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.

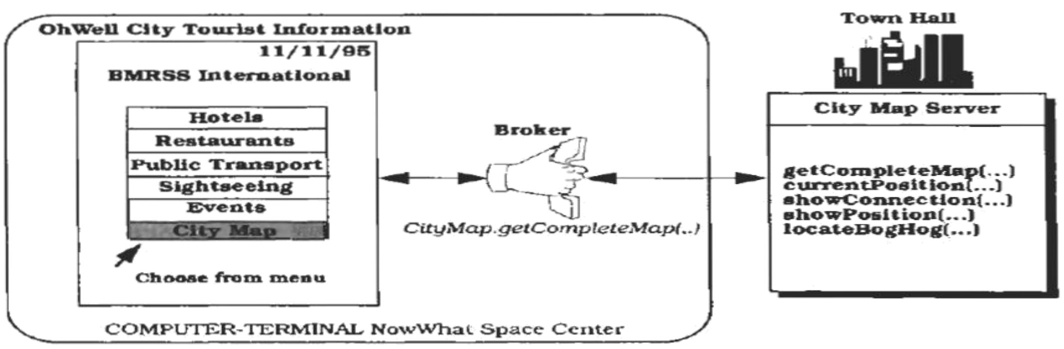
We introduce three patterns related to distributed systems in this category:

- 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:

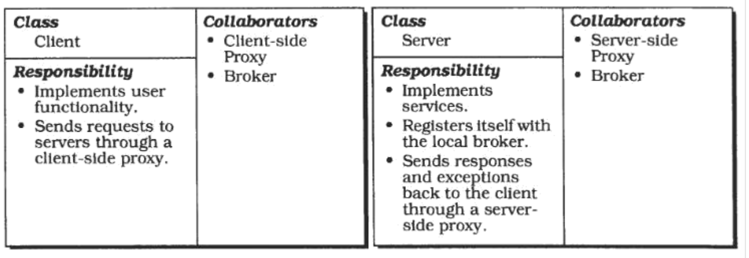
- Introduce a broker component to achieve better decoupling of clients and servers.
- Servers registers themselves with the broker make their services available to clients through method interfaces.
- Clients access the functionality of servers by sending requests via the broker.
- A broker's tasks include 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.

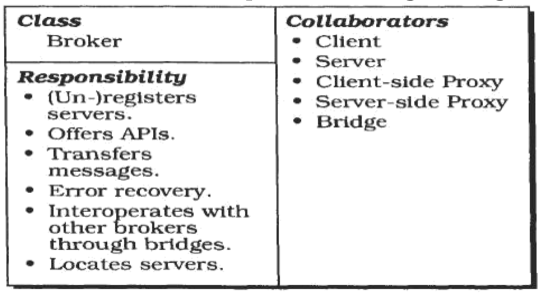


Class Client	Collaborators
Responsibility • Implements user functionality. • Sends requests to servers through a client-side proxy.	• Client-side Proxy • Broker

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:**

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



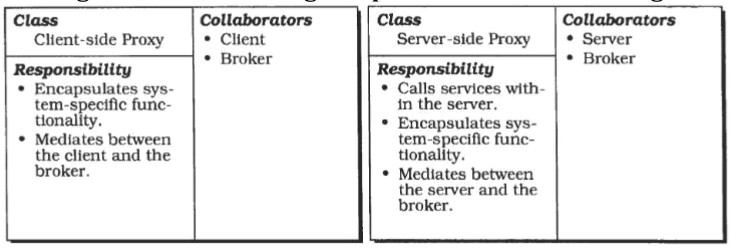
Class Broker	Collaborators
Responsibility • (Un-)registers servers. • Offers APIs. • Transfers messages. • Error recovery. • Interoperates with other brokers through bridges. • Locates servers.	• Client • Server • Client-side Proxy • Server-side Proxy • Bridge

★ **Client side proxy:**

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

★ **Server side proxy:**

- o 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.

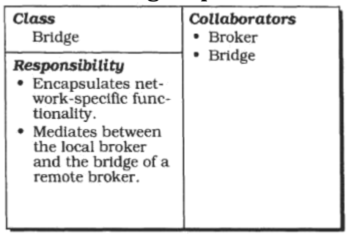


Class Client-side Proxy	Collaborators
Responsibility • Encapsulates system-specific functionality. • Mediates between the client and the broker.	• Client • Broker

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

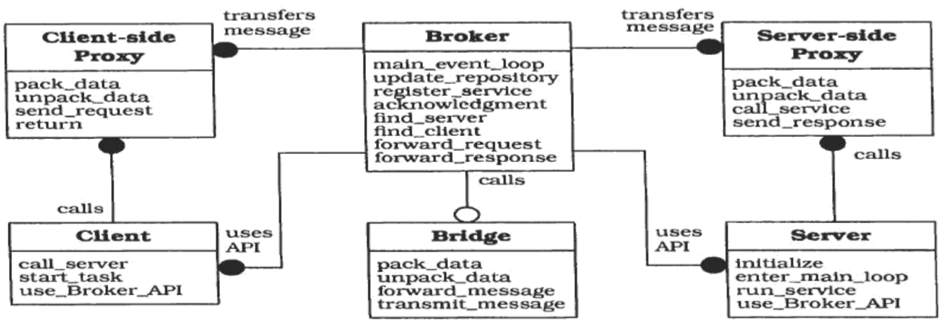
★ **Bridges:**

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



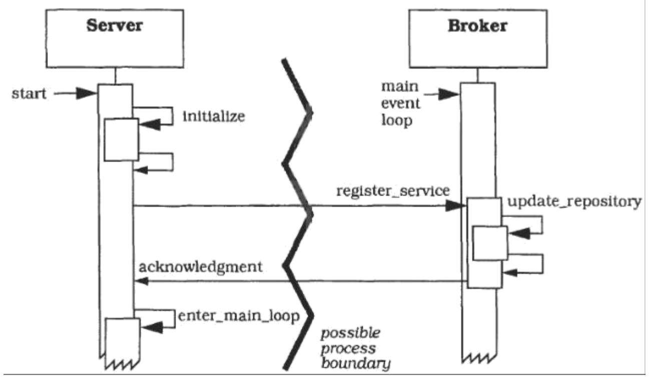
Class Bridge	Collaborators
Responsibility • Encapsulates network-specific functionality. • Mediates between the local broker and the bridge of a remote broker.	• Broker • Bridge

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

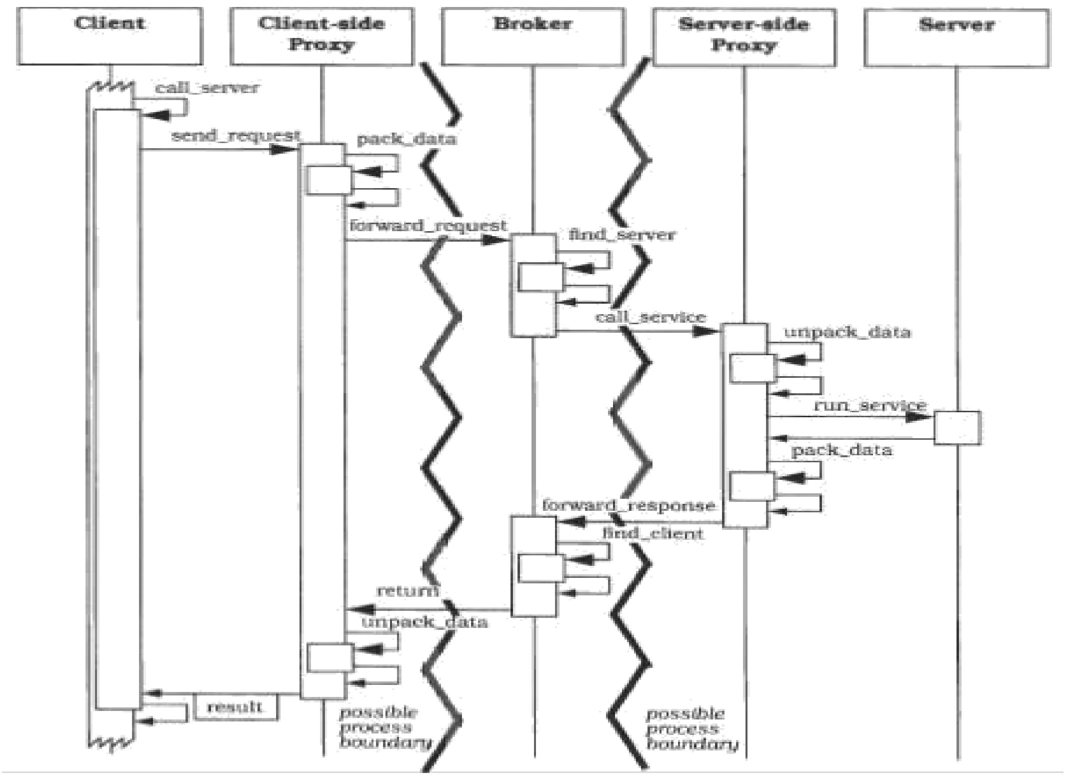


Dynamics:

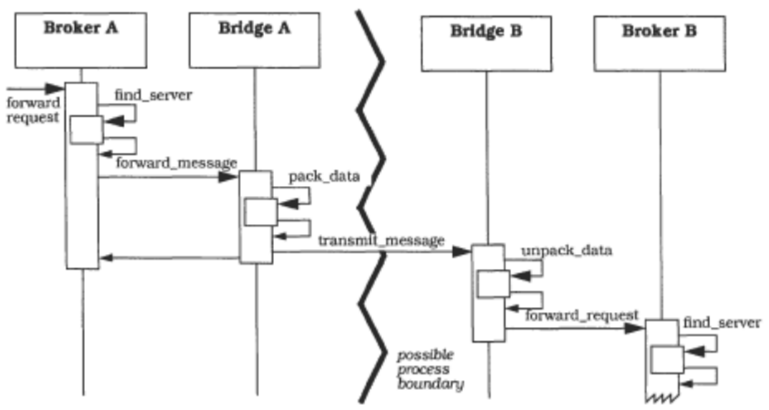
- **Scenario 1.** illustrates the behaviour when a server registers itself with the local broker component:



- **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.

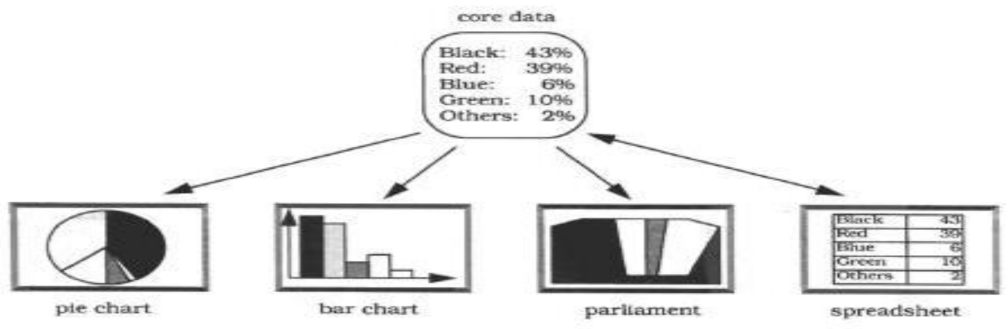
Two patterns that provide a fundamental structural organization for interactive software systems are:

- ➢ 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:

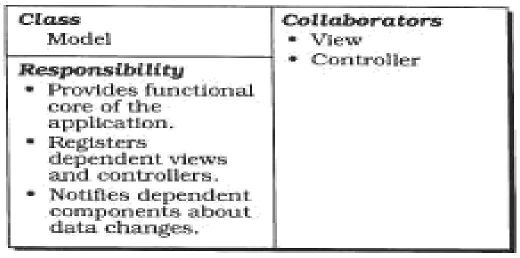
- ✓ Same information is presented differently in different windows. For ex: In a bar or pie chart.
- ✓ The display and behavior of the application must reflect data manipulations immediately.
- ✓ Changes to the user interface should be easy, and even possible at run-time.
- ✓ Supporting different 'look 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:

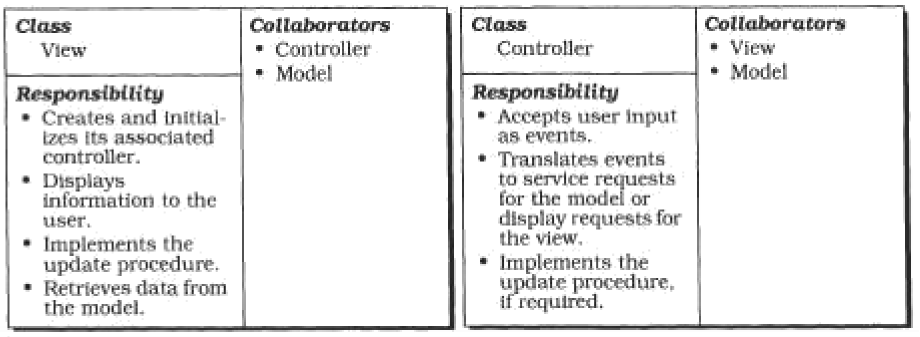
- ★ **Model component:**
  - o Contains the functional core of the application.
  - o Registers dependent views and controllers
  - o Notifies dependent components about data changes (change propagation mechanism)



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

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

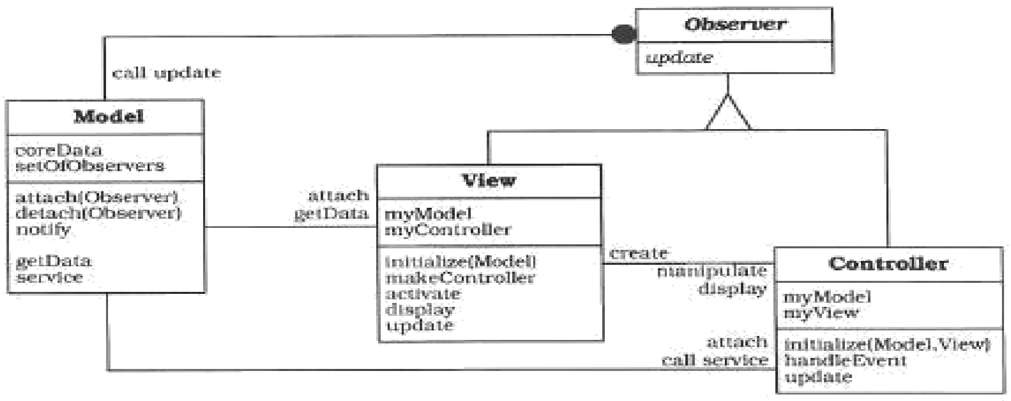
- The controller registers itself with the change-propagation mechanism and implements an update procedure.



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

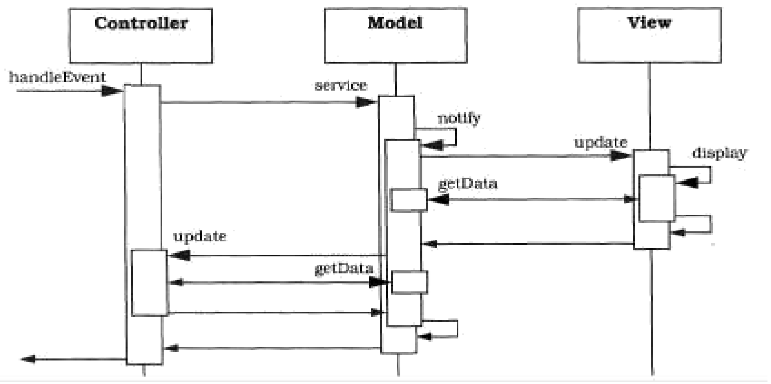
An object-oriented implementation of MVC would define a separate class for each component. In a C++ implementation, view and controller classes share a common parent that defines the update interface. This is shown in the following diagram.



Dynamics:

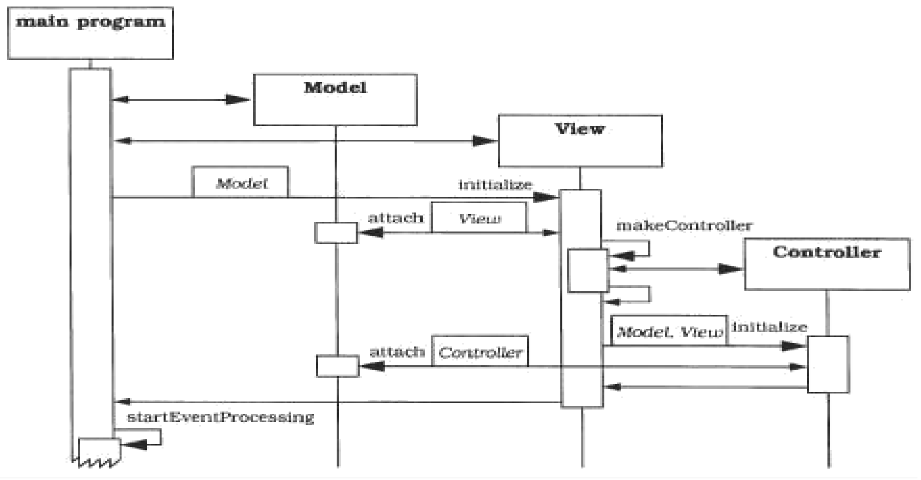
The following scenarios depict the dynamic behavior of MVC. For simplicity only one view-controller pair is shown in the diagrams.

- **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.

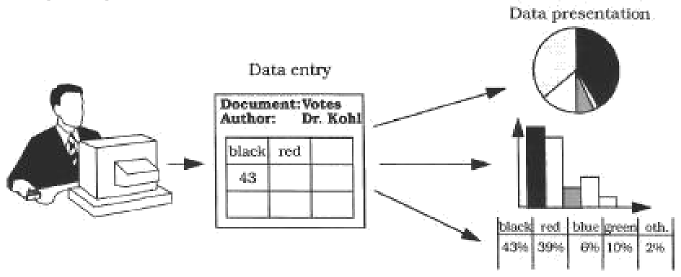
Every agent is responsible for a specific aspect of te applications functionality and consists of three components

- ✓ *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:

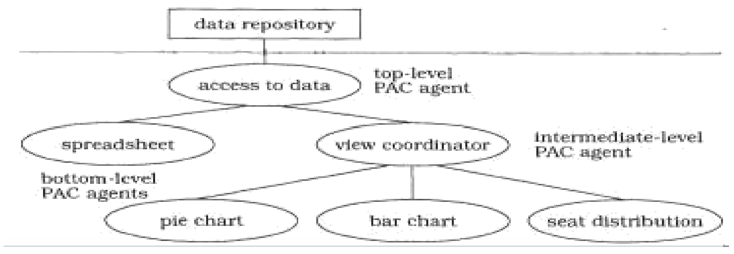
Agents specialized in human-comp interaction accept user input and display data. Other agents maintain the data model of the system and offer functionality that operates on this data. Additional agents are responsible for error handling or communication with other software systems The following forces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

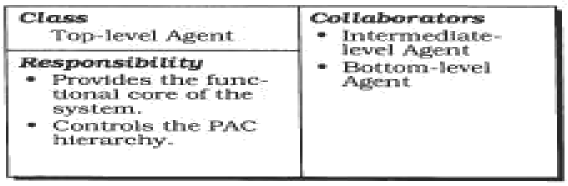
- Main responsibility is to provide the global data model of the software. This is maintained in the abstraction component of top level agent.
- The presentation component of the top level agent may include user-interface elements common to whole application.
- The control component has three responsibilities
  - 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:**

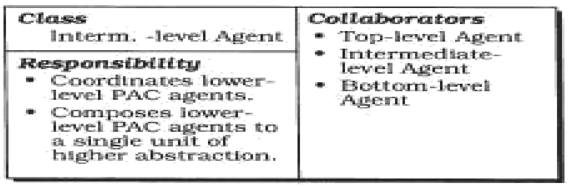
- Represents a specific semantic concept of the application domain, such as mail box in a n/w management system.
- The presentation component of bottom level PAC agents presents a specific view of the corresponding semantic concept, and provides access to all the functions users can apply to it.
- The 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**

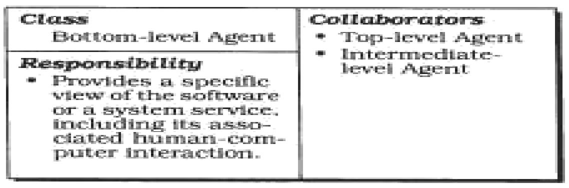
- Can fulfill two different roles: composition and co ordination.
- It defines a new abstraction, whose behavior encompasses both the behavior of its component, and the new characteristics that are added to the composite object.
- Abstraction component maintains the specific data of the intermediate 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

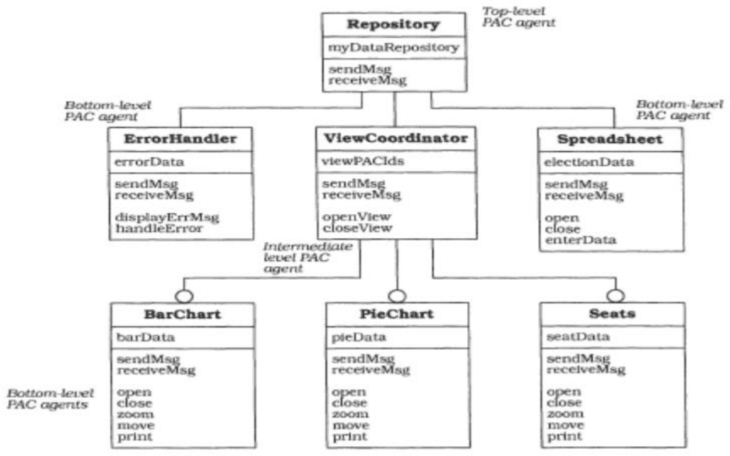


Class Interm. -level Agent	**Collaborators**
Responsibility • Coordinates lower-level PAC agents. • Composes lower-level PAC agents to a single unit of higher abstraction.	• Top-level Agent • Intermediate-level Agent • Bottom-level Agent



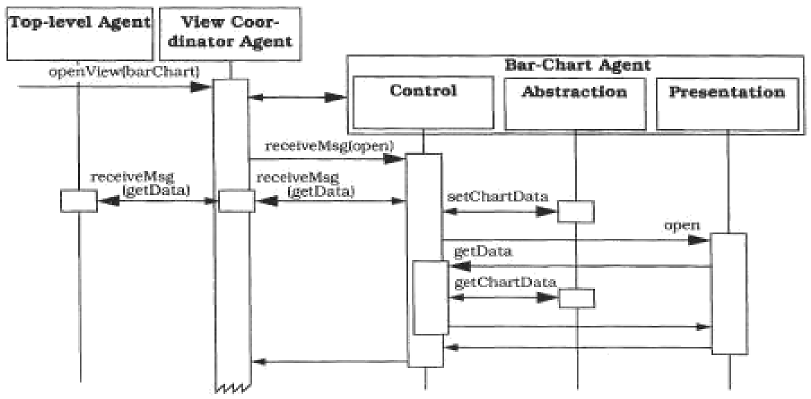
Class Bottom-level Agent	**Collaborators**
Responsibility • Provides a specific view of the software or a system service, including its associated human-computer interaction.	• Top-level Agent • Intermediate-level Agent

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

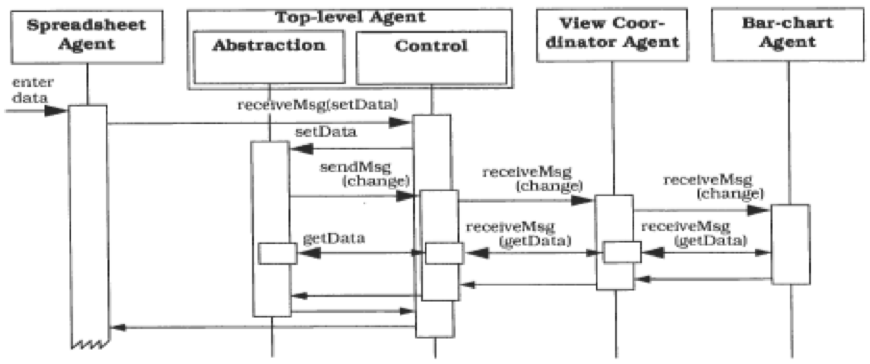


Dynamics:

- **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