							Code:	10IS81
Date:	26/05/2017	Duration:	90 mins	Max Marks:	50	Sem:	VIII	Branch:	ISE

Q No.	Description	Distribution of Marks	Total Marks
1	Explain KWIC with problem statement & following architectural style i) Abstract data types ii) Main Program/Subroutine with shared data	2 M (Problem Statement) 1 M (Diagram) 3 M (ADT) 1 M (Diagram) 3 M (MP/S)	[10]
2	Briefly explain the concept of documenting a view.	1 M (Listing) 9 M (seven parts of a documented view)	[10]
3	List and explain the key concepts of availability tactics?	1 M (Diagram) 3 M (each tactic)	[10]
4	Explain common software architectures.	3 M (Diagram) 2 M (Module) 3 M (Component & Connector) 2 M (Allocation)	[10]
5	Explain Master Slave pattern with an example.	1 M (10 Sections)	[10]
6	Explain Broker Architecture with an example	1 M (10 Sections)	[10]
7	Explain ABC with respect to software process and activities involved in creating software architecture.	1 M (Diagram) 1 M (Listing Activities) 8 M	[10]
8	List and explain the steps to implement a Whole Part structure.	1 M (10 Sections)	[10]

CMR
INSTITUTE OF
TECHNOLOGY

USN | 1 | C | R | | | | | | | |

Improvement Test – May 2017

Sub:	Software Architectures							Code:	10IS81
Date:	26/05/2017	Duration:	90 mins	Max Marks:	50	Sem:	VIII	Branch:	ISE

1 a. Explain KWIC with problem statement & following architectural style
i) Abstract data types
ii) Main Program/Subroutine with shared data

Soln.

Problem Statement:

"The KWIC [Key Word in Context] index system accepts an ordered set of lines, each line is an ordered set of words, and each word is an ordered set of characters. Any line may be ``circularly shifted" by repeatedly removing the first word and appending it at the end of the line. The KWIC index system outputs a listing of all circular shifts of all lines in alphabetical order."

ii) Main Program/Subroutine with shared data

- The solution decomposes the problem according to the four basic functions performed: input, shift, alphabetize, and output. These computational components are coordinated as subroutines by a main program that sequences through them in turn. Data is communicated between the components through shared storage.
- Communication between the computational components and the shared data is an unconstrained read write protocol. This is made possible by the fact that the coordinating program guarantees sequential access to the data.

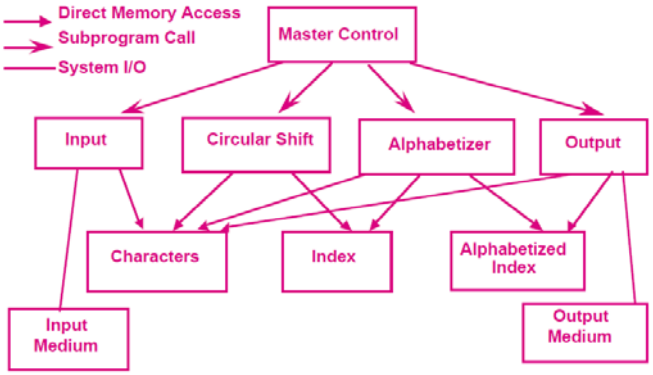


Figure 6: KWIC – Shared Data Solution

- Using this solution data can be represented efficiently, since computations can share the same storage. It has a number of serious drawbacks in terms of its ability to handle changes.
- In particular, a change in data storage format will affect almost all of the modules. Similarly changes in the overall processing algorithm and enhancements to system function are not easily accommodated.
- Finally, this decom-position is not particularly supportive of reuse.

i) *Abstract Data Types*

- In this case data is no longer directly shared by the computational components. Instead, each module provides an interface that permits other components to access data only by invoking procedures in that interface.
- This solution provides the same logical decomposition into processing modules as the first. However, it has a number of advantages over the first solution when design changes are considered. In particular, both algorithms

and data representations can be changed in individual modules without affecting others. Moreover, reuse is better supported than in the first solution because modules make fewer assumptions about the others with which they interact.

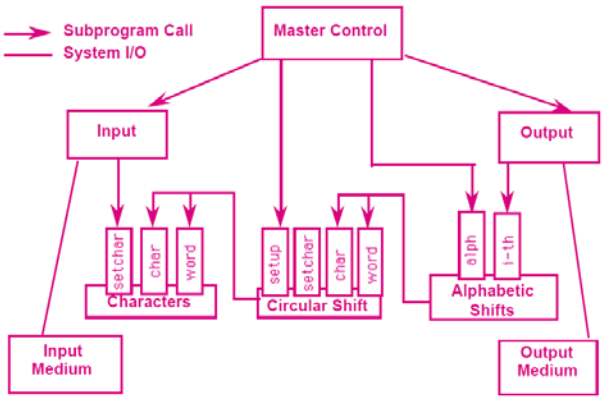


Figure 7: KWIC – Abstract Data Type Solution

The main problem is that to add new functions to the system, the implementer must either modify the existing modules—compromising their simplicity and integrity—or add new modules that lead to performance penalties.

2 a. Briefly explain the concept of documenting a view.

Soln.

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

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

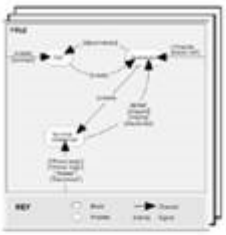
   - Sometimes the primary presentation can be tabular; tables are often a superb way to convey a large amount of information compactly. An example of a textual primary presentation is the A-7E module decomposition view illustrated in Chapter 3. A textual presentation still carries the obligation to present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

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

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

Views

1. Primary Presentation of the View

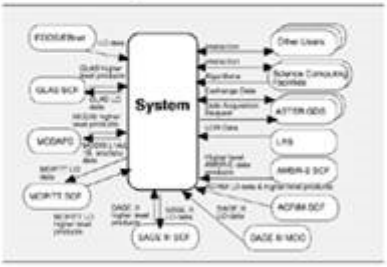


OR

Textual version of the primary presentation

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

3. Context Diagram



4. Variability Guide

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

6. Glossary of Terms

7. Other Information

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

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

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

   - rationale, explaining why the decisions reflected in the view were made and why alternatives were rejected.
   - analysis results, which justify the design or explain what would have to change in the face of a modification.

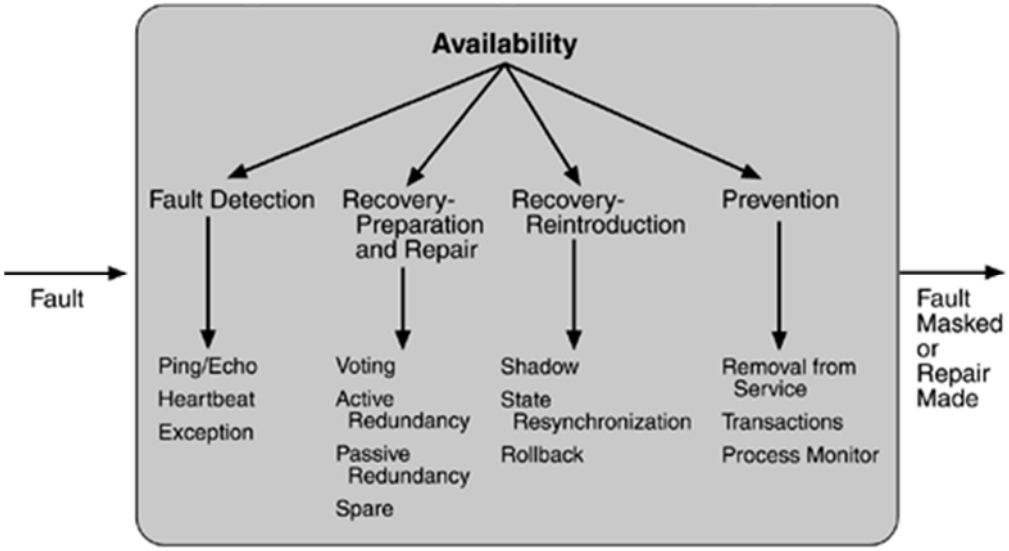
- assumptions reflected in the design.

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

7. **Other information.** The precise contents of this section will vary according to the standard practices of your organization. They might include management information such as authorship, configuration control data, and change histories. Or the architect might record references to specific sections of a requirements document to establish traceability. Strictly speaking, information such as this is not architectural. Nevertheless, it is convenient to record it alongside the architecture, and this section is provided for that purpose. In any case, the first part of this section must detail its specific contents.

3 a. List and explain the key concepts of availability tactics?

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



Fault detection- Three widely used tactics for recognizing faults are ping/echo, heartbeat, and exceptions.

Soln.

- **Ping/echo.** One component issues a ping and expects to receive back an echo, within a predefined time, from the component under scrutiny. This can be used within a group of components mutually responsible for one task.

- **Heartbeat (dead man timer).** In this case one component emits a heartbeat message periodically and another component listens for it. If the heartbeat fails, the originating component is assumed to have failed and a fault correction component is notified.

- **Exceptions.** One method for recognizing faults is to encounter an exception, which is raised when one of the fault classes is recognized. The exception handler typically executes in the same process that introduced the exception.

Fault recovery- Fault recovery consists of preparing for recovery and making the system repair. Some preparation and repair tactics follow.

- **Voting.** Processes running on redundant processors each take equivalent input and compute a simple output value that is sent to a voter. If the voter detects deviant behavior from a single processor, it fails it. The voting algorithm can be "majority rules" or "preferred component" or some other algorithm. This method is used to correct faulty operation of algorithms or failure of a processor and is often used in control systems.

- **Active redundancy** (hot restart). All redundant components respond to events in parallel. Consequently, they are all in the same state. The response from only one component is used (usually the first to respond), and the rest are discarded. When a fault occurs, the downtime of systems using this tactic is usually milliseconds since the backup is current and the only time to recover is the switching time. Active redundancy is often used in a client/server configuration, such as database management systems, where quick responses are necessary even when a fault occurs. In a highly available distributed system, the redundancy may be in the communication paths.
- **Passive redundancy** (warm restart/dual redundancy/triple redundancy). One component (the primary) responds to events and informs the other components (the standbys) of state updates they must make. When a fault occurs, the system must first ensure that the backup state is sufficiently fresh before resuming services. This approach is also used in control systems, often when the inputs come over communication channels or from sensors and have to be switched from the primary to the backup on failure.
- **Spare.** A standby spare computing platform is configured to replace many different failed components. It must be rebooted to the appropriate software configuration and have its state initialized when a failure occurs. Making a checkpoint of the system state to a persistent device periodically and logging all state changes to a persistent device allows for the spare to be set to the appropriate state.
- **Shadow operation**. A previously failed component may be run in "shadow mode" for a short time to make sure that it mimics the behavior of the working components before restoring it to service.
- **State resynchronization**. The passive and active redundancy tactics require the component being restored to have its state upgraded before its return to service. The updating approach will depend on the downtime that can be sustained, the size of the update, and the number of messages required for the update
- **Checkpoint/rollback**. A checkpoint is a recording of a consistent state created either periodically or in response to specific events. Sometimes a system fails in an unusual manner, with a detectably inconsistent state.

Fault prevention-

- **Removal from service.** This tactic removes a component of the system from operation to undergo some activities to prevent anticipated failures.
- **Transactions**. A transaction is the bundling of several sequential steps such that the entire bundle can be undone at once. Transactions are used to prevent any data from being affected if one step in a process fails and also to prevent collisions among several simultaneous threads accessing the same data.
- **Process monitor**. Once a fault in a process has been detected, a monitoring process can delete the nonperforming process and create a new instance of it, initialized to some appropriate state as in the spare tactic.

4 a. Explain common software architectures.

1. Module

Soln.

Module-based structures include the following.

- ***Decomposition:*** The units are modules related to each other by the "is a submodule of " relation, showing how larger modules are decomposed into smaller ones recursively until they are small enough to be easily understood. Modules in this structure represent a common starting point for design, as the architect enumerates what the units of software will have to do and assigns each item to a module for subsequent design and eventual

implementation. The decomposition structure provides a large part of the system's modifiability, by ensuring that likely changes fall within the purview of at most a few small modules.

- ***Uses:*** The units of this important but overlooked structure are also modules procedures or resources on the interfaces of modules. The units are related by the uses relation. One unit uses another if the correctness of the first requires the presence of a correct version of the second. The uses structure is used to engineer systems that can be easily extended to add functionality or from which useful functional subsets can be easily extracted.
- ***Layered:*** When the uses relations in this structure are carefully controlled in a particular way, a system of layers emerges, in which a layer is a coherent set of related functionality. 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 classes. The relation is "inherits-from" or "is-an-instance-of." This view supports reasoning about collections of similar behavior or capability and parameterized differences which are captured by subclassing. The class structure allows us to reason about re-use and the incremental addition of functionality.

2. **Component-and-Connector**

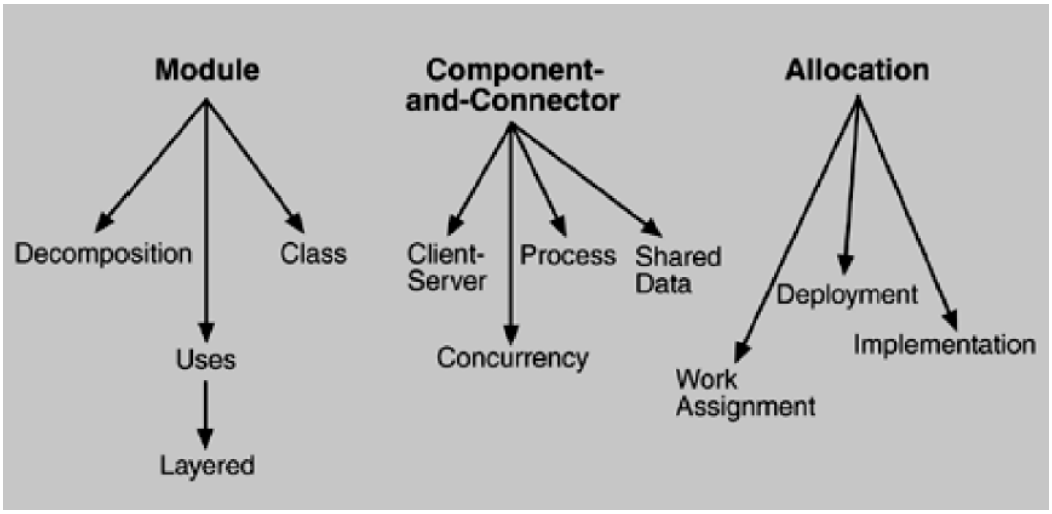
These structures include the following.

- ***Process, or communicating processes:*** Like all component-and-connector structures, this one is orthogonal to the module-based structures and deals with the dynamic aspects of a running system. The units here are processes or threads that are connected with each other by communication, synchronization, and/or exclusion operations. The relation in this is attachment, showing how the components and connectors are hooked together. The process structure is important in helping to engineer a system's execution performance and availability.
- ***Concurrency:*** This component-and-connector structure allows the architect to determine opportunities for parallelism and the locations where resource contention may occur. The units are components and the connectors are "logical threads." A logical thread is a sequence of computation that can be allocated to a separate physical thread later in the design process. The concurrency structure is used early in design to identify the requirements for managing the issues associated with concurrent execution.
- ***Shared data, or repository:*** This structure comprises components and connectors that create, store, and access persistent data. If the system is in fact structured around one or more shared data repositories, this structure is a good one to illuminate. It shows how data is produced and consumed by runtime software elements, and it can be used to ensure good performance and data integrity.
- ***Client-server:*** If the system is built as a group of cooperating clients and servers, this is a good component-and-connector structure to illuminate. The components are the clients and servers, and the connectors are protocols and messages they share to carry out the system's work. This is useful for separation of concerns, for physical distribution, and for load balancing.

3. **Allocation:** Allocation structures include the following.

- ***Deployment***. The deployment structure shows how software is assigned to hardware-processing and communication elements. The elements are software, hardware entities (processors), and communication pathways. Relations are "allocated-to," showing on which physical units the software elements reside, and "migrates-to," if the allocation is dynamic. This view allows an engineer to reason about performance, data integrity, availability, and security. It is of particular interest in distributed or parallel systems.
- ***Implementation***. This structure shows how software elements (usually modules) are mapped to the file structure(s) in the system's development, integration, or configuration control environments. This is critical for the management of development activities and build processes.
- ***Work assignment***. This structure assigns responsibility for implementing and integrating the modules to the appropriate development teams. Having a work assignment structure as part of the architecture makes it clear that the decision about who does the work has architectural as well as management implications. The architect will

know the expertise required on each team. Also, on large multi-sourced distributed development projects, the work assignment structure is the means for calling out units of functional commonality and assigning them to a single team, rather than having them implemented by everyone who needs them.



5 a. Explain Master Slave pattern with an example.

*The **Master-Slave** design pattern supports fault tolerance, parallel computation and computational accuracy. A master component distributes work to identical slave components and computes a final result from the results these slaves return.*

Context:

- Portioning work into semantically identical subtasks

Problem:

Divide and conquer: here work is partitioned into several equal subtasks that are processed independently. The result of the whole calculation is computed from the results provided by each partial process. Several forces arise when implementing such a structure

- Clients should not be aware that the calculation is based on the 'divide and conquer' principle.
- Neither clients nor the processing of subtasks should depend on the algorithms for partitioning work and assembling the final result.
- It can be helpful to use different but semantically identical implementations for processing subtasks.
- Processing of subtasks sometimes need co-ordination for ex. In simulation applications using the finite element method.

Solution:

Soln.

- Introduce a co ordination instance b/w clients of the service and the processing of individual subtasks.
- A master component divides work into equal subtasks, delegates these subtasks to several independent but semantically identical slave components and computes a final result from the partial results the slaves return.
- The general principle is found in three application areas
  - Fault tolerance - Failure of service executions can be detected and handled
  - Parallel computing - A complex task is divided into a fixed number of identical sub-tasks that are executed in parallel.
  - Computational accuracy - Inaccurate results can be detected and handled.

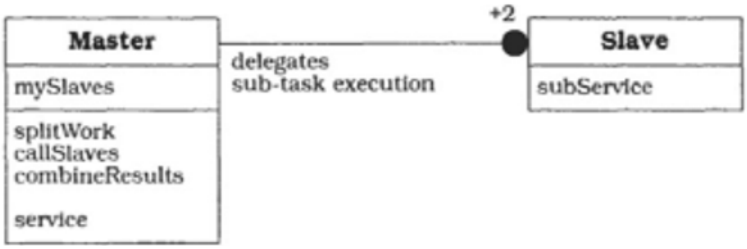
Structure:

- **Master component:**
  - Provides the service that can be solved by applying the 'divide and conquer' principle.
  - It implements functions for partitioning work into several equal subtasks, starting and controlling their processing and computing a final result from all the results obtained.
  - It also maintains references to all slaves instances to which it delegates the processing of subtasks.
- **Slave component:**
  - Provides a sub-service that can process the subtasks defined by the master
  - There are at least two instances of the slave component connected to the master.

Class	Collaborators
Master	• Slave
Responsibility	
• Partitions work among several slave components • Starts the execution of slaves • Computes a result from the sub-results the slaves return.	

Class	Collaborators
Slave	-
Responsibility	
• Implements the sub-service used by the master.	

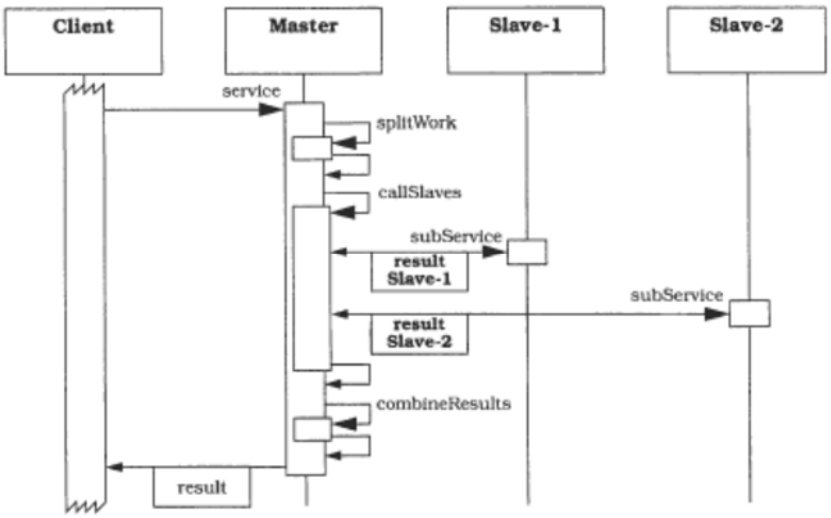
The structure defined by the Master-Slave pattern is illustrated by the following OMT diagram.



- ***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.



- ***Implementation:***
  - **Divide work:**

a. Specify how the computation of the task can be split into a set equal sub tasks.

b. Identify the sub services that are necessary to process a subtask.

- **Combine sub-task results**

a. Specify how the final result of the whole service can be computed with the help of the results obtained from processing individual sub-tasks.

- **Specify co operation between master and slaves**
  a. 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.
  b. 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.
- **Implement the slave component**s according to the specifications developed in previous step.
- **Implement the master** according to the specifications developed in step 1 to 3
  1. There are two options for dividing a task into subtasks.
     a. The first is to split work into a fixed number of subtasks.
     b. The second option is to define as many subtasks as necessary or possible.
  2. Use strategy pattern to support dynamic exchange and variations of algorithms for subdividing a task.

Variants:

There are 3 application areas for master slave pattern.

- **Master-slave for fault tolerance:** In this variant the master just delegates the execution of a service to a fixed number of replicated implementations, each represented by a slave.
- **Master-slave for parallel computation:** In this variant the master divides a complex task into a number of identical sub-tasks, each of which is executed in parallel by a separate slave.
- **Master-slave for computational concurrency:** In this variant the execution of a service is delegated to at least three different implementations, each of which is a separate slave.

Other variants

- **Slaves as processes:** To handle slaves located in separate processes, you can extend the original Master-Slave structure with two additional components
- **Slaves as threads:** In this variant the master creates the threads, launches the slaves, and waits for all threads to complete before continuing with its own computation.
- **Master-slave with slave co ordination:** In this case the computation of all slaves must be regularly suspended for each slave to coordinate itself with the slaves on which it depends, after which the slaves resume their individual computation.

Known uses:

a. **Matrix multiplication**. Each row in the product matrix can be computed by a separate slave.
b. **Transform-coding** an image, for example in computing the discrete cosine transform (DCT) of every 8 x 8 pixel block in an image. Each block can be computed by a separate slave.
c. Computing the **cross-correlation** of two signals
d. The **Workpool model** applies the master-slave pattern to implement process control for parallel
e. computing
f. The concept of **Gaggles** builds upon the principles of the Master-Slave pattern to handle 'plurality' in an object-oriented software system. **A *gaggle*** represents a set of replicated service objects.

Consequences: The Master-Slave design pattern provides several *Benefits:*

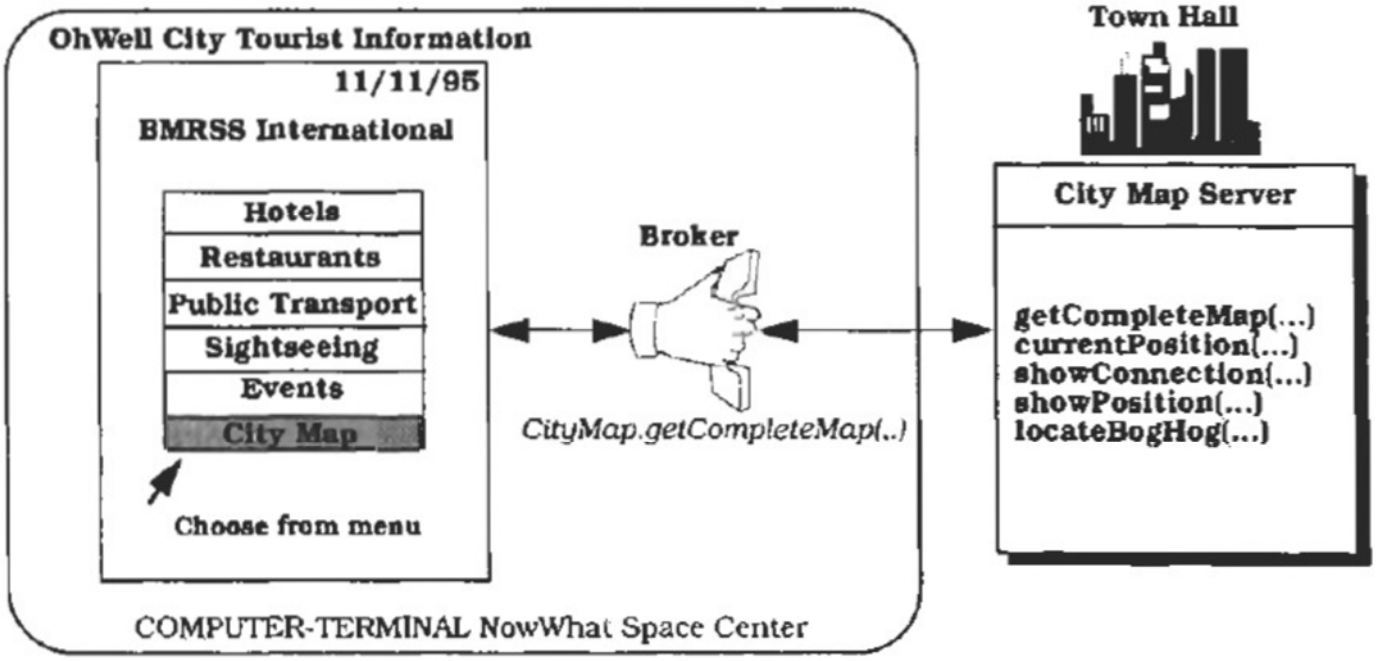
- **Exchangeability and extensibility:** By providing an abstract slave class, it is possible to exchange existing slave implementations or add new ones without major changes to the master.
- **Separation of concerns:** The introduction of the master separates slave and client code from the code for 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. This may decrease the changeability and portability.
- **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

6 a. Explain Broker Architecture with an example

The broker architectural pattern can be used to structure distributed software systems with decoupled components that interact by remote service invocations. A broker component is responsible for coordinating communication, such as requests, as well as for transmitting results and exceptions.



Soln. ***Example:*** Suppose we are developing a city information system (CIS) designed to run on a wide area network. Some computers in the network host one or more services that maintain information about events, restaurants, hotels, historical monuments or public transportation. Computer terminals are connected to the network. Tourists throughout the city can retrieve information in which they are interested from the terminals using a World Wide Web (WWW) browser. This front-end software supports the on-line retrieval of information from the appropriate servers and its display on the screen. The data is distributed across the network, and is not all maintained in the terminals.

Context:
Your environment is a distributed and possibly heterogeneous system with independent co-operating components.
Problem:
Building a complex software system as a set of decoupled and interoperating components, rather than as a monolithic application, results in greater flexibility, maintainability and changeability. By partitioning functionality into independent components the system becomes potentially distributable and scalable. Services for adding, removing, exchanging, activating and locating components are also needed. From a developer's viewpoint, there should essentially be no difference between developing software for centralized systems and developing for distributed ones. We have to balance the following forces:

- Components should be able to access services provided by other through remote, location-transparent service invocations.
- You need to exchange, add or remove components at run time.
- The architecture should hide system and implementation-specific details from the users of component and services.

Solution:

- Introduce a broker component to achieve better decoupling of clients and servers.
- Servers registers themselves with the broker make their services available to clients through method interfaces.
- Clients access the functionality of servers by sending requests via the broker.
- A broker's tasks include locating the appropriate server, forwarding the request to the server, and transmitting results and exceptions back to the client.
- The Broker pattern reduces the complexity involved in developing distributed applications, because it makes distribution transparent to the developer.

Structure:

The broker architectural pattern comprises six types of participating components.

- **Server:**

o Implements objects that expose their functionality through interfaces that consists of operations and attributes.
o These interfaces are made available either through an interface definition language (IDL) or through a binary standard.
o There are two kind of servers:

a. Servers offering common services to many application domains.
b. Servers implementing specific functionality for a single application domain or task.

- **Client:**

o Clients are applications that access the services of at least one server.
o To call remote services, clients forward requests to the broker. After an operation has executed they receive responses or exceptions from the broker.
o Interaction b/w servers and clients is based on a dynamic model, which means that servers may also act as clients.

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

- **Brokers:**

o It is a messenger that is responsible for transmission of requests from clients to servers, as well as the transmission of responses and exceptions back to the client.
o It offers API'S to clients and servers that include operations for registering servers and for invoking server methods.
o When a request arrives from server that is maintained from local broker, the broker passes the request directly to the server. If the server is currently inactive, the broker activates it.
o If the specified server is hosted by another broker, the local broker finds a route to the remote broker and forwards the request this route.
o Therefore there is a need for brokers to interoperate through bridges.

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

Implementation:

1) Define an object existing model, or use an existing model.

Each object model must specify entities such as object names, requests, objects, values, exceptions, supported types, interfaces and operations.

2) Decide which kind of component-interoperability the system should offer.

a. You can design for interoperability either by specifying a binary standard or by introducing a high-level IDL.
b. IDL file contains a textual description of the interfaces a server offers to its clients.
c. The binary approach needs support from your programming language.

3) Specify the API'S the broker component provides for collaborating with clients and servers.

a. Decide whether clients should only be able to invoke server operations statically, allowing clients to bind the invocations at complete time, or you want to allow dynamic invocations of servers as well.
b. This has a direct impact on size and no. of API'S.

4) **Use proxy objects to hide implementation details from clients and servers**.

a. Client side proxies package procedure calls into message and forward these messages to the local broker component.
b. Server side proxies receive requests from the local broker and call the methods in the interface implementation of the corresponding server.

5) Design the broker component in parallel with steps 3 and 4

During design and implementations, iterate systematically through the following steps

5.1 Specify a detailed on-the-wire protocol for interacting with client side and server side proxies.
5.2 A local broker must be available for every participating machine in the network.
5.3 When a client invokes a method of a server the broker system is responsible for returning all results and exceptions back to the original client.
5.4 If the provides do not provide mechanisms for marshalling and un marshalling parameters results, you must include functionality in the broker component.
5.5 If your system supports asynchronous communication b/w clients and servers, you need to provide message buffers within the broker or within the proxies for temporary storage of messages.
5.6 Include a directory service for associating local server identifiers with the physical location of the corresponding servers in the broker.
5.7 When your architecture requires system-unique identifiers to be generated dynamically during server registration, the broker must offer a name service for instantiating such names.
5.8 If your system supports dynamic method invocation the broker needs some means for maintaining type information about existing servers.
5.9 Plan the broker's action when the communication with clients, other brokers, or servers fails.

6) Develop IDL compliers

An IDL compiler translates the server interface definitions to programming language code. When many

programming languages are in use, it is best to develop the compiler as afrarnework that allows the developer to add his own code generators.

Variants:

- **Direct communication broker system:**
  a. We may sometime choose to relax the restriction that clients can only forward requests through the local brokers for efficiency reasons
  b. In this variant, clients can communicate with server directly.
  c. Broker tells the clients which communication channel the server provides.
  d. The client can then establish a direct link to the requested server
- **Message passing broker system:**
  a. This variant is suitable for systems that focus on the transmission of data, instead of implementing a remote produce call abstraction.
  b. In this context, a message is a sequence of raw data together with additional information that specifies the type of a message, its structure and other relevant attributes.
  c. Here servers use the type of a message to determine what they must do, rather than offering services that clients can invoke.

Trader system:

a. A client request is usually forwarded to exactly one uniquely – identified servers.
b. In a trader system, the broker must know which server(s) can provide the service and forward the request to an appropriate server.
c. Here client side proxies use service identifiers instead of server identifiers to access server functionality.
d. The same request might be forwarded to more than one server implementing the same service.

Adapter broker system:

a. Adapter layer is used to hide the interfaces of the broker component to the servers using additional layer to enhance flexibility
b. This adapter layer is a part of the broker and is responsible for registering servers and interacting with servers.
c. By supplying more than one adapter, support different strategies for server granularity and server location.
d. Example: use of an object oriented database for maintaining objects.

- **Callback broker system:**

a. Instead of implementing an active communication model in which clients produce requests and servers consume then and also use a reactive model.
b. It's a reactive model or event driven, and makes no distinction b/w clients and servers.
c. Whenever an event arrives, the broker invokes the call back method of the component that is registered to react to the event
d. The execution of the method may generate new events that in turn cause the broker to trigger new call back method invocations.

Known uses:

- CORBA
- SOM/DSOM
- OLE 2.x
- WWW
- ATM-P

Consequences:

The broker architectural pattern has some important *Benefits*:

- **Location transparency:** Achieved using the additional 'broker' component
- **Changeability and extensibility of component:** If servers change but their interfaces remain the same, it has no functional impact on clients.

- **Portability of a broker system:** Possible because broker system hides operating system and network system details from clients and servers by using indirection layers such as API'S, proxies and bridges.
- **Interoperability between different broker systems:** Different Broker systems may interoperate if they understand a common protocol for the exchange of messages.
- **Reusability:** When building new client applications, you can often base the functionality of your application on existing services.

The broker architectural pattern has some important *Liabilities:*

- **Restricted efficiency:** Broker system is quite slower in execution.
- **Lower fault tolerance:** Compared with a non-distributed software system, a Broker system may offer lower fault tolerance.

Following aspect gives benefits as well as liabilities.

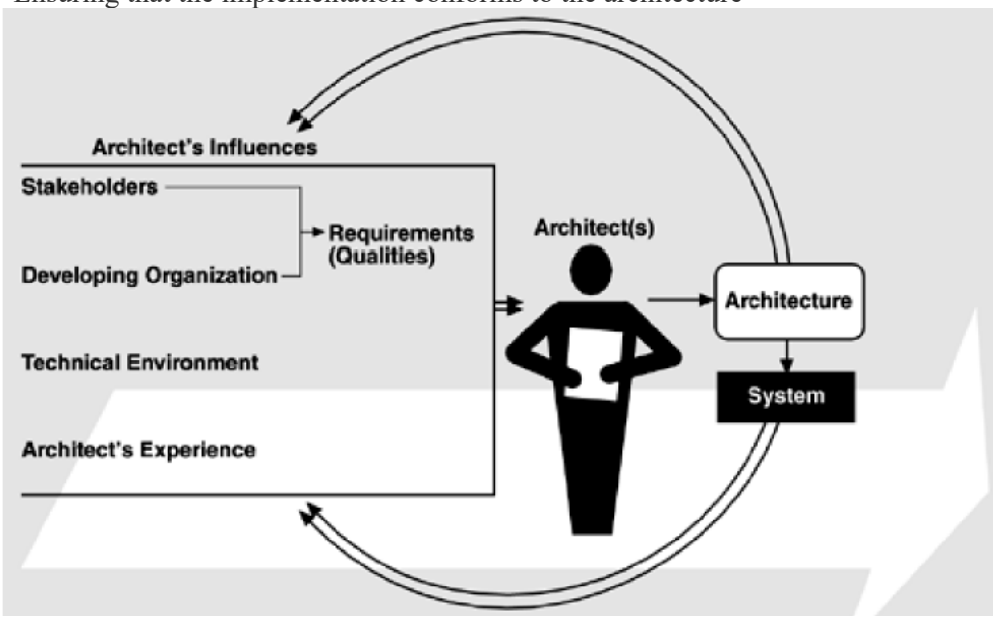
Testing and debugging: Testing is more robust and easier itself to test. However it is a tedious job because of many components involved.

7 a. Explain ABC with respect to software process and activities involved in creating software architecture.

Software process is the term given to the organization, ritualization, and management of software development activities. These activities include the following:

- Creating the business case for the system
- Understanding the requirements
- Creating or selecting the architecture
- Documenting and communicating the architecture
- Analyzing or evaluating the architecture
- Implementing the system based on the architecture
- Ensuring that the implementation conforms to the architecture

Soln.



ARCHITECTURE ACTIVITIES

1. Creating the Business Case for the System

Creating a business case is broader than simply assessing the market need for a system. It is an important step in creating and constraining any future requirements.

- 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 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.

2. Understanding the Requirements

There are a variety of techniques for eliciting requirements from the stakeholders. For example, 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.

One fundamental decision with respect to the system being built is the extent to which it is a variation on other systems that have been constructed. Since it is a rare system these days that is not similar to other systems, requirements elicitation techniques extensively involve understanding these prior systems' characteristics.

Another technique that helps us understand requirements is the creation of prototypes. Prototypes may help to model desired behavior, design the user interface, or analyze resource utilization. This helps to make the system "real" in the eyes of its stakeholders and can quickly catalyze decisions on the system's design and the design of its user interface.

Regardless of the technique used to elicit the requirements, the desired qualities of the system to be constructed determine the shape of its architecture. Specific tactics have long been used by architects to achieve particular quality attributes. An architectural design embodies many tradeoffs, and not all of these tradeoffs are apparent when specifying requirements. It is not until the architecture is created that some tradeoffs among requirements become apparent and force a decision on requirement priorities.

3. Creating or Selecting the Architecture

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.

4. 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. Toward this end, the architecture's documentation should be informative, unambiguous, and readable by many people with varied backgrounds.

5. Analyzing or Evaluating the Architecture

In any design process there will be multiple candidate designs considered. Some will be rejected immediately. Others will contend for primacy. Choosing among these 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. Becoming more widespread are analysis techniques to evaluate the quality attributes that an architecture imparts to a system. Scenario-based techniques provide one of the most general and effective approaches for evaluating an architecture.

6. Implementing 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. Having an environment or infrastructure that actively assists developers in creating and maintaining the architecture is better.

7. Ensuring Conformance to an Architecture

Finally, when an architecture is created and used, it goes into a maintenance phase. Constant vigilance is required to ensure that the actual architecture and its representation remain faithful to each other during this phase. Although work in this area is comparatively immature, there has been intense activity in recent years.

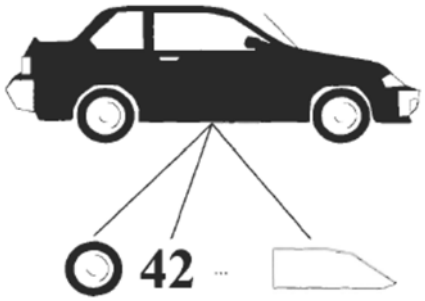
8 a. List and explain the steps to implement a Whole Part structure.

Soln.

The 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 themselves may be composed of even smaller objects such as circles and polygons. It is the responsibility of the car object to implement functionality that operates on the car as a whole, such as rotating or drawing.



Context:

Implementing aggregate objects

Problem:

- In almost every software system objects that are composed of other objects exists. Such aggregate objects do not represent loosely-coupled set of components. Instead, they form units that are more than just a mere collection of their parts.
- The combination of the parts makes new behavior emerge- such behavior is called emergent behavior.
- We need to balance following forces when modeling such structures;
  a) A complex object should either be decomposed into smaller objects, or composed of existing objects, to support reusability, changeability and the recombination of the constituent objects in other types of aggregate.
  b) Clients should see the aggregate object as an atomic object that does not allow any direct access to its constituent parts.

Solution:

- Introduce a component that encapsulates smaller objects and prevents clients from accessing theses constitutes parts directly.
- Define an interface for the aggregate that is the only means of access to the functionalities of the encapsulated objects allowing the aggregate to appear as semantic unit.
- The principle of whole-part pattern is applicable to the organization of three types of relationship
  a. An *assembly-parts* relationship which differentiation b/w a product and its parts or subassemblies.
  b. A *container-contents* relationship, in which the aggregated object represents a container.
  c. The *collection-members* relationship, which helps to group similar objects.

Structure:

The Whole-Part pattern introduces two types of participant:

- **Whole**

a. Whole object represents an aggregation of smaller objects, which we call parts.
b. It forms a semantic grouping of its parts in that it co ordinates and organizes their collaboration.
c. 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.

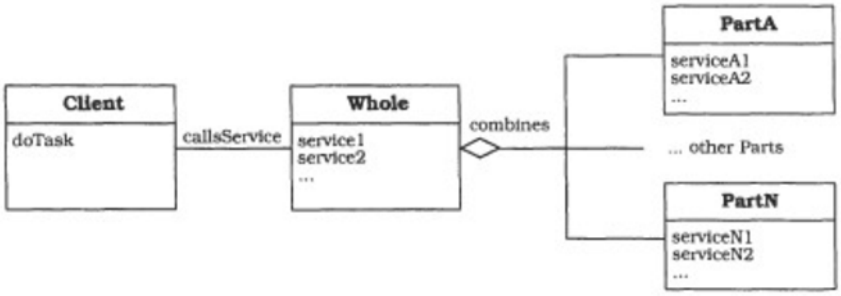
- **Part**

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

Class Whole	Collaborators • Part
Responsibility • Aggregates several smaller objects. • Provides services built on top of part objects. • Acts as a wrapper around its constituent parts.	

Class Part	Collaborators -
Responsibility • Represents a particular object and its services.	

Static relationship between whole and its part are illustrated in the OMT diagram below



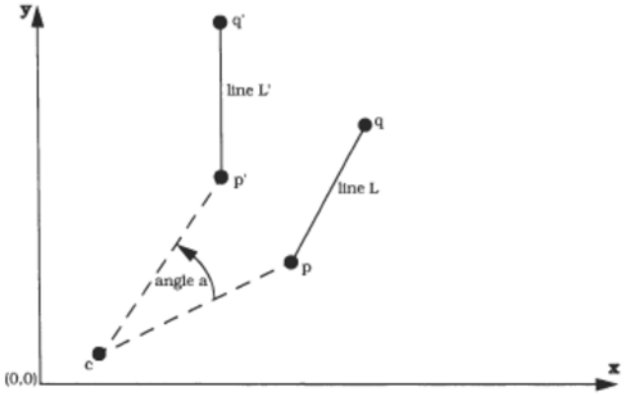
Dynamics:

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.

The rotation of a point **p** around a center c with an angle a can be calculated using the following formula:

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

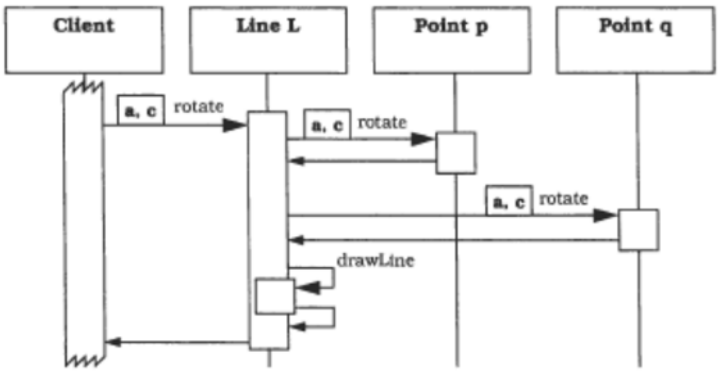
In the diagram below the rotation of the line given by the points p and q is illustrated.



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_1 and q_1 as endpoints.



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.

Variants:

- **Shared parts:**

  This variant relaxes the restriction that each Part must be associated with exactly one Whole by allowing several Wholes to share the same Part.
- **Assembly parts**

  In this variant the Whole may be an object that represents an assembly of smaller objects.
- **Container contents**

  In this variant a container is responsible for maintaining differing contents
- **Collection members**

  This variant is a specialization of Container-Contents, in that the Part objects all have the same type.
- **Composite pattern**

  It is applicable to Whole-Part hierarchies in which the Wholes and their Parts can be treated uniformly-that is, in which

both implement the same abstract interface.

Known uses:

- The key abstractions of many **object-oriented applications** follow the Whole-Part pattern.
- Most **object-oriented class libraries** provide collection classes such as lists, sets and maps. These classes implement the Collection- Member and Container-Contents variants.
- **Graphical user interface toolkits** such as Fresco or **ET++** use the Composite variant of the Whole-Part pattern.

Consequences:

The whole-part pattern offers several *Benefits:*

- **Changeability of parts:** Part implementations may even be completely exchanged without any need to modify other parts or clients.
- **Separation of concerns:** Each concern is implemented by a separate part.
- **Reusability in two aspects:**

a. Parts of a whole can be reused in other aggregate objects
b. Encapsulation of parts within a whole prevents clients from 'scattering' the use of part objects all over its source code.

The whole-part pattern suffers from the following *Liabilities:*

- **Lower efficiency through indirection**

Since the Whole builds a wrapper around its Parts, it introduces an additional level of indirection between a client request and the Part that fulfils it.

- **Complexity of decomposition into parts.**

An appropriate composition of a Whole from different Parts is often hard to find, especially when a bottom up approach is applied.