UNIT 5
ARCHITECTURAL PATTERNS-2

DISTRIBUTED SYSTEMS

What are the advantages of distributed systems that make them so interesting?
Distributed systems allow better sharing and utilization of the resources available within the network.

- Economics: computer network that incorporates both pc's and workstations offer a better price/performance ratio than mainframe computer.
- Performance and scalability: a huge increase in performance can be gained by using the combine computing power of several network nodes.
- Inherent distribution: some applications are inherently distributed. Ex: database applications that follow a client-server model.
- Reliability: A machine on a network in a multiprocessor system can crash without affecting the rest of the system.

Disadvantages
They need radically different software than do centralized systems.

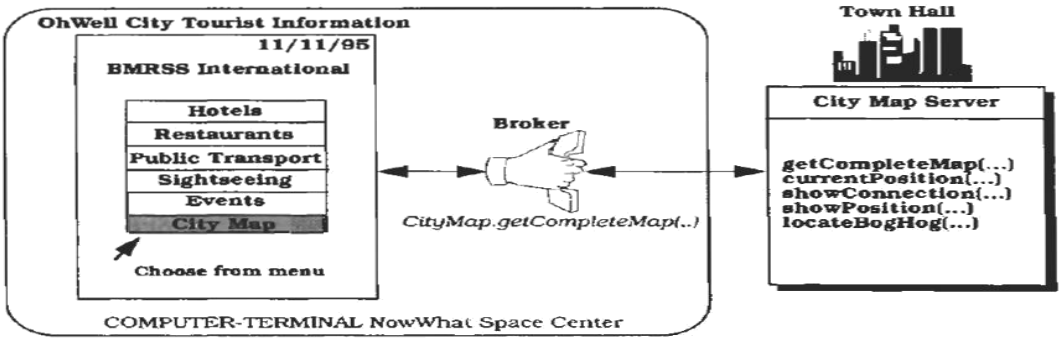
We introduce three patterns related to distributed systems in this category:

- The **Pipes and Filters pattern** provides a structure for systems that process a stream of data. Each processing step is encapsulated in a filter component. Data is passed through pipes between adjacent filters.
- The **Microkernel pattern** applies to software systems that must be able to adapt to changing system requirements. It separates a minimal functional core from extended functionality and customer-specific parts
- The **Broker pattern** can be used to structure distributed software systems with decoupled components that interact by remote service invocations.

BROKER

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.

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:**

- Implements objects that expose their functionality through interfaces that consists of operations and attributes.
- These interfaces are made available either through an interface definition language (IDL) or through a binary standard.
- There are two kind of servers:
  - ♠ Servers offering common services to many application domains.
  - ♠ Servers implementing specific functionality for a single application domain or task.

★ **Client:**

- Clients are applications that access the services of at least one server.
- To call remote services, clients forward requests to the broker. After an operation has executed they receive responses or exceptions from the broker.
- 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:**

- 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.
- It offers API'S to clients and servers that include operations for registering servers and for invoking server methods.
- 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.
- 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.
- Therefore there is a need for brokers to interoperate through bridges.

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

★ **Client side proxy:**

- They represent a layer b/w client and the broker.
- The proxies allow the hiding of implementation details from the clients such as
- The inter process communication mechanism used for message transfers b/w clients and brokers.
- The creation and deletion of blocks.
- The marshalling of parameters and results.

★ **Server side proxy:**

- Analogous to client side proxy. The difference that they are responsible for receiving requests, unpacking incoming messages, un marshalling the parameters, and calling the appropriate service.

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

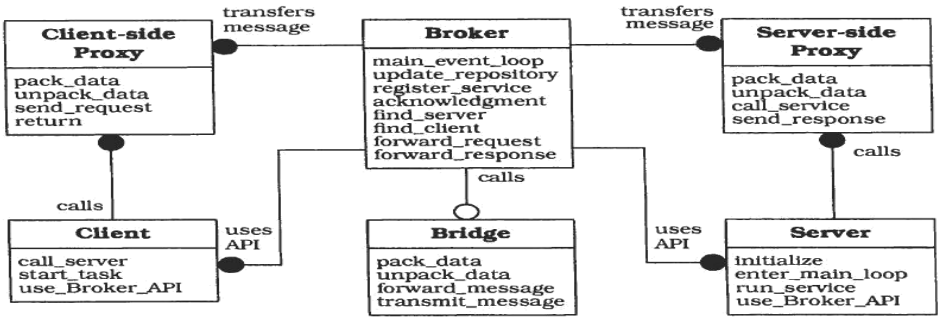
Class: Server-side Proxy	Collaborators
Responsibility • Calls services within the server. • Encapsulates system-specific functionality. • Mediates between the server and the broker.	• Server • Broker

★ **Bridges:**

- These are optional components used for hiding implementation details when two brokers interoperate.

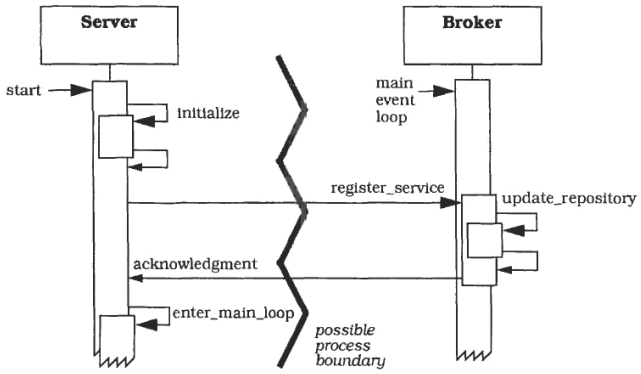
Class: Bridge	Collaborators
Responsibility • Encapsulates network-specific functionality. • Mediates between the local broker and the bridge of a remote broker.	• Broker • Bridge

The following diagram shows the objects involved in a broker system.

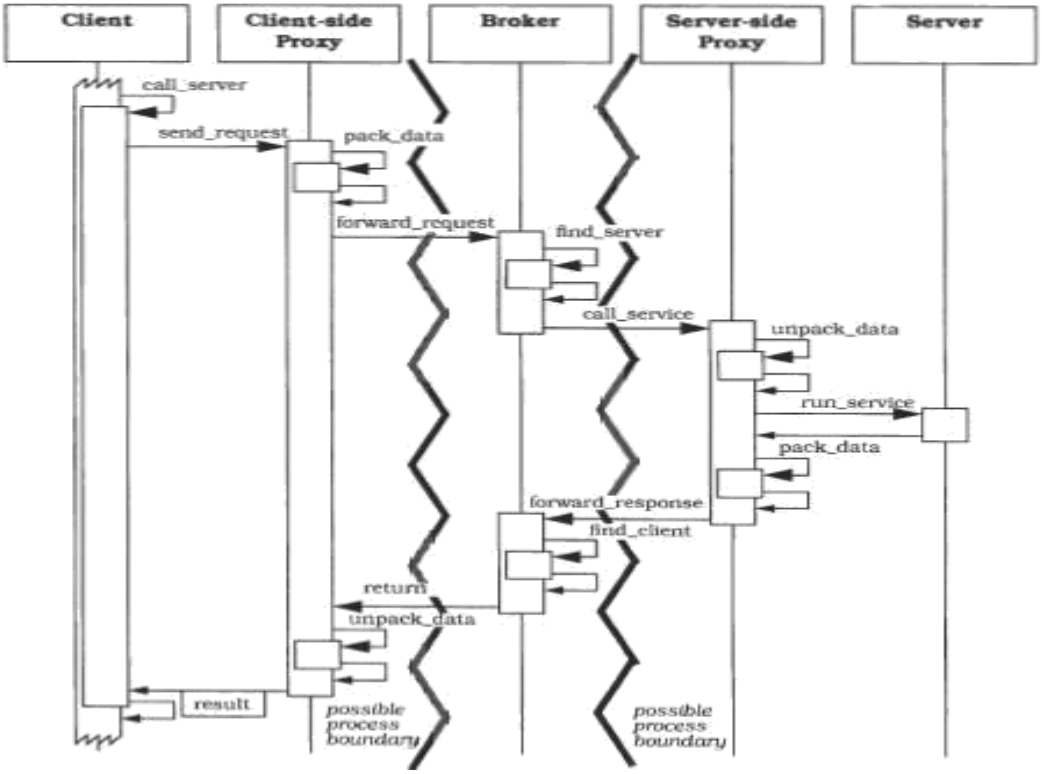


Dynamics:

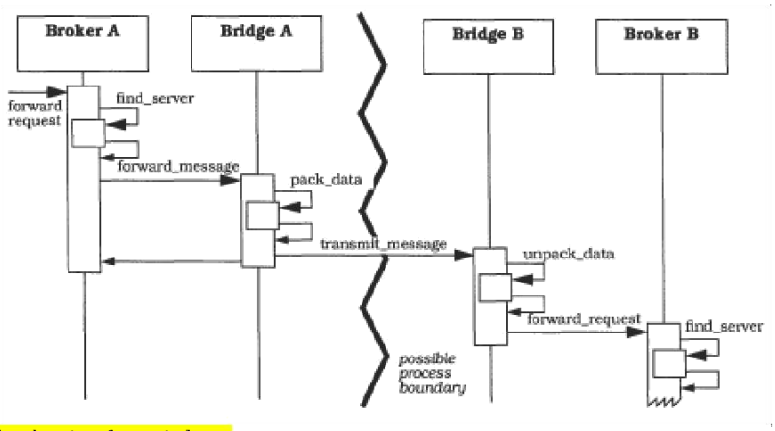
- **Scenario 1.** illustrates the behaviour when a server registers itself with the local broker component:



- **Scenario II** illustrates the behaviour 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.



- **Scenario III** illustrates the interaction of different brokers via bridge components:



[Please refer text book if you need detailed explanation of scenarios]

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.**
   - You can design for interoperability either by specifying a binary standard or by introducing a high-level IDL.
   - IDL file contains a textual description of the interfaces a server offers to its clients.
   - The binary approach needs support from your programming language.
3) **Specify the API'S the broker component provides for collaborating with clients and servers.**
   - 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.
   - This has a direct impact on size and no. of API'S.
4) **Use proxy objects to hide implementation details from clients and servers.**
   - Client side proxies package procedure calls into message and forward these messages to the local broker component.
   - 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.

Example resolved:

Our example CIS system offers different kinds of services. For example, a separate server workstation provides all the information related to public transport. Another server is responsible for collecting and publishing information on vacant hotel rooms. A tourist may be interested in retrieving information from several hotels, so we decide to provide this data on a single workstation. Every hotel can connect to the workstation and perform updates.

Variants:

- **Direct communication broker system:**
  - ★ We may sometime choose to relax the restriction that clients can only forward requests through the local brokers for efficiency reasons
  - ★ In this variant, clients can communicate with server directly.
  - ★ Broker tells the clients which communication channel the server provides.
  - ★ The client can then establish a direct link to the requested server
- **Message passing broker system:**
  - ✓ This variant is suitable for systems that focus on the transmission of data, instead of implementing a remote produce call abstraction.
  - ✓ 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.
  - ✓ 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 client request is usually forwarded to exactly one uniquely – identified servers.
  - ✓ In a trader system, the broker must know which server(s) can provide the service and forward the request to an appropriate server.
  - ✓ Here client side proxies use service identifiers instead of server identifiers to access server functionality.
  - ✓ The same request might be forwarded to more than one server implementing the same service.
- **Adapter broker system:**
  - ✓ Adapter layer is used to hide the interfaces of the broker component to the servers using additional layer to enhance flexibility
  - ✓ This adapter layer is a part of the broker and is responsible for registering servers and interacting with servers.
  - ✓ By supplying more than one adapter, support different strategies for server granularity and server location.
  - ✓ Example: use of an object oriented database for maintaining objects.
- **Callback broker system:**
  - ✓ Instead of implementing an active communication model in which clients produce requests and servers consume then and also use a reactive model.
  - ✓ It's a reactive model or event driven, and makes no distinction b/w clients and servers.
  - ✓ Whenever an event arrives, the broker invokes the call back method of the component that is registered to react to the event
  - ✓ 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

[Please refer text book if you need detailed explanation of these uses]

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.

INTERACTIVE SYSTEMS

These systems allow a high degree of user interaction, mainly achieved with the help of graphical user interfaces.

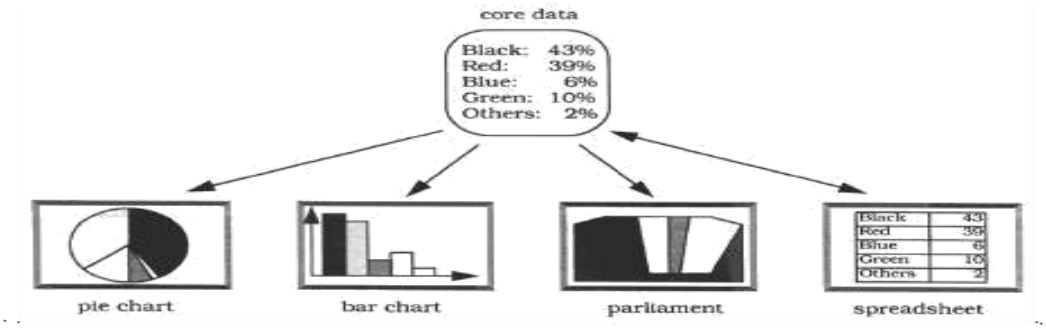
Two patterns that provide a fundamental structural organization for interactive software systems are:

- ➢ Model-view-controller pattern
- ➢ Presentation-abstraction-control pattern

MODEL-VIEW-CONTROLLER (MVC)

- *MVC architectural pattern divides an interactive application into three components.*
  - ✓ *The model contains the core functionality and data.*
  - ✓ *Views display information to the user.*
  - ✓ *Controllers handle user input.*
- *Views and controllers together comprise the user interface.*
- *A change propagation mechanism ensures consistence between the user interface and the model.*

Example:



Consider a simple information system for political elections with proportional representation. This offers a spreadsheet for entering data and several kinds of tables and charts for presenting the current results. Users can interact with the system via a graphical interface. All information displays must reflect changes to the voting data immediately.

Context:

Interactive applications with a flexible human-computer interface

Problem:

Different users place conflicting requirements on the user interface. A typist enters information into forms via the keyboard. A manager wants to use the same system mainly by clicking icons and buttons. Consequently, support for several user interface paradigms should be easily incorporated. How do you modularize the user interface functionality of a web application so that you can easily modify the individual parts?

The following forces influence the solution:

- ✓ Same information is presented differently in different windows. For ex: In a bar or pie chart.
- ✓ The display and behavior of the application must reflect data manipulations immediately.
- ✓ Changes to the user interface should be easy, and even possible at run-time.
- ✓ Supporting different 'look and feel' standards or porting the user interface should not affect code in the core of the application.

Solution:

- ★ MVC divides an interactive application into the three areas: processing, output and input.
- ★ Model component encapsulates core data and functionality and is independent of o/p and i/p.
- ★ View components display user information to user a view obtains the data from the model. There can be multiple views of the model.
- ★ Each view has an associated controller component controllers receive input (usually as mouse events) events are translated to service requests for the model or the view. The user interacts with the system solely through controllers.
- ★ The separation of the model from view and controller components allows multiple views of the same model.

Structure:

- ★ **Model component:**
  - Contains the functional core of the application.
  - Registers dependent views and controllers
  - Notifies dependent components about data changes (change propagation mechanism)

Class Model	Collaborators • View • Controller
Responsibility • Provides functional core of the application. • Registers dependent views and controllers. • Notifies dependent components about data changes.	

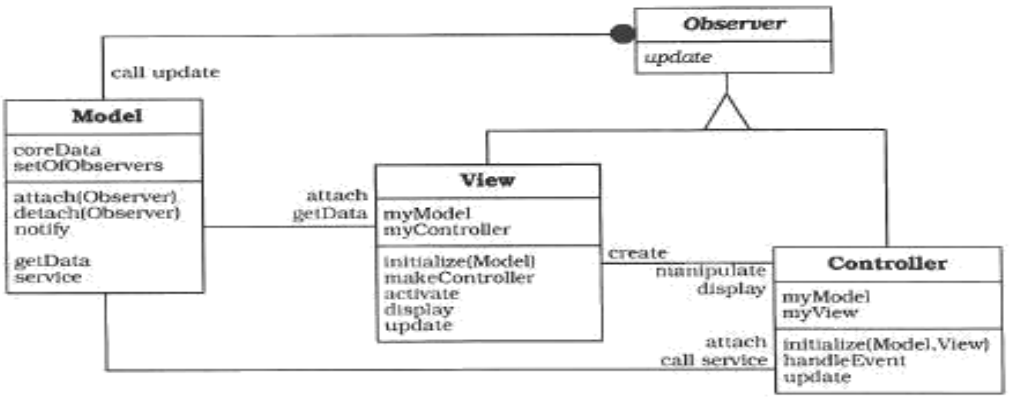
- ★ **View component:**
  - Presents information to the user
  - Retrieves data from the model
  - Creates and initializes its associated controller
  - Implements the update procedure
- ★ **Controller component:**
  - Accepts user input as events (mouse event, keyboard event etc)
  - Translates events to service requests for the model or display requests for the view.

- The controller registers itself with the change-propagation mechanism and implements an update procedure.

Class: View	Collaborators
Responsibility • Creates and initializes its associated controller. • Displays information to the user. • Implements the update procedure. • Retrieves data from the model.	• Controller • Model

Class: Controller	Collaborators
Responsibility • Accepts user input as events. • Translates events to service requests for the model or display requests for the view. • Implements the update procedure, if required.	• View • Model

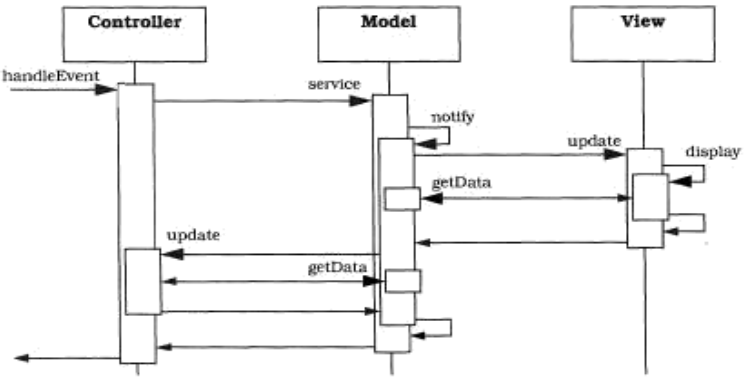
An object-oriented implementation of MVC would define a separate class for each component. In a C++ implementation, view and controller classes share a common parent that defines the update interface. This is shown in the following diagram.



Dynamics:

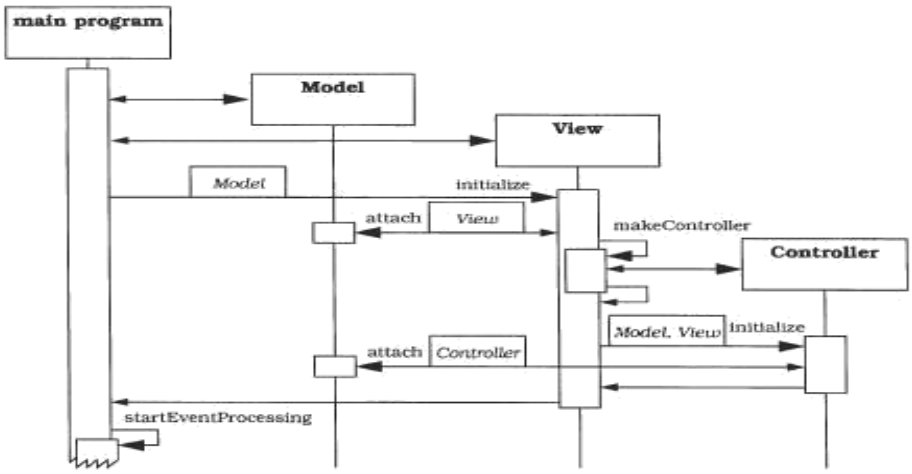
The following scenarios depict the dynamic behavior of MVC. For simplicity only one view-controller pair is shown in the diagrams.

- **Scenario** I shows how user input that results in changes to the model triggers the change-propagation mechanism:
  - The controller accepts user input in its event-handling procedure, interprets the event, and activates a service procedure of the model.
  - The model performs the requested service. This results in a change to its internal data.
  - The model notifies all views and controllers registered with the change-propagation mechanism of the change by calling their update procedures.
  - Each view requests the changed data from the model and redisplays itself on the screen.
  - Each registered controller retrieves data from the model to enable or disable certain user functions..
  - The original controller regains control and returns from its event handling procedure.



- **Scenario II** shows how the MVC triad is initialized. The following steps occur:
  - The model instance is created, which then initializes its internal data structures.
  - **A** view object is created. This takes a reference to the model as a parameter for its initialization.

- The view subscribes to the change-propagation mechanism of the model by calling the attach procedure.
- The view continues initialization by creating its controller. It passes references both to the model and to itself to the controller's initialization procedure.
- The controller also subscribes to the change-propagation mechanism by calling the attach procedure.
- After initialization, the application begins to process events.



Implementation:

1) Separate human-computer interaction from core functionality

- Analysis the application domain and separate core functionality from the desired input and output behavior

2) Implement the change-propagation mechanism

- Follow the publisher subscriber design pattern for this, and assign the role of the publisher to the model.

3) Design and implement the views

- design the appearance of each view
- Implement all the procedures associated with views.

4) Design and implement the controllers

- For each view of application, specify the behavior of the system in response to user actions.
- We assume that the underlying pattern delivers every action of and user as an event. A controller receives and interprets these events using a dedicated procedure.

5) Design and implement the view controller relationship.

- A view typically creates its associated controller during its initialization.

6) Implement the setup of MVC.

- The setup code first initializes the model, then creates and initializes the views.
- After initialization, event processing is started.
- Because the model should remain independent of specific views and controllers, this set up code should be placed externally.

7) Dynamic view creation

- If the application allows dynamic opening and closing of views, it is a good idea to provide a component for managing open views.

8) 'pluggable' controllers

- The separation of control aspects from views supports the combination of different controllers with a view.
- This flexibility can be used to implement different modes of operation.

9) Infrastructure for hierarchical views and controllers

- Apply the composite pattern to create hierarchically composed views.
- If multiple views are active simultaneously, several controllers may be interested in events at the same time.

10) Further decoupling from system dependencies.

- Building a framework with an elaborate collection of view and controller classes is expensive. You may want to make these classes platform independent. This is done in some Smalltalk systems

Variants:

Document View - This variant relaxes the separation of view and controller. In several GUI platforms, window display and event handling are closely interwoven. You can combine the responsibilities of the view and the controller from MVC in a single component by sacrificing exchangeability of controllers. This kind of structure is often called Document-View architecture. The view component of Document-View combines the responsibilities of controller and view in MVC, and implements the user interface of the system.

Known uses:

- **Smalltalk** - The best-known example of the use of the Model-View-Controller pattern is the user-interface framework in the Smalltalk environment
- **MFC** - The Document-View variant of the Model-View-Controller pattern is integrated in the Visual C++ environment-the Microsoft Foundation Class Library-for developing Windows applications.
- **ET++** - ET++ establishes 'look and feel' independence by defining a class **`Windowport`** that encapsulates the user interface platform dependencies.

Consequences:

Benefits:

- **Multiple views of the same model:**
  It is possible because MVC strictly separates the model from user interfaces components.
- **Synchronized views:**
  It is possible because of change-propagation mechanism of the model.
- **'pluggable views and controller:**
  It is possible because of conceptual separation of MVC.
- **Exchangeability of 'look and feel'**
  Because the model is independent of all user-interface code, a port of MVC application to a new platform does not affect the functional core of the application.
- **Framework potential**
  It is possible to base an application framework on this pattern.

Liabilities:

- **Increased complexity**
  Following the Model-View-Controller structure strictly is not always the best way to build an interactive application
- **Potential for excessive number of updates**
  If a single user action results in many updates, the model should skip unnecessary change notifications.
- **Intimate connection b/w view and controller**
  Controller and view are separate but closely-related components, which hinders their individual reuse.
- **Closed coupling of views and controllers to a model**
  Both view and controller components make direct calls to the model. This implies that changes to the model's interface are likely to break the code of both view and controller.
- **Inevitability of change to view and controller when porter**
  This is because all dependencies on the user-interface platform are encapsulated within view and controller.
- **Difficulty of using MVC with modern user interface tools**
  If portability is not an issue, using high-level toolkits or user interface builders can rule out the use of MVC.

PRESENTATION-ABSTRACTION-CONTROL

PAC defines a structure for interactive s/w systems in the form of a hierarchy of cooperating agents.

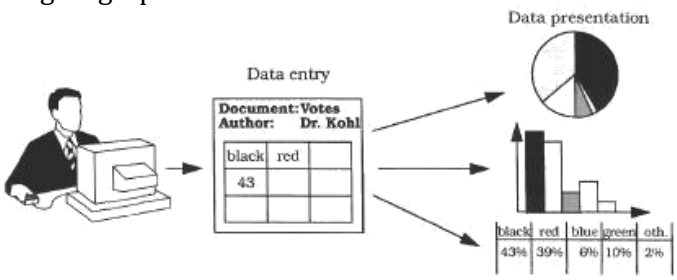
Every agent is responsible for a specific aspect of te applications functionality and consists of three components

- ✓ *Presentation*
- ✓ *Abstraction*
- ✓ *Control*

The subdivision separates the human-computer interaction aspects of the agent from its functional core and its communication with other agents.

Example:

Consider a simple information system for political elections with proportional representation. This offers a spreadsheet for entering data and several kinds of tables and charts for presenting current standings. Users interact with the software through a graphical interface.



Context:

Development of an interactive application with the help of agents

Problem:

Agents specialized in human-comp interaction accept user input and display data. Other agents maintain the data model of the system and offer functionality that operates on this data. Additional agents are responsible for error handling or communication with other software systems

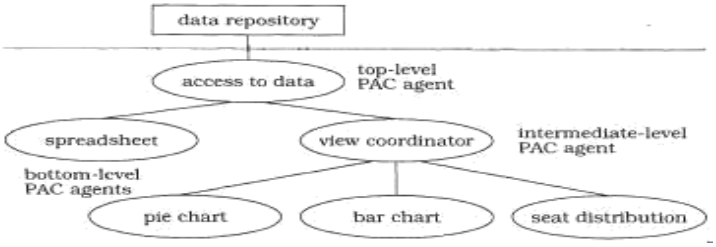
The following forces affect solution:

- ★ Agents often maintain their own state and data however, individual agents must effectively co operate to provide the overall task of the application. To achieve this they need a mechanism for exchanging data, messages and events.
- ★ Interactive agents provide their own user interface, since their respective human-comp interactions often differ widely
- ★ Systems evolve over time. Their presentation aspect is particularly prone to change. The use of graphics, and more recently, multimedia features are ex: of pervasive changes to user interfaces. Changes to individual agents, or the extension of the system with new agents, should not affect the whole system.

Solution:

- ♠ Structure the interactive application as a tree-like hierarchy f PAC agents every agent is responsible for a specific agent of the applications functionality and consists of three components:
  - ▶ Presentation
  - ▶ Abstraction
  - ▶ Control
- ♠ The agents presentation component provides the visible behavior of the PAC agent
- ♠ Its abstraction component maintains the data model that underlies the agent, and provides functionality that operates on this data.
- ♠ Its control component connects the presentation and abstraction components and provides the functionality that allow agent to communicate with other PAC agents.
- ♠ The top-level PAC agent provides the functional core of the system. Most other PAC agents depend or operate on its core.

- ♠ The bottom-level PAC agents represent self contained semantic concepts on which users of the system can act, such as spreadsheets and charts.
- ♠ The intermediate-level PAC agents represent either combination of, or relationship b/w. lower level agent.



Structure:

★ **Top level PAC agent:**

- Main responsibility is to provide the global data model of the software. This is maintained in the abstraction component of top level agent.
- The presentation component of the top level agent may include user-interface elements common to whole application.
- The control component has three responsibilities
  - Allows lower level agents to make use of the services of manipulate the global data model.
  - It co ordinates the hierarchy of PAC agents. It maintains information about connections b/w top level agent and lower-level agents.
  - It maintains information about the interaction of the user with system.

★ **Bottom level PAC agent:**

- Represents a specific semantic concept of the application domain, such as mail box in a n/w management system.
- The presentation component of bottom level PAC agents presents a specific view of the corresponding semantic concept, and provides access to all the functions users can apply to it.
- The abstraction component has a similar responsibility as that of top level PAC agent maintaining agent specific data.
- The control component maintains consistency b/w the abstraction and presentation components, by avoiding direct dependencies b/w them. It serves as an adapter and performs both interface and data adaption.

★ **Intermediate level PAC agent**

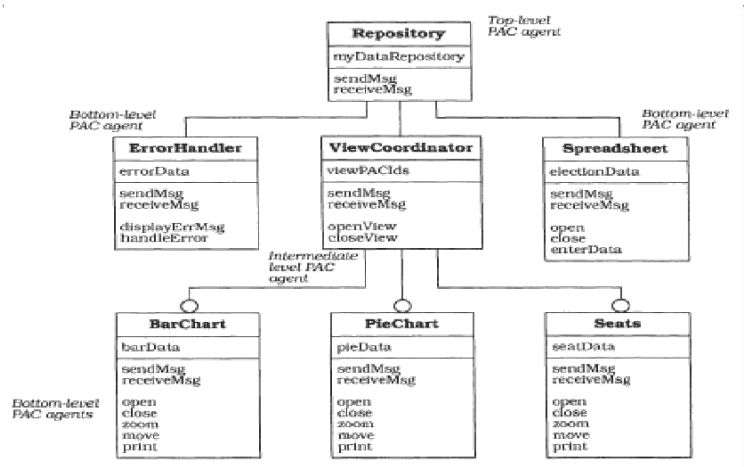
- Can fulfill two different roles: composition and co ordination.
- It defines a new abstraction, whose behavior encompasses both the behavior of its component, and the new characteristics that are added to the composite object.
- Abstraction component maintains the specific data of the intermediate level PAC agent.
- Presentation component implements its user interface.
- Control component has same responsibilities of those of bottom level and top level PAC agents.

Class: Top-level Agent	Collaborators
Responsibility • Provides the functional core of the system. • Controls the PAC hierarchy.	• Intermediate-level Agent • Bottom-level Agent

Class: Interm. -level Agent	Collaborators
Responsibility • Coordinates lower-level PAC agents. • Composes lower-level PAC agents to a single unit of higher abstraction.	• Top-level Agent • Intermediate-level Agent • Bottom-level Agent

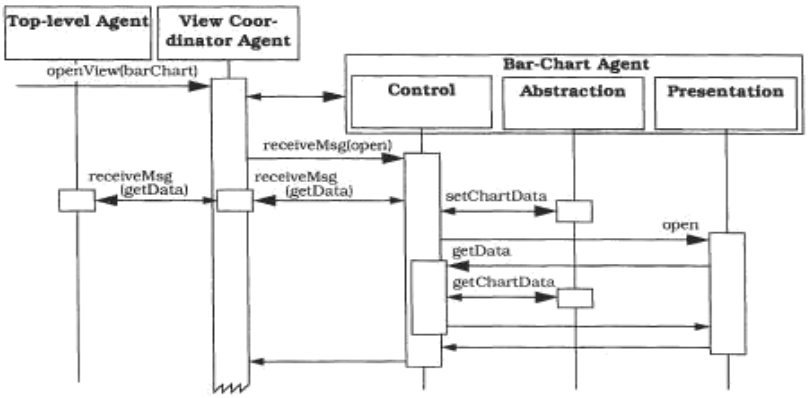
Class: Bottom-level Agent	Collaborators
Responsibility • Provides a specific view of the software or a system service, including its associated human-computer interaction.	• Top-level Agent • Intermediate-level Agent

The following OMT diagram illustrates the PAC hierarchy of the information system for political elections

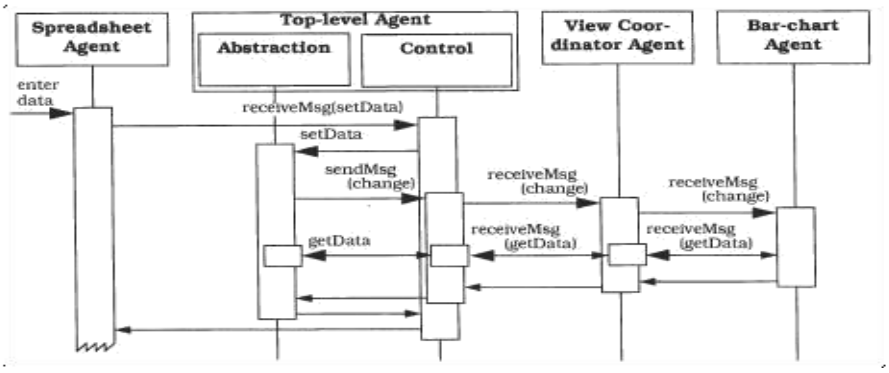


Dynamics:

- **Scenario I** describes the cooperation between different PAC agents when opening a new bar-chart view of the election data. It is divided into five phases:
  - A user asks the presentation component of the view coordinator agent to open a new bar chart.
  - The control of the view coordinator agent instantiates the desired bar-chart agent.
  - The view coordinator agent sends an 'open' event to the control component of the new bar-chart agent.
  - The control component of the bar-chart agent first retrieves data from the top-level PAC agent. The view coordinator agent mediates between bottom and top-level agents. The data returned to the bar-chart agent is saved in its abstraction component. Its control component then calls the presentation component to display the chart.
  - The presentation component creates a new window on the screen, retrieves data from the abstraction component by requesting it from the control component, and finally displays it within the new window.



- **Scenario II** shows the behavior of the system after new election data is entered, providing a closer look at the internal behavior of the toplevel PAC agent. It has five phases:
  - The user enters new data into a spreadsheet. The control component of the spreadsheet agent forwards this data to the toplevel PAC agent.
  - The control component of the top-level PAC agent receives the data and tells the top-level abstraction to change the data repository accordingly. The abstraction component of the top-level agent asks its control component to update all agents that depend on the new data. The control component of the top-level PAC agent therefore notifies the view coordinator agent.
  - The control component of the view coordinator agent forwards the change notification to all view PAC agents it is responsible for coordinating.
  - As in the previous scenario, all view PAC agents then update their data and refresh the image they display.



Implementation:

1) **Define a model of the application**
   Analyze the problem domain and map it onto an appropriate s/w structure.
2) **Define a general strategy for organizing the PAC hierarchy**
   Specify general guidelines for organizing the hierarchy of co operating agents.
   One rule to follow is that of "lowest common ancestor". When a group of lower level agents depends on the services or data provided by another agent, we try to specify this agent as the root of the sub tree formed by the lower level agents. As a consequence, only agents that provide global services rise to the top of the hierarchy.
3) **Specify the top-level PAC agent**
   Identify those parts of the analysis model that represent the functional core of the system.
4) **Specify the bottom-level PAC agent**
   Identify those components of the analysis model that represent the smallest self-contained units of the system on which the user can perform operations or view presentations.
5) **Specify bottom-level PAC agents for system service**
   Often an application includes additional services that are not directly related to its primary subject. In our example system we define an error handler.
6) **Specify intermediate level PAC agents to compose lower level PAC agents**
   Often, several lower-level agents together form a higher-level semantic concept on which users can operate.
7) **Specify intermediate level PAC agents to co ordinate lower level PAC agents**
   Many systems offer multiple views of the same semantic concept. For example, in text editors you find 'layout' and 'edit" views of a text document. When the data in one view changes, all other views must be updated.
8) **Separate core functionality from human-comp interaction.**
   For every PAC agent, introduce presentation and abstraction component.
9) **Provide the external interface to operate with other agents**
   Implement this as part of control component.
10) **Link the hierarchy together**
    Connect every PAC agent with those lower level PAC agents with which it directly co operates

Variants:

★ **PAC agents as active objects** - Every PAC agent can be implemented as an active object that lives in its own thread of control.

★ **PAC agents as processes** - To support PAC agents located in different processes or on remote machines, use proxies to locally represent these PAC agents and to avoid direct dependencies on their physical location.

Known uses:

- **Network traffic management**
  - Gathering traffic data from switching units.
  - Threshold checking and generation of overflow exceptions.

- Logging and routing of network exceptions.
- Visualization of traffic flow and network exceptions.
- Displaying various user-configurable views of the whole network.
- Statistical evaluations of traffic data.
- Access to historic traffic data.
- System administration and configuration.

- **Mobile robot**
  - Provide the robot with a description of the environment it will work in, places in this environment, and routes between places.
  - Subsequently modify the environment.
  - Specify missions for the robot.
  - Control the execution of missions.
  - Observe the progress of missions.

Consequences:

Benefits:-

- **separation of concerns**
  - Different semantic concepts in the application domain are represented by separate agents.
- **Support for change and extension**
  - Changes within the presentation or abstraction components of a PAC agent do not affect other agents in the system.
- **Support for multi tasking**
  - PAC agents can be distributed easily to different threads, processes or machines.
  - Multi tasking also facilitates multi user applications.

Liabilities:

- ✓ **Increased system complexity**

  Implementation of every semantic concept within an application as its own PAC agent may result in a complex system structure.
- ✓ **Complex control component**
  - Individual roles of control components should be strongly separated from each other. The implementations of these roles should not depend on specific details of other agents.
  - The interface of control components should be independent of internal details.
  - It is the responsibility of the control component to perform any necessary interface and data adaptation.
- ✓ **Efficiency:**
  - Overhead in communication between PAC agents may impact system efficiency.
  - Example: All intermediate-level agents are involved in data exchange. if a bottom-level agent retrieve data from top-level agent.
- ✓ **Applicability:**
  - The smaller the atomic semantic concepts of an applications are, and the greater the similarly of their user interfaces, the less applicable this pattern is.

UNIT 5 – QUESTION BANK

No.	QUESTION	YEAR	MARKS
1	Explain the variants of broker architecture	Dec 09	10
2	Depict the dynamic behavior of MVC, with any one scenario	Dec 09	5
3	Give the CRC cards for top level, intermediate level and bottom level PAC-agents	Dec 09	5
4	What do you mean by broker architecture? What are the steps involved in implementing distributed broker architecture patterns?	June 10	10
5	Explain with a neat diagram, the dynamic scenarios of MVC	June 10	10
6	What is the necessity of proxies and bridge components in a broker system? Explain	Dec 10	6
7	Explain the possible dynamic behavior of MVC pattern, with suitable sketches	Dec 10	9
8	Highlight the limitations of PAC pattern	Dec 10	5
9	Give detailed explanation on the different steps involved in the implementation of the broker pattern	June 11	15
10	Propose the description of a scenario that depicts the dynamic behavior of MVC in detail. Support the description with appropriate pictorial representation	June 11	5
11	Discuss the most relevant scenario, illustrating the dynamic behavior of a broker system	Dec 11	10
12	Discuss the consequences of PAC architectural pattern	Dec 11	10
13	Define broker architectural pattern. Explain with a diagram the objects involved in a broker system	June 12	7
14	Depict the dynamic behavior of MVC, with any one scenario	June 12	7
15	Give the CRC cards for top level, intermediate level and bottom level PAC-agents	June 12	6

UNIT 6
ARCHITECTURAL PATTERNS-3

ADAPTABLE SYSTEMS

The systems that evolve over time - new functionality is added and existing services are changed are called adaptive systems.
They must support new versions of OS, user-interface platforms or third-party components and libraries. Design for change is a major concern when specifying the architecture of a software system.
We discuss two patterns that helps when designing for change.

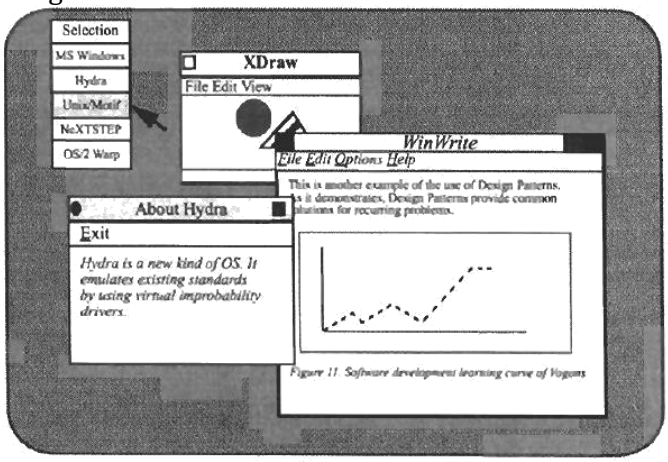
- ***Microkernel pattern*** → applies to software systems that must be able to adapt to changing system requirements.
- ***Reflection pattern*** → provides a mechanism for changing structure and behavior of software systems dynamically.

MICROKERNEL

The microkernel architectural pattern applies to software systems that must be able to adapt to changing system requirements. It separates a minimal functional core from extended functionality and customer-specific parts the microkernel also serves as a socket for plugging in these extensions and coordinating their collaboration.

Example:

Suppose we intend to develop a new operating system for desktop computers called Hydra. One requirement is that this innovative operating system must be easily portable to the relevant hardware platforms, and must be able to accommodate future developments easily. It must also be able to run applications written for other popular operating systems such as Microsoft Windows and UNIX System V. A user should be able to choose which operating system he wants from a pop-up menu before starting an application. Hydra will display `all` the applications currently running within its main window:



Context:

The development of several applications that use similar programming interfaces that build on the same care functionality.

Problem:

Developing software for an application domain that needs to cope with a broad spectrum of similar standards and technology is a nontrivial task well known. Ex: are application platform such as OS and GUI'S.
The following forces are to be considered when designing such systems.

- ♣ The application platform must cope with continuous hardware and software evolution.

- ♣ The application platform should be portable, extensible and adaptable to allow easy integration of emerging technologies.

Application platform such as an OS should also be able to emulate other application platforms that belong to the same application domain. This leads to following forces.

- ♠ The applications in your domain need to support different but, similar application platforms.
- ♠ The applications may be categorized into groups that use the same functional core in different ways, requiring the underlying application platform to emulate existing standards.

Additional force must be taken to avoid performance problem and to guarantee scalability.

- The functional core of the application platform should be separated into a component with minimal memory size, and services that consume as little processing power as possible.

Solution:

- Encapsulates the fundamental services of your applications platform in a microkernel component.
- It includes functionality that enables other components running in different process to communicate with each other.
- It provides interfaces that enable other components to access its functionality.
  - Core functionality should be included in ***internal servers.***
  - ***External servers*** implement their own view of the underlying microkernel.
  - ***Clients*** communicate with external servers by using the communication facilities provided by microkernel.

Structure:

Microkernel pattern defines 5 kinds of participating components.

- ♠ Internal servers
- ♠ External servers
- ♠ Adapters
- ♠ Clients
- ♠ Microkernel

❖ **Microkernel**

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

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

❖ **An internal server (subsystem)**

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

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

❖ **An external server (personality)**

- Uses the microkernel for implementing its own view of the underlying application domain.
- Each external server runs in separate process.
- 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.
- Different external servers implement different policies for specific application domains.

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

❖ **Client:**

- It is an application that is associated with exactly one external server. It only accesses the programming interfaces provided by the external server.
- 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.

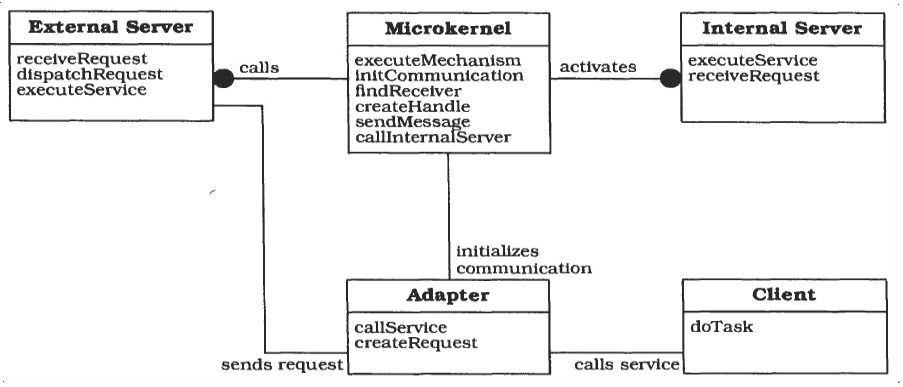
❖ **Adapter** (emulator)

- 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.
- They are part of the clients address space.
- The following OMT diagram shows the static structure of a microkernel system.

Class Client	Collaborators • Adapter
Responsibility • Represents an application.	

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

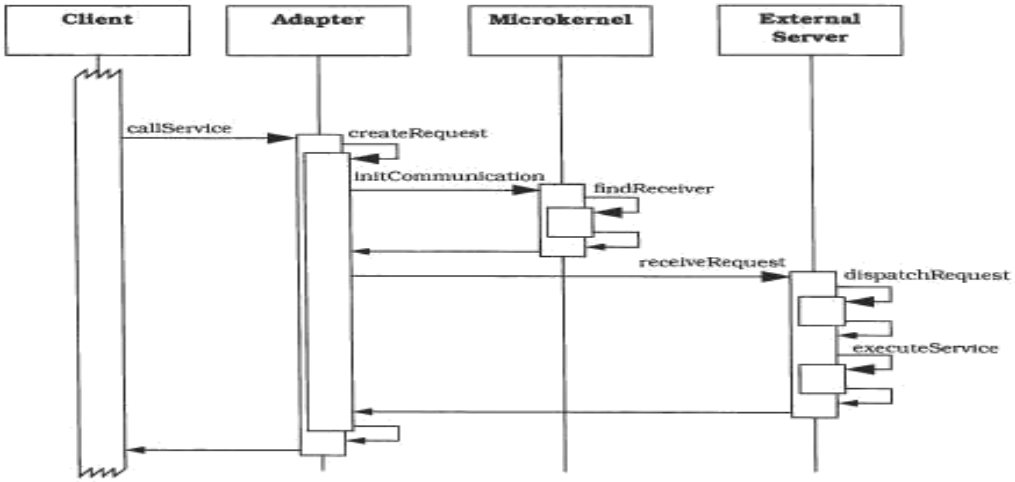
The following OMT diagram shows the static structure of a Microkernel system.



Dynamics:

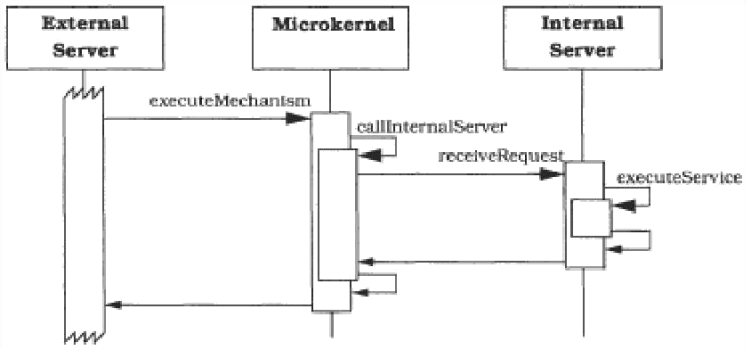
Scenario I demonstrates the behavior when a client calls a service of its external server:

- At a certain point in its control flow the client requests a service from an external server by calling the adapter.
- The adapter constructs a request and asks the microkernel for a communication link with the external server.
- The microkernel determines the physical address of the external server and returns it to the adapter.
- After retrieving this information, the adapter establishes a direct communication link to the external server.
- The adapter sends the request to the external server using a remote procedure call.
- The external server receives the request, unpacks the message and delegates the task to one of its own methods. After completing the requested service, the external server sends all results and status information back to the adapter.
- The adapter returns to the client, which in turn continues with its control flow.



Scenario II illustrates the behavior of a Microkernel architecture when an external server requests a service that is provided by an internal server

- The external server sends a service request to the microkernel.
- A procedure of the programming interface of the microkernel is called to handle the service request. During method execution the microkernel sends a request to an internal server.
- After receiving the request, the internal server executes the requested service and sends all results back to the microkernel.
- The microkernel returns the results back to the external server.
- Finally, the external server retrieves the results and continues with its control flow.



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.

13. **Develop client applications**
or use existing ones for the ready-to-run microkernel system.

Example resolved:

Shortly after the development of Hydra has been completed, we are asked to integrate an external server that emulates the Apple MacOS operating system. To provide a MacOS emulation on top of Hydra, the following activities are necessary:

- ✓ *Building an external server* on top of the Hydra microkernel that implements all the programming interfaces provided by MacOS, including the policies of the Macintosh user interface.
- ✓ *Providing an adapter* that is designed as a library, dynamically linked to clients.
- ✓ *Implementing the internal servers* required for MacOS.

Variants:

- *Microkernel system with indirect client-server communications.*
  In this variant, a client that wants to send a request or message to an external server asks the microkernel for a communication channel.
- *Distributed microkernel systems.*
  In this variant a microkernel can also act as a message backbone responsible for sending messages to remote machines or receiving messages from them.

Known uses:

- The **Mach** microkernel is intended to form a base on which other operating systems can be emulated. One of the commercially available operating systems that use Mach as its system kernel is NeXTSTEP.
- The operating system **Amoeba** consists of two basic elements: the microkernel itself and a collection of servers (subsystems) that are used to implement the majority of Amoeba's functionality. The kernel provides four basic services: the management of processes and threads, the low-level-management of system memory, communication services, both for point-to-point communication as well as group-communication, and low-level I/O services.
- **Chorus** is a commercially-available Microkernel system that was originally developed specifically for real-time applications.
- **Windows NT** was developed by Microsoft as an operating system for high-performance servers. **MKDE**
- (Microkernel Datenbank Engine) system introduces an architecture for database engines that follows the Microkernel pattern.

Consequences:

The microkernel pattern offers some important *Benefits:*

- **Portability:**
  High degree of portability
- **Flexibility and extensibility:**
  - ✓ If you need to implement an additional view, all you need to do is add a new external server.
  - ✓ Extending the system with additional capabilities only requires the additional or extension of internal servers.
- **Separation of policy and mechanism**
  The microkernel component provides all the mechanisms necessary to enable external servers to implement their policies.

Distributed microkernel has further benefits:

- ♠ **Scalability**
  A distributed Microkernel system is applicable to the development of operating systems or database systems for computer networks, or multiprocessors with local memory
- ♠ **Reliability**
  A distributed Microkernel architecture supports availability, because it allows you to run the same server on more than one machine, increasing availability. Fault tolerance may be easily supported because distributed systems allow you to hide failures from a user.

- **Transparency**
  In a distributed system components can be distributed over a network of machines. In such a configuration, the Microkernel architecture allows each component to access other components without needing to know their location.

The microkernel pattern also has *Liabilities:*

- **Performance:**
  Lesser than monolithic software system therefore we have to pay a price for flexibility and extensibility.
- **Complexity of design and implementation:**
  Develop a microkernel is a non-trivial task.

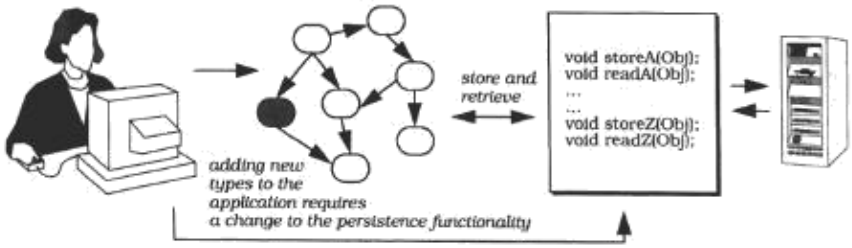
REFLECTION

The reflection architectural pattern provides a mechanism for changing structure and behavior of software systems dynamically. It supports the modification of fundamental aspects, such as the type structures and function call mechanisms. In this pattern, an application is split into two parts:

- *A Meta level provides information about selected system properties and makes the s/w self aware.*
- *A base level includes application logic changes to information kept in the Meta level affect subsequent base-level behavior.*

Example:

Consider a C++ application that needs to write objects to disk and read them in again. Many solutions to this problem, such as implementing type-specific store and read methods, are expensive and error-prone. Persistence and application functionality are strongly interwoven. Instead we want to develop a persistence component that is independent of specific type structures



Context:

Building systems that support their own modification a prior

Problem:

- Designing a system that meets a wide range of different requirements a prior can be an overwhelming task.
- A better solution is to specify an architecture that is open to modification and extension i.e., we have to design for change and evolution.
- Several forces are associated with the problem:
  - ✓ Changing software is tedious, error prone and often expensive.
  - ✓ Adaptable software systems usually have a complex inner structure. Aspects that are subject to change are encapsulated within separate components.
  - ✓ The more techniques that are necessary for keep in a system changeable the more awkward and complex its modifications becomes.
  - ✓ Changes can be of any scale, from providing shortcuts for commonly used commands to adapting an application framework for a specific customer.
  - ✓ Even fundamental aspects of software systems can change for ex. communication mechanisms b/w components.

Solution:

★ Make the software self-aware, and make select aspects of its structure and behavior accessible for adaptation and change.
  - This leads to an architecture that is split into two major parts: A Meta level
  - A base level

★ Meta level provides a self representation of the s/w to give it knowledge of its own structure and behavior and consists of so called *Meta objects* (they encapsulate and represent information about the software). Ex: type structures algorithms or function call mechanisms.

★ Base level defines the application logic. Its implementation uses the Meta objects to remain independent of those aspects that are likely to change.

★ An interface is specified for manipulating the Meta objects. It is called the *Meta object protocol* (MOP) and allows clients to specify particular changes.

Structure:

- ♥ Meta level
- ♥ Meta objects protocol(MOP)
- ♥ Base level

❖ **Meta level**

✓ Meta level consists of a set of Meta objects. Each Meta object encapsulates selected information about a single aspect of a structure, behavior, or state of the base level.
There are three sources for such information.
- It can be provided by run time environment of the system, such as C++ type identification objects.
- It can be user defined such as function call mechanism
- It can be retrieved from the base level at run time.

✓ All Meta objects together provide a self representation of an application.

✓ What you represent with Meta objects depends on what should be adaptable only system details that are likely to change r which vary from customer to customer should be encapsulated by Meta objects.

✓ The interface of a Meta objects allows the base level to access the information it maintains or the services it offers.

❖ **Base level**

✓ It models and implements the application logic of the software. Its component represents the various services the system offers as well as their underlying data model.

✓ It uses the info and services provided by the Meta objects such as location information about components and function call mechanisms. This allows the base level to remain flexible.

✓ Base level components are either directly connected to the Meta objects and which they depend or submit requests to them through special retrieval functions.

Class Base Level	***Collaborators***
Responsibility • Implements the application logic. • Uses information provided by the meta level.	• Meta Level

Class Meta Level	***Collaborators***
Responsibility • Encapsulates system internals that may change. • Provides an interface to facilitate modifications to the meta-level.	• Base Level

❖ **Meta object protocol (MOP)**

✓ Serves an external interface to the Meta level, and makes the implementation of a reflective system accessible in a defined way.

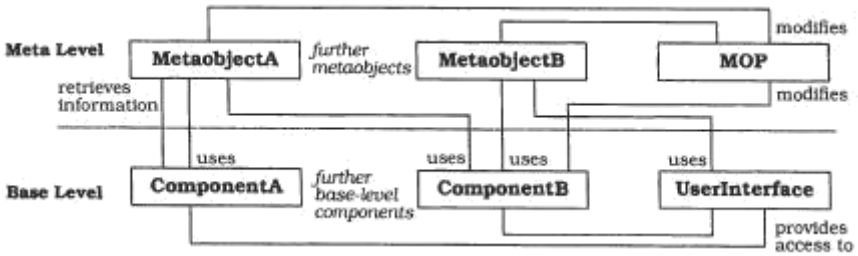
✓ Clients of the MOP can specify modifications to Meta objects or their relationships using the base level

- ✓ MOP itself is responsible for performing these changes. This provides a reflective application with explicit control over its own modification.
- ✓ Meta object protocol is usually designed as a separate component. This supports the implementation of functions that operate on several Meta objects.
- ✓ To perform changes, the MOP needs access to the internals of Meta objects, and also to base level components (sometimes).

  One way of providing this access is to allow the MOP to directly operate on their internal states. Another way (safer, inefficient) is to provide special interface for their manipulation, only accessible by MOP.

Class Metaobject Protocol	Collaborators • Meta Level • Base Level
Responsibility • Offers an interface for specifying changes to the meta level. • Performs specified changes	

The general structure of a reflective architecture is very much like a Layered system



Dynamics:

Interested students can refer text book for scenarios since the diagrams are too hectic & can't be memorized

Implementation:

Iterate through any subsequence if necessary.

1. Define a model of the application

Analyze the problem domain and decompose it into an appropriate s/w structure.

2. Identify varying behavior

- ✓ Analyze the developed model and determine which of the application services may vary and which remain stable.
- ✓ Following are ex: of system aspects that often vary ○ Real time constraints
  - ○ Transaction protocols
  - ○ Inter Process Communication mechanism
  - ○ Behavior in case of exceptions
  - ○ Algorithm for application services.

3. **Identify structural aspects** of the system, which when changed, should not affect the implementation of the base level.

4. Identify system services that support both the variation of application services identified

In step 2 and the independence of structural details identified in step 3

Eg: for system services are

- ✓ Resource allocation
- ✓ Garbage allocation
- ✓ Page swapping
- ✓ Object creation.

5. Define the meta objects

- ✓ For every aspect identified in 3 previous steps, define appropriate Meta objects.

✓ Encapsulating behavior is supported by several domain independent design patterns, such as objectifier strategy, bridge, visitor and abstract factory.

6. Define the MOP

✓ There are two options for implementing the MOP.

- Integrate it with Meta objects. Every Meta object provides those functions of the MOP that operate on it.
- Implement the MOP as a separate component.

✓ Robustness is a major concern when implementing the MOP. Errors in change specifications should be detected wherever possible.

7. Define the base level

✓ Implement the functional core and user interface of the system according to the analysis model developed in step 1.

✓ Use Meta objects to keep the base level extensible and adaptable. Connect every base level component with Meta objects that provide system information on which they depend, such as type information etc.

✓ Provide base level components with functions for maintaining the relationships with their associated Meta objects. The MOP must be able to modify every relationship b/w base level and Meta level.

Example resolved:

Unlike languages like CLOS or Smalltalk. C++ does not support reflection very well-only the standard class type_info provides reflective capabilities: we can identify and compare types. One solution for providing extended type information is to include a special step in the compilation process. In this, we collect type information from the source files of the application, generate code for instantiating the' metaobjects, and link this code with the application. Similarly, the 'object creator' metaobject is generated. Users specify code for instantiating an 'empty' object of every type, and the toolkit generates the code for the metaobject. Some parts of the system are compiler-dependent, such as offset and size calculation.

Variants:

♥ *Reflection with several Meta levels*

Sometimes metaobjects depend on each other. Such a software system has an infinite number of meta levels in which each meta level is controlled by a higher one, and where each meta level has its own metaobject protocol. In practice, most existing reflective software comprises only one or two meta levels.

Known uses:

❖ **CLOS.**

This is the classic example of a reflective programming language [Kee89]. In CLOS, operations defined for objects are called generic functions, and their processing is referred to as generic function invocation. Generic function invocation is divided into three phases:

- ♣ The system first determines the methods that are applicable to a given invocation.
- ♣ It then sorts the applicable methods in decreasing order of precedence.
- ♣ The system finally sequences the execution of the list of applicable methods.

❖ **MIP**

It is a run-time type information system for C++. The functionality of MIP is separated into four layers:

- ♣ The first layer includes information and functionality that allows software to identify and compare types.
- ♣ The second layer provides more detailed information about the type system of an application.
- ♣ The third layer provides information about relative addresses of data members, and offers functions for creating 'empty' objects of user-defined types.
- ♣ The fourth layer provides full type information, such as that about friends of a class, protection of data members, or argument and return types of function members.

❖ **PGen**

It allows an application to store and read arbitrary C++ object structures.

- **NEDIS**

  NEDIS includes a meta level called run-time data dictionary. It provides the following services and system information:
  - Properties for certain attributes of classes, such as their allowed value ranges.
  - Functions for checking attribute values against their required properties.
  - Default values for attributes of classes, used to initialize new objects.
  - Functions specifying the behavior of the system in the event of errors
  - Country-specific functionality, for example for tax calculation.
  - Information about the 'look and feel' of the software, such as the layout of input masks or the language to be used in the user interface.
- **OLE 2.0**

  It provides functionality for exposing and accessing type information about OLE objects and their interfaces.

Consequences:

The reflection architecture provides the following *Benefits:*

- **No explicit modification of source code:**

  You just specify a change by calling function of the MOP.
- **Changing a software system is easy**

  MOP provides a safe and uniform mechanism for changing s/w. it hides all specific techniques such as use of visitors, factories from user.
- **Support for many kind of change:**

  Because Meta objects can encapsulate every aspect of system behavior, state and structure.

The reflection architecture also has *Liabilities:*

- **Modifications at meta level may cause damage:**
  - ✓ Incorrect modifications from users cause serious damage to the s/w or its environment. Ex: changing a database schema without suspending the execution of objects in the applications that use it or passing code to the MOP that includes semantic errors
  - ✓ Robustness of MOP is therefore of great importance.
- **Increased number of components:**

  It includes more Meta objects than base level components.
- **Lower efficiency:**

  Slower than non reflective systems because of complex relnp b/w base and meta level.
- **Not all possible changes to the software are supported**

  Ex: changes or extensions to base level code.
- **Not all languages support reflection**

  Difficult to implement in C ++

UNIT 6 – QUESTION BANK

No.	QUESTION	YEAR	MARKS
1	Explain the benefits and liabilities of microkernel pattern	Dec 09	10
2	Enumerate the implementation steps of reflection pattern	Dec 09	10
3	What are the steps involved in implementing the microkernel system?	June 10	12
4	What are the benefits and liabilities of reflection architecture pattern?	June 10	8
5	List and explain the participating components of a microkernel pattern	Dec 10	10
6	Explain the known uses of reflection pattern	Dec 10	10
7	Discuss the benefits and liabilities of microkernel pattern	June 11	10
8	Give the detailed explanation on the different known applications offered by the reflection pattern	June 11	10
9	Explain in brief, the components comprising the structure of microkernel architectural pattern	Dec 11	10
10	With an example, explain when the reflection architectural pattern is used. What are its benefits?	Dec 11	10
11	What are the steps involved in implementing the microkernel system?	June 12	8
12	Explain the known uses of reflection pattern	June 12	6
13	Explain the advantages and disadvantages of reflection architectural pattern	June 12	6

UNIT 7
SOME DESIGN PATTERNS

INTRODUCTION:

Design patterns are medium scale patterns. They are smaller in scale than architectural patterns, but are at a higher level than the programming language specific idioms.
We group design patterns into categories of related patterns, in the same way as we did for architectural patterns:

- **Structural Decomposition**
  This category includes patterns that support a suitable decomposition of subsystems and complex components into co-operating parts. The Whole-Part pattern is the most general pattern we are aware of in this category.
- **Organization of Work.**
  This category comprises patterns that define how components collaborate together to solve a complex problem. We describe the Master-Slave pattern, which helps you to organize the computation of services for which fault tolerance or computational accuracy is required.
- **Access Control.**
  Such patterns guard and control access to services or components. We describe the Proxy pattern here.
- **Management.**
  This category includes patterns for handling homogenous collections of objects, services and components in their entirety. We describe two patterns: the Command Processor pattern addresses the management and scheduling of user commands, while the View Handler pattern describes how to manage views in a software system.
- **Communication.**
  Patterns in this category help to organize communication between components. Two patterns address issues of inter-process communication: the Forwarder-Receiver pattern deals with peer-to-peer communication, while the Client Dispatcher-Server pattern describes location-transparent communication in a Client-Server structure.

STRUCTURAL DECOMPOSITION:

Subsystems and complex components are handled more easily if structured into smaller independent components, rather than remaining as monolithic block of code.
We discuss whole-part design pattern that supports the structural decomposition of the component.

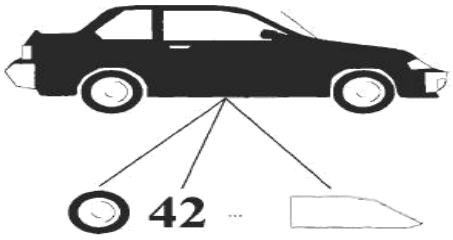
WHOLE-PART

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

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 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.
  - ✓ 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
  - ✓ An *assembly-parts* relationship which differentiation b/w a product and its parts or subassemblies.
  - ✓ A *container-contents* relationship, in which the aggregated object represents a container.
  - ✓ The *collection-members* relationship, which helps to group similar objects.

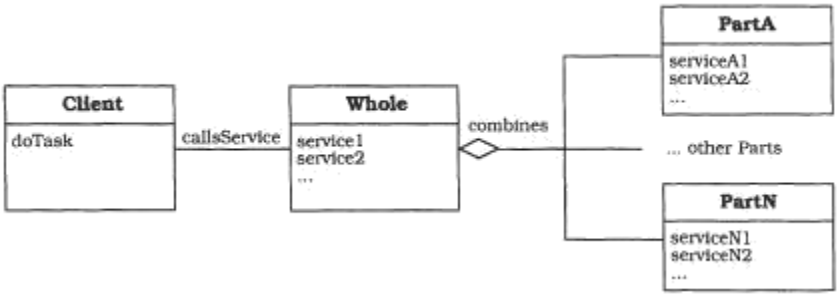
Structure:
The Whole-Part pattern introduces two types of participant:

- **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.

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

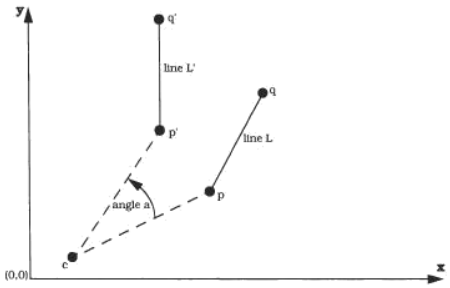


Dynamics:

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:

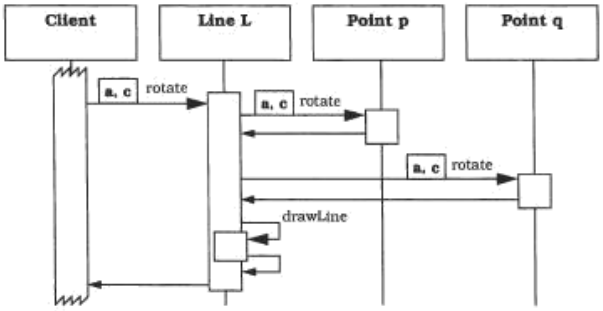
$$\vec{p}' = \begin{bmatrix} \cos a & -\sin a \\ \sin a & \cos a \end{bmatrix} \cdot (\vec{p} - \vec{c}) + \vec{c}$$

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:

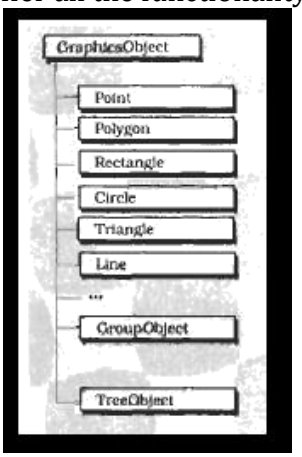
1. **Design the public interface of the whole**
   - Analyze the functionality the whole must offer to its clients.
   - Only consider the clients view point in this step.
   - Think of the whole as an atomic component that is not structured into parts.
2. **Separate the whole into parts, or synthesize 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.

Example resolved:

In our CAD system we decide to define a Java package that provides the basic functionality for graphical objects. The class library consists of atomic objects such as circles or lines that the user can combine to form more complex entities. We implement these classes directly instead of using the standard Java package awt (Abstract Windowing Toolkit) because awt does not offer all the functionality we need.



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:**
  - Parts of a whole can be reused in other aggregate objects
  - 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.

ORGANIZATION OF WORK

The implementation of complex services is often solved by several components in co operation. To organize work optimally within such structures you need to consider several aspects.

We describe one pattern for organizing work within a system → maser-slave pattern.

MASTER-SLAVE

*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.*

Example:

Travelling salesman problem

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:

- 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:**

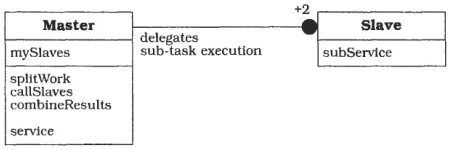
✓ Provides a sub-service that can process the subtasks defined by the master

✓ There are at least two instances of the slave component connected to the master.

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

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

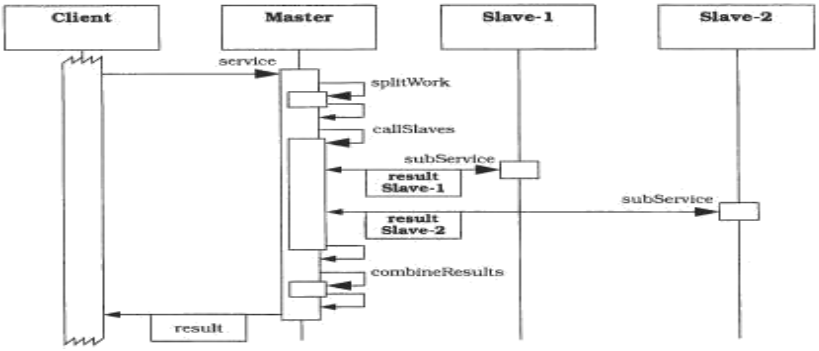
The structure defined by the Master-Slave pattern is illustrated by the following OMT diagram.



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:

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.

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:

- **Matrix multiplication**. Each row in the product matrix can be computed by a separate slave.
- **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.
- Computing the **cross-correlation** of two signals
- The **Workpool model** applies the master-slave pattern to implement process control for parallel computing
- 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

ACCESS CONTROL

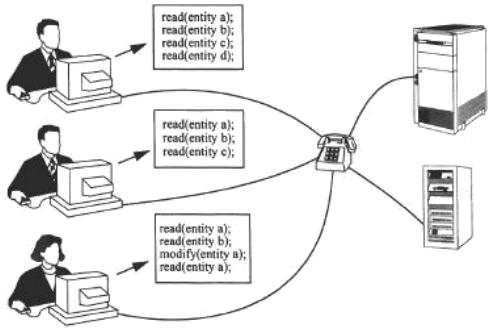
Sometimes a component or even a whole subsystem cannot or should not be accessible directly by its clients. Here we describe one design pattern that helps to protect access to a particular component: → *The proxy design pattern*

PROXY

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.



Context:

A client needs access to the services of another component direct access is technically possible, but may not be the best approach.

Problem:

It is often inappropriate to access a component directly.
A solution to such a design problem has to balance the following forces.

- Accessing the component should be runtime efficient, cost effective and safe for both the client and the component
- Access to component should be transparent and simple for the client. The client should particularly not have to change its calling behavior and syntax from that used to call any other direct access component.
- The client should be well aware of possible performance or financial penalties for accessing remote clients. Full transparency can obscure cost differences between services.

Solution:

- Let the client communicate with a representative rather than the component itself.
- This representative called a 'proxy' offers the interface of the component but performs additional pre and post processing such as access control checking or making read only copies of the 'original'.

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**
  - 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.

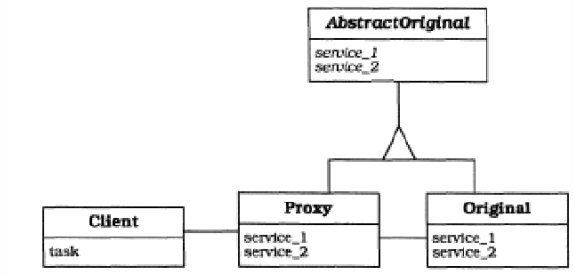
Class: Client	Collaborators: • Proxy
Responsibilities • Uses the interface provided by the proxy to request a particular service. • Fulfills its own task.	

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.	

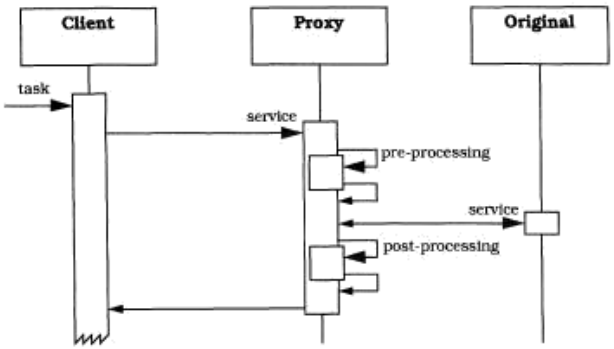
The following **OMT** diagram shows the relationships between the classes graphically:



Dynamics:

The following diagram shows a typical dynamic scenario of a Proxy structure.

- ♣ While working on its task the client asks the proxy to carry out a service.
- ♣ The proxy receives the incoming service request and pre-processes it.
- ♣ If the proxy has to consult the original to fulfill the request, it forwards the request to the original using the proper communication protocols and security measures.
- ♣ The original accepts the request and fulfills it. It sends the response back to the proxy.
- ♣ The proxy receives the response. Before or after transferring it to the client it may carry out additional post-processing actions such as caching the result, calling the destructor of the original or releasing a lock on a resource.



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.

Variants:

- ➢ **Remote proxy:**
  Clients of remote components should be scheduled from network addresses and IPC protocols.
- ➢ **Protection proxy:**
  Components must be protected from unauthorized access.
- ➢ **Cache proxy:**
  Multiple local clients can share results from remote components.
- ➢ **Synchronization proxy:**
  Multiple simultaneous accesses to a component must be synchronized.
- ➢ **Counting proxy:**
  Accidental deletion of components must be prevented or usage statistics collected
- ➢ **Virtual proxy:**
  Processing or loading a component is costly while partial information about the component may be sufficient.
- ➢ **Firewall proxy:**

Local clients should be protected from the outside world.

Known uses:

- **NeXT STEP**
  The Proxy pattern is used in the NeXTSTEP operating system to provide local stubs for remote objects. Proxies are created by a special server on the first access to the remote object.
- **OMG-COBRA**
  It uses the Proxy pattern for two purposes. So called 'client-stubs', or IDL-stubs, guard clients against the concrete implementation of their servers and the Object Request Broker.
- **OLE**
  In Microsoft OLE servers may be implemented as libraries dynamically linked to the address space of the client, or as separate processes. Proxies are used to hide whether a particular server is local or remote from a client.
- **WWW proxy**
  It gives people inside the firewall concurrent access to the outside world. Efficiency is increased by caching recently transferred files.
- **Orbix**
  It is a concrete OMG-CORBA implementation, uses remote proxies. A client can bind to an original by specifying its unique identifier.

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.

UNIT 7 – QUESTION BANK

No.	QUESTION	YEAR	MARKS
1.	Give the structure of whole port design pattern with CRC	Dec 09	5
2.	What are the applications of master slave pattern	Dec 09	10
3.	What are the variants of proxy pattern?	Dec 09	5
4.	Discuss the five steps implementation of master slave pattern	June 10	10
5.	Define proxy design pattern. Discuss the benefits and liabilities of the same	June 10	10
6.	Briefly explain the benefits of master slave design pattern	Dec 10	6
7.	List and explain the steps to implement a whole-part structure	Dec 10	8
8.	With a neat sketch, explain the typical dynamic scenario of a proxy structure	Dec 10	6
9.	Enumerate with explanation the different steps, which constitute the implementation of the whole part structure for a CAD system for 2D modeling.	June 11	14
10.	Briefly comment on the different steps carried out to realize the implementation of the proxy pattern	June 11	6
11.	Explain the variants of whole-part design pattern, in brief	Dec 11	10
12.	Explain the dynamics part of master slave design pattern	Dec 11	8
13.	Mention any two benefits of proxy design pattern	Dec 11	2
14.	Briefly explain the benefits of master slave design pattern	June 12	6
15.	What are the variants of proxy pattern?	June 12	6
16.	List and explain the steps to implement a whole-part structure	June 12	8

UNIT 8
DESIGNING AND DOCUMENTING SOFTWARE ARCHITECTURE

CHAPTER 7:
DESIGNING THE ARCHITECTURE

ARCHITECTURE IN THE LIFE CYCLE

Any organization that embraces architecture as a foundation for its software development processes needs to understand its place in the life cycle. Several life-cycle models exist in the literature, but one that puts architecture squarely in the middle of things is the evolutionary delivery life cycle model shown in figure 7.1.

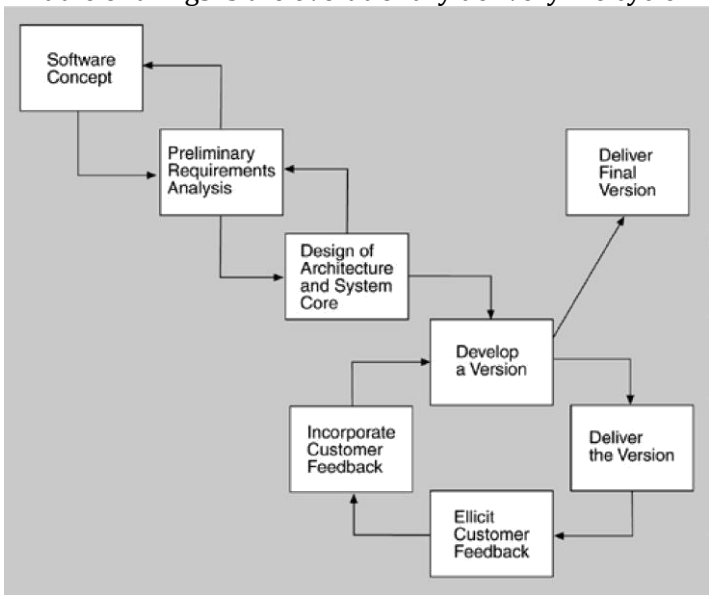


Figure 7.1. Evolutionary Delivery Life Cycle

The intent of this model is to get user and customer feedback and iterate through several releases before the final release. The model also allows the adding of functionality with each iteration and the delivery of a limited version once a sufficient set of features has been developed.

WHEN CAN I BEGIN DESIGNING?

- The life-cycle model shows the design of the architecture as iterating with preliminary requirements analysis. Clearly, you cannot begin the design until you have some idea of the system requirements. On the other hand, it does not take many requirements in order for design to begin.
- An architecture is "shaped" by some collection of functional, quality, and business requirements. We call these shaping requirements *architectural drivers* and we see examples of them in our case studies like modifiability, performance requirements availability requirements and so on.
- To determine the architectural drivers, identify the highest priority business goals. There should be relatively few of these. Turn these business goals into quality scenarios or use cases.

7.2 DESIGNING THE ARCHITECTURE

A method for designing an architecture to satisfy both quality requirements and functional requirements is called attribute-driven design (ADD). ADD takes as input a set of quality attribute scenarios and employs knowledge about the relation between quality attribute achievement and architecture in order to design the architecture.

ATTRIBUTE DRIVEN DESIGN

ADD is an approach to defining a software architecture that bases the decomposition process on the quality attributes the software has to fulfill. It is a recursive decomposition process where, at each stage, tactics and architectural patterns are chosen to satisfy a set of quality scenarios and then functionality is allocated to instantiate the module types provided by the pattern.

The output of ADD is the first several levels of a module decomposition view of an architecture and other views as appropriate.

Garage door opener example

- Design a product line architecture for a garage door opener with a larger home information system the opener is responsible for raising and lowering the door via a switch, remote control, or the home information system. It is also possible to diagnose problems with the opener from within the home information system.
- **Input** to ADD: a set of requirements
  - Functional requirements as use cases
  - Constraints
  - Quality requirements expressed as system specific quality scenarios
- **Scenarios** for garage door system
  - Device and controls for opening and closing the door are different for the various products in the product line
  - The processor used in different products will differ
  - If an obstacle is (person or object) is detected by the garage door during descent, it must stop within 0.1 second
  - The garage door opener system needs to be accessible from the home information system for diagnosis and administration.
  - It should be possible to directly produce an architecture that reflects this protocol

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.

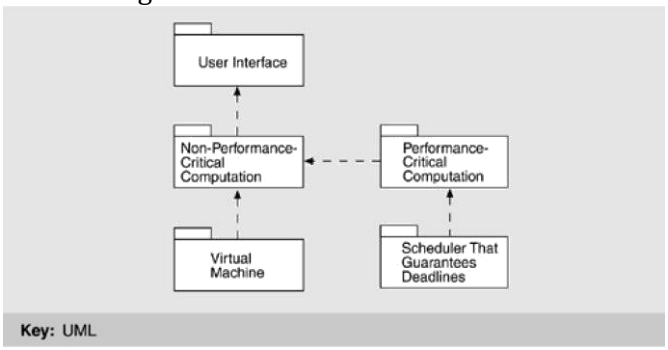


Figure 7.2. Architectural pattern that utilizes tactics to achieve garage door drivers

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

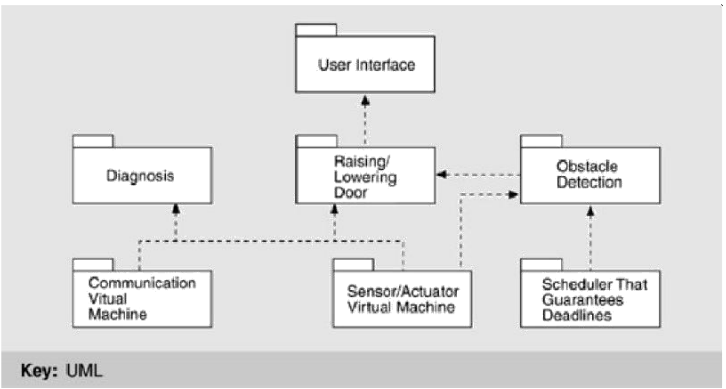


Figure 7.3. First-level decomposition of garage door opener

♥ *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
- It documents what this module provides to others.
- 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 ○ *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

7.3 FORMING THE TEAM STRUCTURES

- ♥ Once the architecture is accepted we assign teams to work on different portions of the design and development.
- ♥ Once architecture for the system under construction has been agreed on, teams are allocated to work on the major modules and a work breakdown structure is created that reflects those teams.
- ♥ Each team then creates its own internal work practices.
- ♥ For large systems, the teams may belong to different subcontractors.
- ♥ Teams adopt "work practices' including
  - Team communication via website/bulletin boards
  - Naming conventions for files
  - Configuration/revision control system
  - Quality assurance and testing procedure

The teams within an organization work on modules, and thus within team high level of communication is necessary

7.4 CREATING A SKELETAL SYSTEM

- ✓ Develop a skeletal system for the incremental cycle.
- ✓ Classical software engineering practice recommends -> "stubbing out"
- ✓ Use the architecture as a guide for the implementation sequence
- ✓ First implement the software that deals with execution and interaction of architectural components
  - Communication between components
  - Sometimes this is just install third-party middleware
- ✓ Then add functionality
  - By risk-lowering
  - Or by availability of staff

Once the elements providing the next increment of functionality have been chosen, you can employ the uses structure to tell you what additional software should be running correctly in the system to support that functionality. This process continues, growing larger and larger increments of the system, until it is all in place.

CHAPTER 9:
DOCUMENTING SOFTWARE ARCHITECTURES

9.1 USES OF ARCHITECTURAL DOCUMENTATION

- Architecture documentation is both prescriptive and descriptive. That is, for some audiences it prescribes what should be true by placing constraints on decisions to be made. For other audiences it describes what is true by recounting decisions already made about a system's design.
- All of this tells us that different stakeholders for the documentation have different needs—different kinds of information, different levels of information, and different treatments of information.
- One of the most fundamental rules for technical documentation in general, and software architecture documentation in particular, is to write from the point of view of the reader. Documentation that was easy to write but is not easy to read will not be used, and "easy to read" is in the eye of the beholder—or in this case, the stakeholder.
- Documentation facilitates that communication. Some examples of architectural stakeholders and the information they might expect to find in the documentation are given in Table 9.1.
- In addition, each stakeholders come in two varieties: seasoned and new. A new stakeholder will want information similar in content to what his seasoned counterpart wants, but in smaller and more introductory doses. Architecture documentation is a key means for educating people who need an overview: new developers, funding sponsors, visitors to the project, and so forth.

Table 9.1. Stakeholders and the Communication Needs Served by Architecture

Stakeholder	Use
Architect and requirements engineers who represent customer(s)	To negotiate and make tradeoffs among competing requirements
Architect and designers of constituent parts	To resolve resource contention and establish performance and other kinds of runtime resource consumption budgets
Implementors	To provide inviolable constraints (plus exploitable freedoms) on downstream development activities
Testers and integrators	To specify the correct black-box behavior of the pieces that must fit together
Maintainers	To reveal areas a prospective change will affect
Designers of other systems with which this one must interoperate	To define the set of operations provided and required, and the protocols for their operation
Quality attribute specialists	To provide the model that drives analytical tools such as rate-monotonic real-time schedulability analysis, simulations and simulation generators, theorem provers, verifiers, etc. These tools require information about resource consumption, scheduling policies, dependencies, and so forth. Architecture documentation must contain the information necessary to evaluate a variety of quality attributes such as security, performance, usability, availability, and modifiability. Analyses for each attributes have their own information needs.
Managers	To create development teams corresponding to work assignments identified, to plan and allocate project resources, and to track progress by the various teams
Product line managers	To determine whether a potential new member of a product family is in or out of scope, and if out by how much
Quality assurance team	To provide a basis for conformance checking, for assurance that implementations have been faithful to the architectural prescriptions

Source: Adapted from Clements 03

9.2 VIEWS

The concept of a view, which you can think of as capturing a structure, provides us with the basic principle of documenting software architecture

> *Documenting an architecture is a matter of documenting the relevant views and then adding documentation that applies to more than one view.*

This principle is useful because it breaks the problem of architecture documentation into more tractable parts, which provide the structure for the remainder of this chapter:

- Choosing the relevant views
- Documenting view
- Documenting information that applies to more than one view

9.3 CHOOSING THE RELEVANT VIEWS

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:**

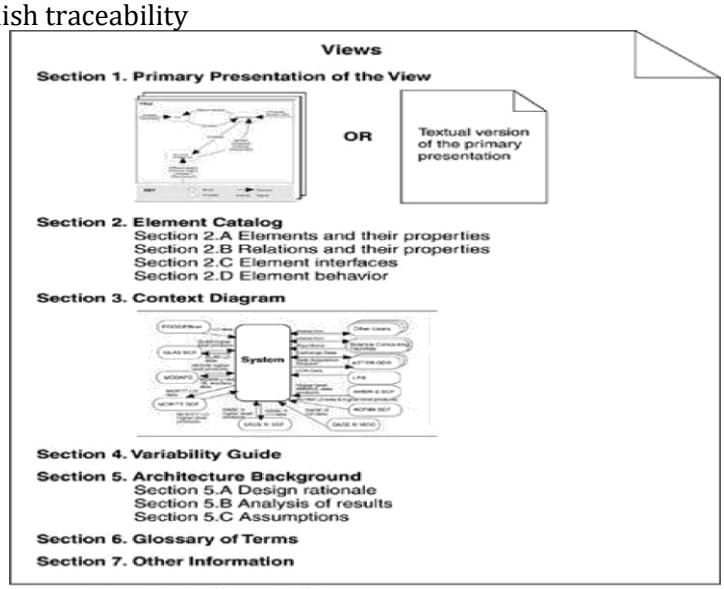
   The candidate view list from step 1 is likely to yield an impractically large number of views. To reduce the list to a manageable size, first look for views in the table that require only overview depth or that serve very few stakeholders. See if the stakeholders could be equally well served by another 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.

9.4 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
  - The options among which a choice is to be made
  - 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
  - rationale, explaining why the decisions reflected in the view were made and why alternatives were rejected
  - analysis results, which justify the design or explain what would have to change in the face of a modification
  - assumptions reflected in the design
- **Glossary of terms** 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



Source: Adapted from [Clements 03].

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.

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.

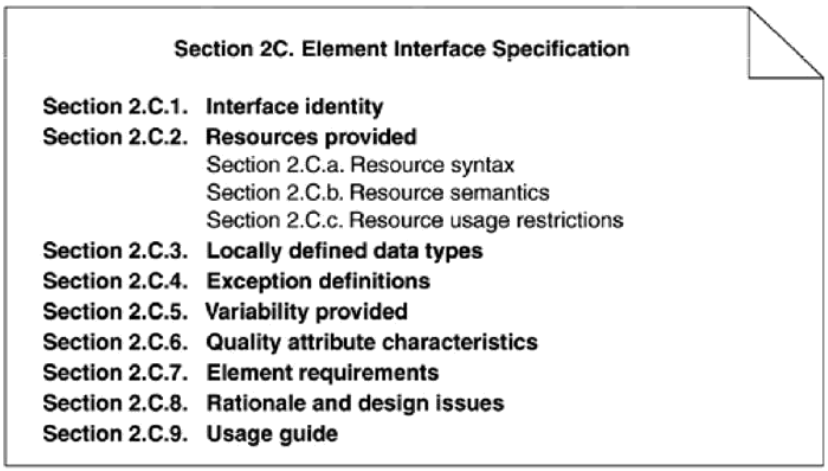
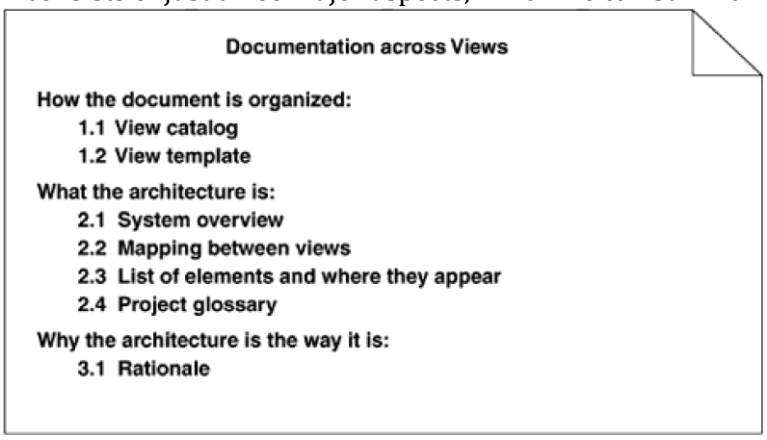


Figure 9.2. The nine parts of interface documentation

9.5 DOCUMENTATION ACROSS VIEWS

Cross-view documentation consists of just three major aspects, which we can summarize as how-what-why:



Source: Adapted from [Clements 03].

Figure 9.3. Summary of cross-view documentation

HOW THE DOCUMENTATION IS ORGANIZED TO SERVE A STAKEHOLDER

Every suite of architectural documentation needs an introductory piece to explain its organization to a novice stakeholder and to help that stakeholder access the information he or she is most interested in. There are two kinds of "how" information:

★ **View Catalog**

A view catalog is the reader's introduction to the views that the architect has chosen to include in the suite of documentation.

There is one entry in the view catalog for each view given in the documentation suite. Each entry should give the following:

- The name of the view and what style it instantiates
- A description of the view's element types, relation types, and properties
- A description of what the view is for
- Management information about the view document, such as the latest version, the location of the view document, and the owner of the view document

★ **View Template**

A view template is the standard organization for a view. It helps a reader navigate quickly to a section of interest, and it helps a writer organize the information and establish criteria for knowing how much work is left to do.

WHAT THE ARCHITECTURE IS

This section provides information about the system whose architecture is being documented, the relation of the views to each other, and an index of architectural elements.

♣ **System Overview**

This is a short prose description of what the system's function is, who its users are, and any important background or constraints. The intent is to provide readers with a consistent mental model of the system and its purpose. Sometimes the project at large will have a system overview, in which case this section of the architectural documentation simply points to that.

♣ **Mapping between Views**

Since all of the views of an architecture describe the same system, it stands to reason that any two views will have much in common. Helping a reader of the documentation understand the relationships among views will give him a powerful insight into how the architecture works as a unified conceptual whole. Being clear about the relationship by providing mappings between views is the key to increased understanding and decreased confusion.

♣ **Element List**

The element list is simply an index of all of the elements that appear in any of the views, along with a pointer to where each one is defined. This will help stakeholders look up items of interest quickly.

- **Project Glossary**

  The glossary lists and defines terms unique to the system that have special meaning. A list of acronyms, and the meaning of each, will also be appreciated by stakeholders. If an appropriate glossary already exists, a pointer to it will suffice here.

WHY THE ARCHITECTURE IS THE WAY IT IS: RATIONALE

Cross-view rationale explains how the overall architecture is in fact a solution to its requirements. One might use the rationale to explain:

- The implications of system-wide design choices on meeting the requirements or satisfying constraints.
- The effect on the architecture when adding a foreseen new requirement or changing an existing one.
- The constraints on the developer in implementing a solution.
- Decision alternatives that were rejected.

In general, the rationale explains why a decision was made and what the implications are in changing it.

UNIT 8 – QUESTION BANK

No.	QUESTION	YEAR	MARKS
1.	What are the three steps for choosing views for a project?	Dec 09	6
2.	Write a note on view catalog	Dec 09	4
3.	What are the options for representing connectors and systems in UML? ★ **OUT OF SYLLABUS**	Dec 09	10
4.	Explain with a neat diagram, the evolutionary delivery life cycle model	June 10	8
5.	What are the suggested standard organization points for interface documentation?	June 10	12
6.	List the steps of ADD	Dec 10	4
7.	Write a note on creating a skeletal system	Dec 10	6
8.	What are the uses of architectural documentation? Bring out the concept of view as applied to architectural documentation.	Dec 10	10
9.	Briefly explain the different steps performed while designing an architecture using the ADD method	June 11	10
10.	write short notes on: i)forming team structures ii)documenting across views iii)documenting interfaces	June 11	10
11.	Explain the steps involved in designing an architecture, using the attribute driven design	Dec 11	10
12.	"Architecture serves as a communication vehicle among stakeholders. Documentation facilitates that communication." Justify.	Dec 11	10
13.	List the steps of ADD method of architectural design	June 12	6
14.	Explain with a neat diagram, the evolutionary delivery life cycle model	June 12	6
15.	What are the suggested standard organization points for view documentation?	June 12	8