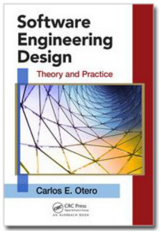
Chapter 4: Patterns and Styles in Software Architecture

Session III: Interactive and Hierarchical Systems

Software Engineering Design: Theory and Practice
by Carlos E. Otero



SESSION'S AGENDA

- ➢ Distributed Systems (cont)
  - ✓ Overview
  - ✓ Patterns
    - ▪ Broker
- ➢ Interactive Systems
  - ✓ Overview
  - ✓ Patterns
    - ▪ Model-View-Controller
- ➢ Hierarchical Systems
  - ✓ Main program and Subroutine
  - ✓ Layered

DISTRIBUTED SYSTEMS

- ➢ As refresher, distributed systems are decomposed into multiple processes that (typically) collaborate through the network.
  - ✓ These systems are ubiquitous in today's modern systems thanks to wireless, mobile, and internet technology.
  - ✓ These types of distributed systems are easy to spot, since their deployment architecture entails multiple physical nodes.
  - ✓ However, with the advent of multi-core processors, distributed architectures are also relevant to software that executes on a single node with multiprocessor capability.

- ➢ Some examples of distributed systems include:
  - ✓ Internet systems, web services, file- or music-sharing systems, high-performance systems, etc.

- ➢ Common architectural patterns for distributed systems include:
  - ✓ Client-Server Pattern (we discussed this one in last session)
  - ✓ ***Broker Pattern***

BROKER ARCHITECTURAL PATTERN

- The Broker architectural pattern provides mechanisms for achieving better flexibility between clients and servers in a distributed environment.
  - ✓ In a typical client-server system, clients are tightly coupled with servers.
  - ✓ This leads to complexity for systems that need to provide services from multiple servers hosted at different locations.

- In some software systems, clients (in a distributed environment) need to be able to access services from multiple servers without known their actual locations or particular details of communication.
  - ✓ When these concerns are separated, it leads to systems that are flexible and interoperable.

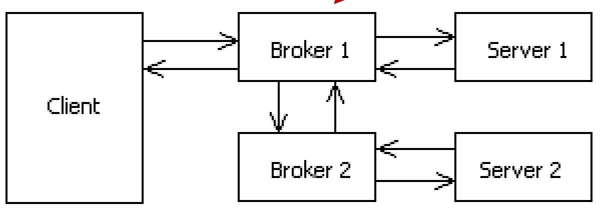
- The broker pattern decreases coupling between client and servers by mediating between them so that one client can transparently access the services of multiple servers

Broker Architectural Pattern

- The Broker architectural pattern includes the following components:

Component	Description
Client	Applications that use the services provided by one or more servers.
ClientProxy	Component that provides transparency (at client) between remote and local components so that remote components appear as local ones.
Broker	Component that mediates between client and server components.
ServerProxy	Component that provides transparency (at server) between remote and local components so that remote components appear as local ones.
Server	Provide services to clients. May also act as client to the Broker.
Bridge	Optional component for encapsulating interoperation among Brokers.

- An box-and-line example of the broker architecture is presented below.

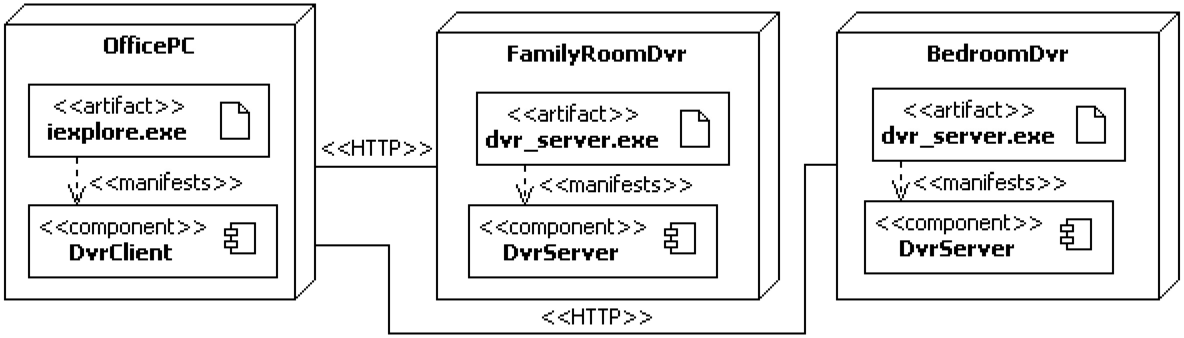


Client relies on the Broker component to access the services from the server

Broker can communicate with other Brokers that provide access to services from other servers

BROKER ARCHITECTURAL PATTERN

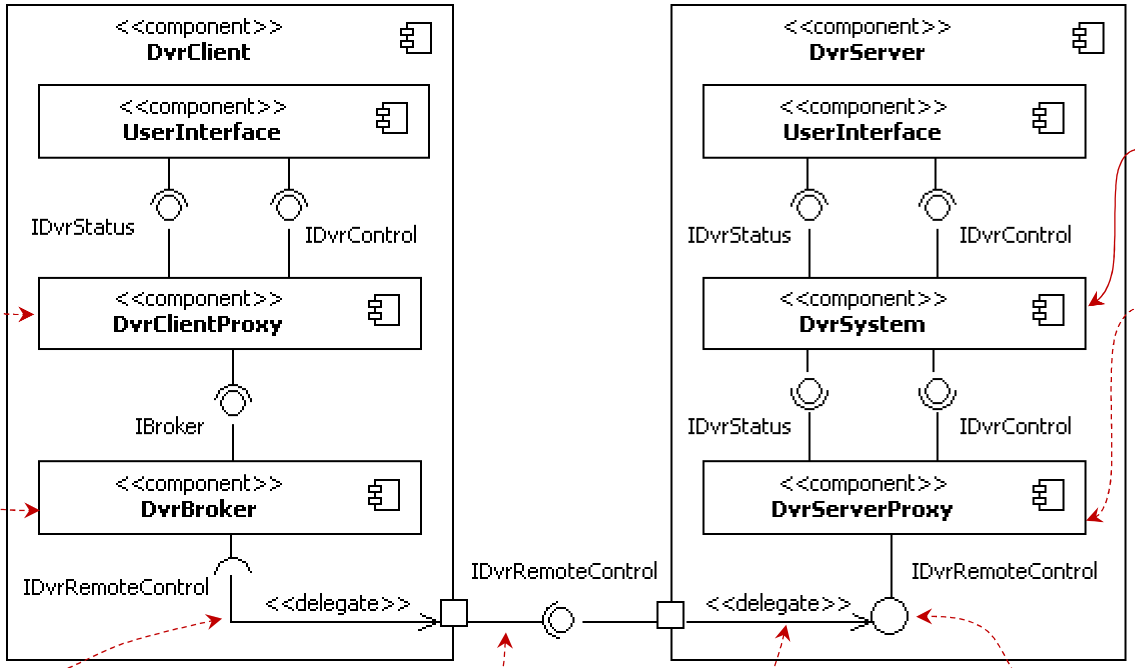
- Consider the typical DVR system found in many houses today.
  - ✓ In this example, there are two DVRs
    - One in the family room
    - The other in a bedroom
  - ✓ DVR services can be accessed from inside the house, using the home office PC.
    - This could be done in many ways, but for simplicity, assume that access is obtained via the browser, using a Java Applet, Microsoft Active X, etc.
    - To gain some insight into the physical architecture of the system, take a look at the following deployment diagram.



Broker Architectural Pattern

<u>Important</u>:
Remember, this is NOT the physical view of the system!!

- The logical architecture can be designed as seen below:



Provides transparency between remote and local components so that remote component appears as local

Mediates between client and server

The needs of this required interface have been delegated to an unnamed port

DvrBroker requires IDvrRemoteControl to enable recordings, access list of recorded shows, or get some other information

Delegate the responsibility of providing this interface to the DvrServerProxy component

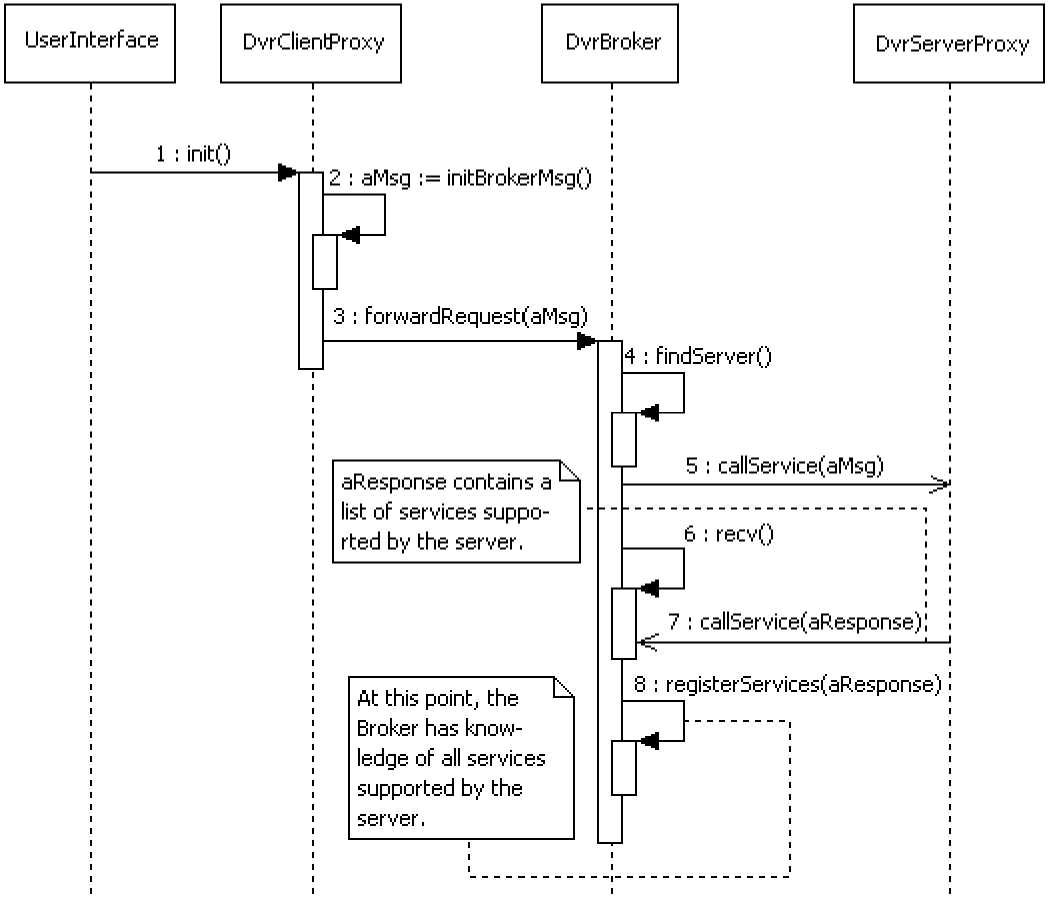
Local component that interfaces with the actual DVR hardware

Allow remote clients to interface with the DvrSystem as if they were local

The remote control interface

Broker Architectural Pattern

Initializing the Broker System



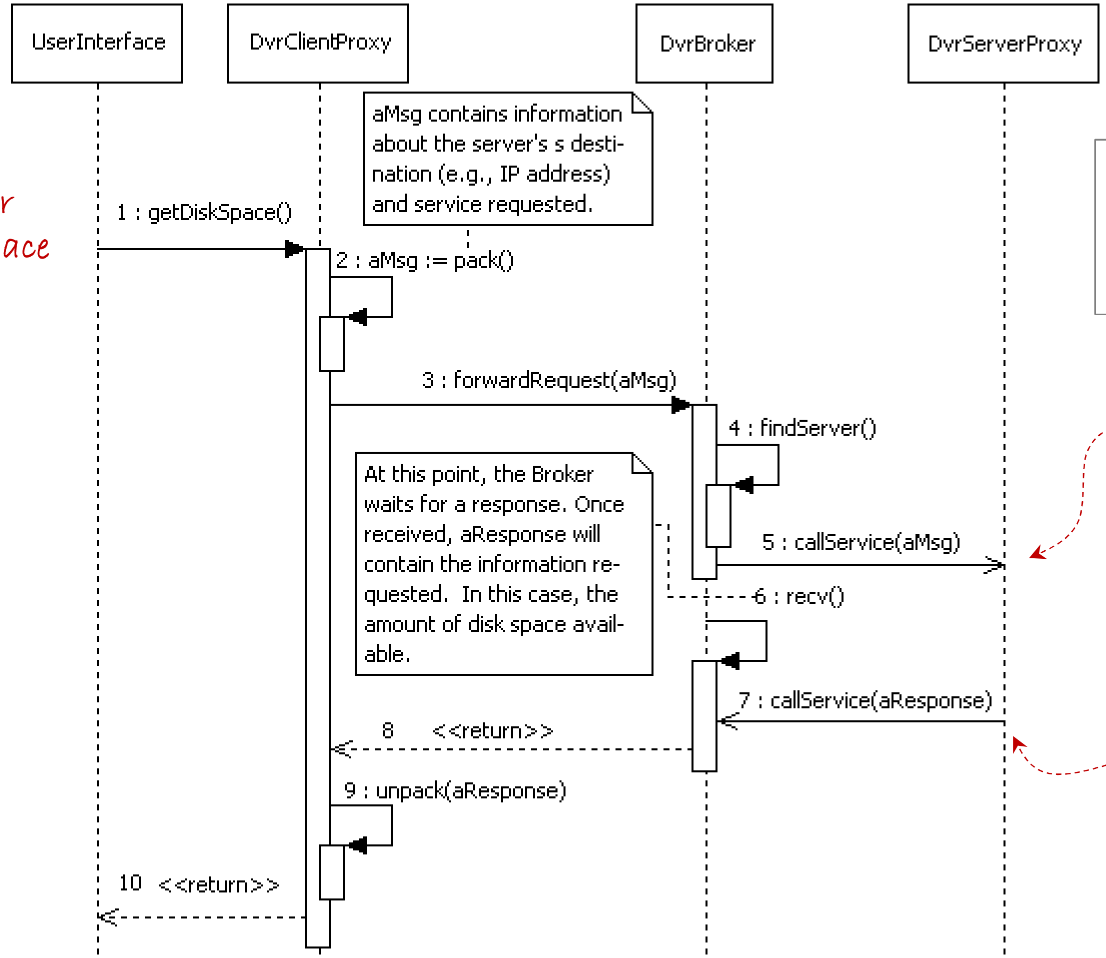
aResponse contains a list of services supported by the server.

At this point, the Broker has knowledge of all services supported by the server.

Important: Remember, this is NOT the physical view of the system!!

BROKER ARCHITECTURAL PATTERN

Requesting for available disk space



aMsg contains information about the server's s destination (e.g., IP address) and service requested.

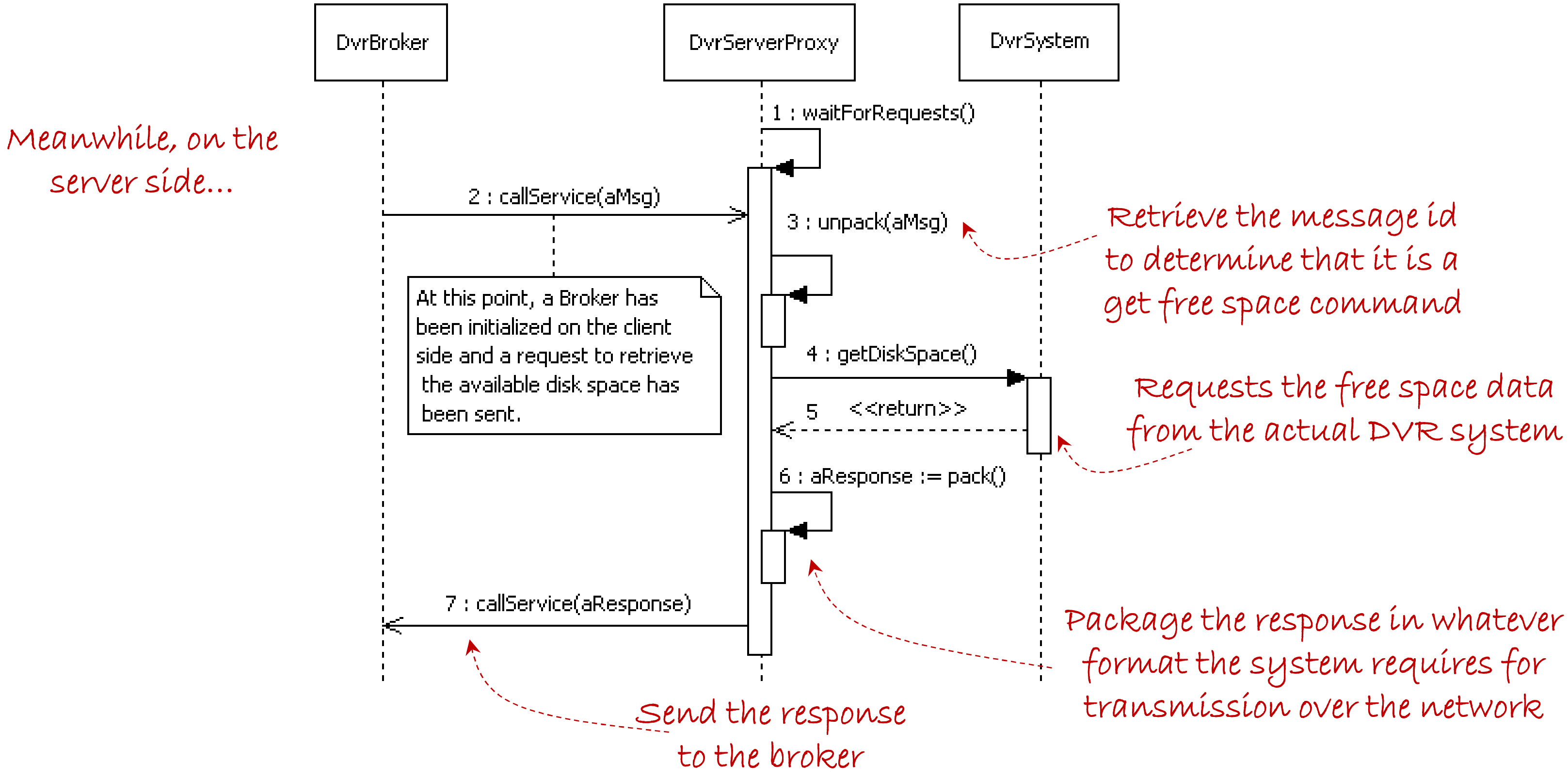
At this point, the Broker waits for a response. Once received, aResponse will contain the information requested. In this case, the amount of disk space available.

<u>Important</u>:
Remember, this is NOT the physical view of the system!!

An async message is sent out to the server

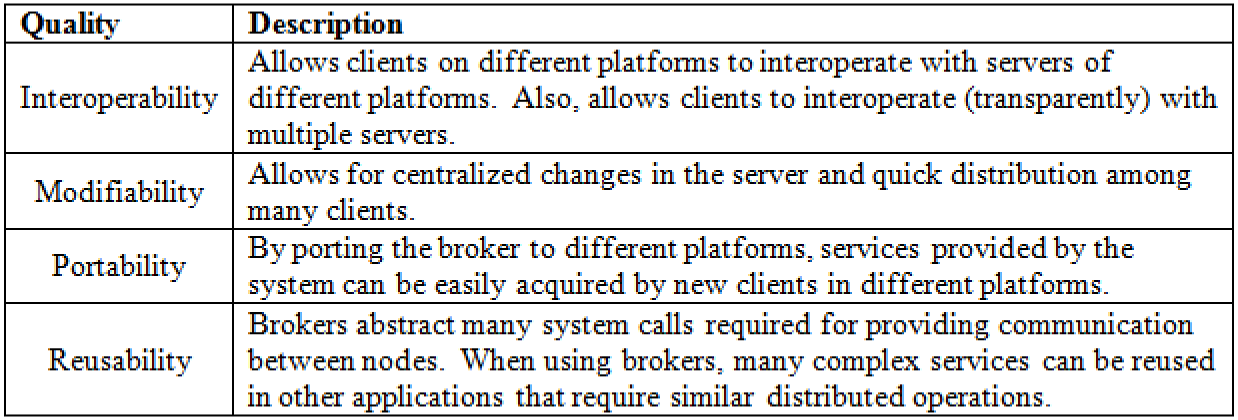
An async message is received with the response

BROKER ARCHITECTURAL PATTERN


DvrBroker
DvrServerProxy
DvrSystem
1 : waitForRequests()
2 : callService(aMsg)
3 : unpack(aMsg)
4 : getDiskSpace()
5
<<return>>
6 : aResponse := pack()
7 : callService(aResponse)
At this point, a Broker has been initialized on the client side and a request to retrieve the available disk space has been sent.
Meanwhile, on the server side...
Retrieve the message id to determine that it is a get free space command
Requests the free space data from the actual DVR system
Package the response in whatever format the system requires for transmission over the network
Send the response to the broker

BROKER ARCHITECTURAL PATTERN

- Quality properties of the Broker architectural pattern include the ones specified below.



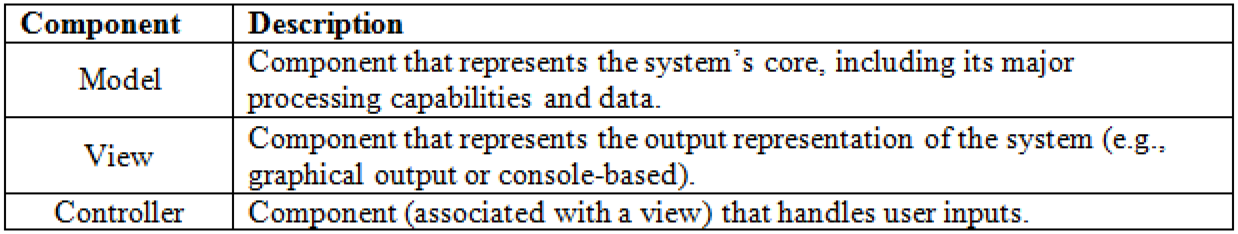
Quality	Description
Interoperability	Allows clients on different platforms to interoperate with servers of different platforms. Also, allows clients to interoperate (transparently) with multiple servers.
Modifiability	Allows for centralized changes in the server and quick distribution among many clients.
Portability	By porting the broker to different platforms, services provided by the system can be easily acquired by new clients in different platforms.
Reusability	Brokers abstract many system calls required for providing communication between nodes. When using brokers, many complex services can be reused in other applications that require similar distributed operations.

INTERACTIVE SYSTEMS

- Interactive systems support user interactions, typically through user interfaces.
  - ✓ When designing these systems, two main quality attributes are of interest:
    - Usability
    - Modifiability

- The mainstream architectural pattern employed in most interactive systems is the Model-View-Controller (MVC).

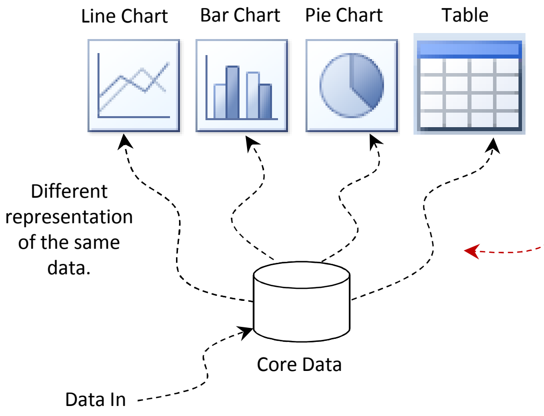
- The MVC pattern is used in interactive applications that require flexible incorporation of human-computer interfaces. With the MVC, systems are decomposed into three main types of components:



Component	**Description**
Model	Component that represents the system's core, including its major processing capabilities and data.
View	Component that represents the output representation of the system (e.g., graphical output or console-based).
Controller	Component (associated with a view) that handles user inputs.

Model-View-Controller Architectural Pattern

Consider the popular example where data needs to be represented in different formats

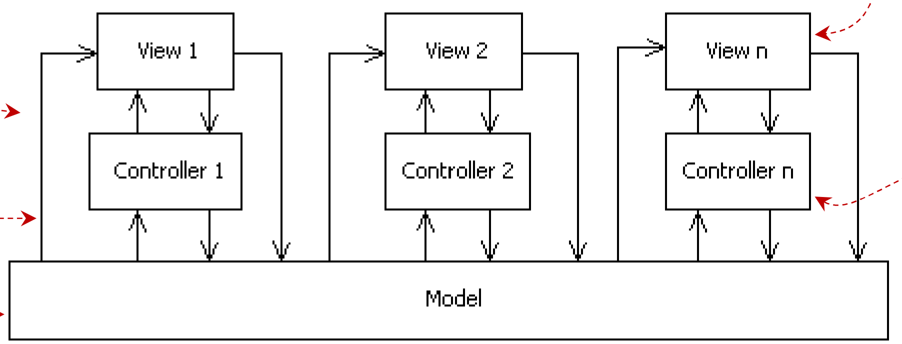


When data changes, all views are updated to reflect the changes.

Box-and-line diagram of the MVC architectural pattern

When Model changes, it updates its Views

<u>Model</u>: Encapsulates core data and functionality



<u>View</u>: Display information to the user. Every View is associated with a Controller.

<u>Controller</u>: Receives input and translates it to service requests.

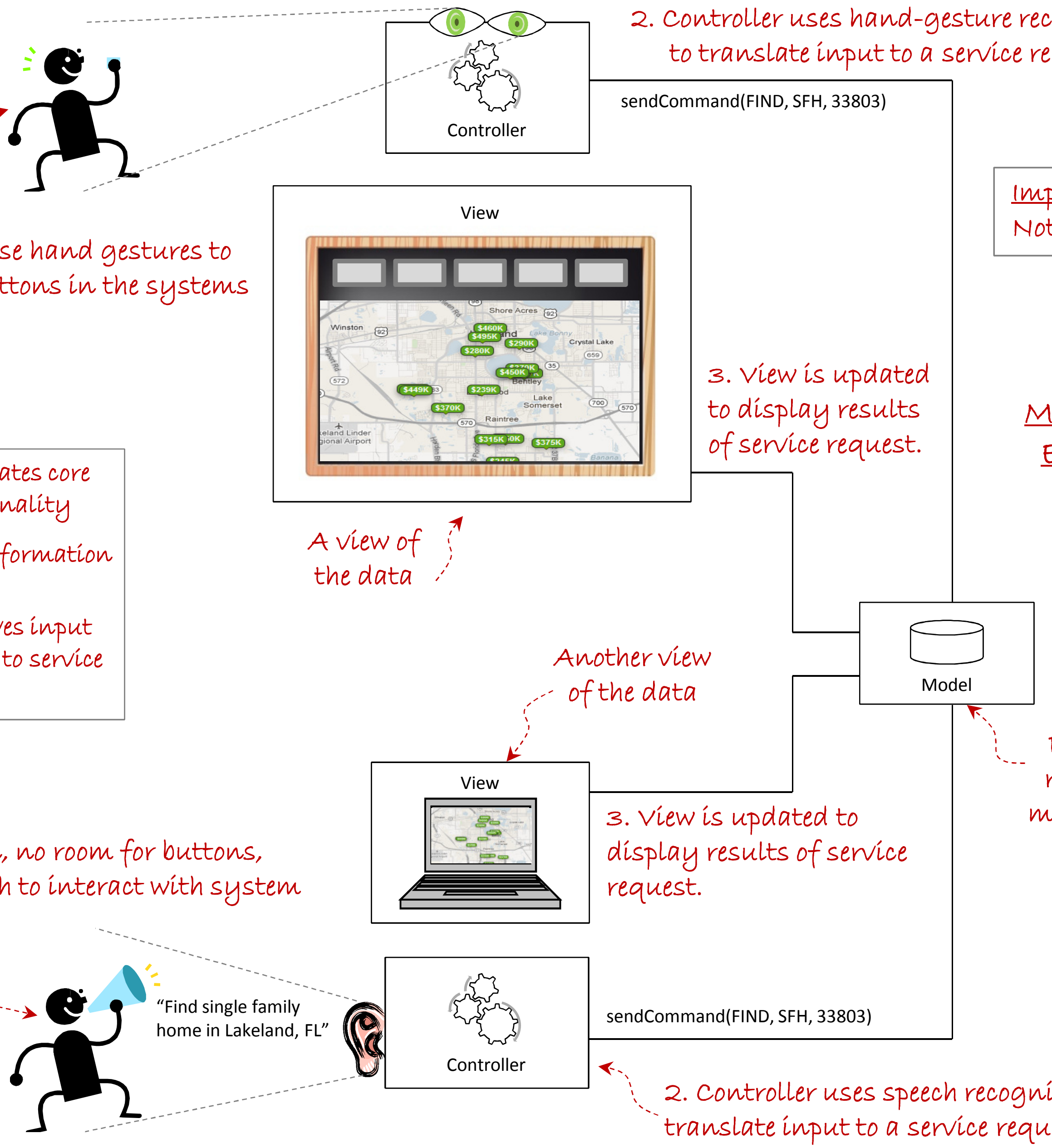
MVC for the Real Estate System

Important:
Not a UML diagram!

Model: Encapsulates core data and functionality

View: Display information to the user

Controller: Receives input and translates it to service requests.



sendCommand(FIND, SFH, 33803)

"Find single family home in Lakeland, FL"

sendCommand(FIND, SFH, 33803)

1. Very large screen, Use hand gestures to interact with control buttons in the systems

2. Controller uses hand-gesture recognition to translate input to a service request

3. View is updated to display results of service request.

A view of the data

Another view of the data

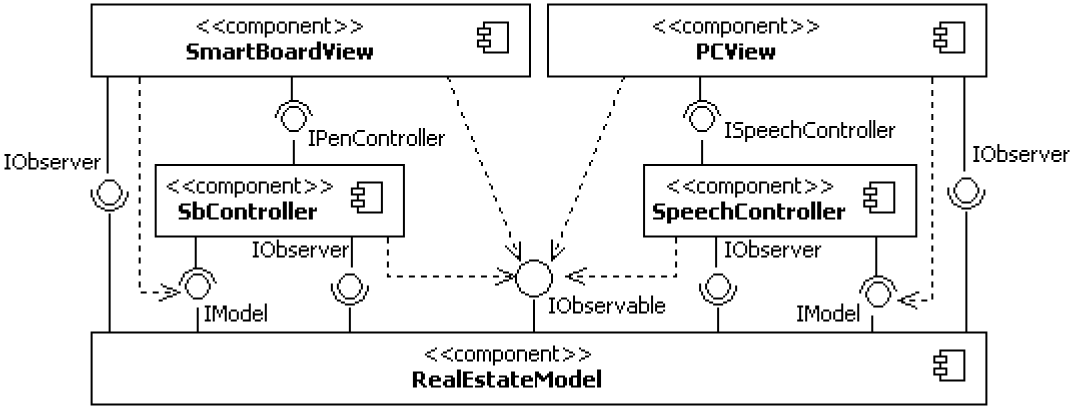
Real estate data resides here. The model encapsulates core data and functionality

1. Small screen, no room for buttons, therefore, use speech to interact with system

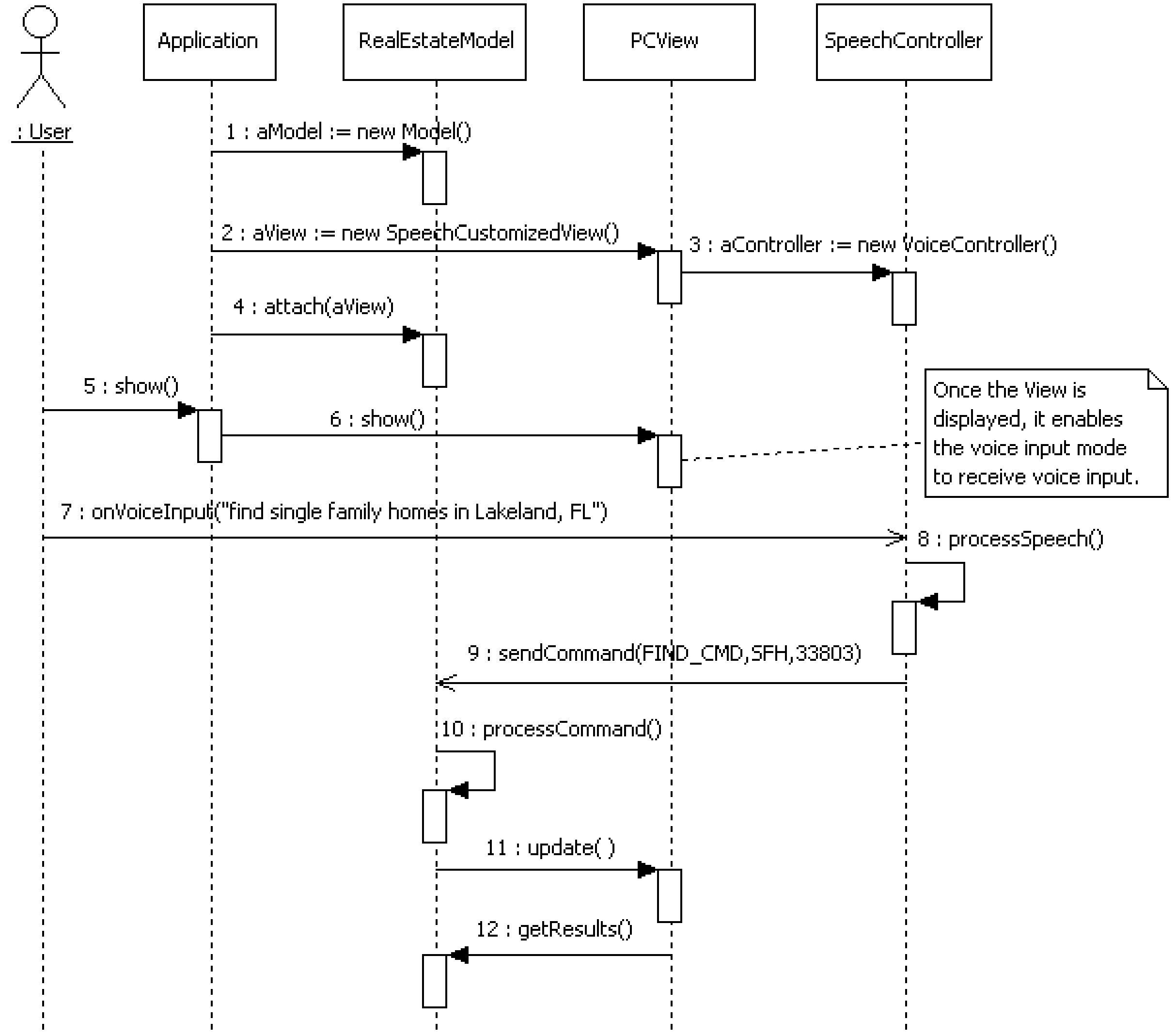
2. Controller uses speech recognition to translate input to a service request

3. View is updated to display results of service request.

MODEL-VIEW-CONTROLLER ARCHITECTURAL PATTERN



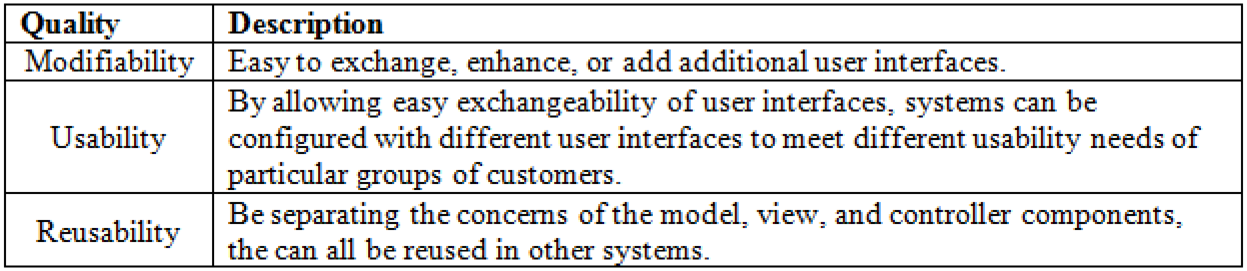
MODEL-VIEW-CONTROLLER ARCHITECTURAL PATTERN


Application
RealEstateModel
PCView
SpeechController
: User
1 : aModel := new Model()
2 : aView := new SpeechCustomizedView()
3 : aController := new VoiceController()
4 : attach(aView)
5 : show()
6 : show()
7 : onVoiceInput("find single family homes in Lakeland, FL")
8 : processSpeech()
9 : sendCommand(FIND_CMD,SFH,33803)
10 : processCommand()
11 : update( )
12 : getResults()

Once the View is displayed, it enables the voice input mode to receive voice input.

MODEL-VIEW-CONTROLLER ARCHITECTURAL PATTERN

- Quality properties of the MVC architectural pattern include the ones specified below.



Quality	Description
Modifiability	Easy to exchange, enhance, or add additional user interfaces.
Usability	By allowing easy exchangeability of user interfaces, systems can be configured with different user interfaces to meet different usability needs of particular groups of customers.
Reusability	Be separating the concerns of the model, view, and controller components, the can all be reused in other systems.

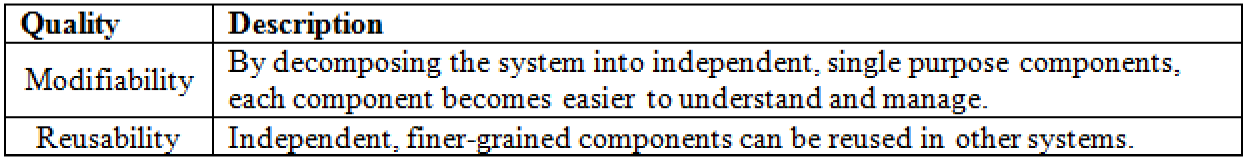
- There are many variations of the MVC architectural pattern.
  - ✓ One popular variation includes the fusion of views and controller components, as made famous in the 1990s by Microsoft's Document-View architecture
  - ✓ Other more extensive variations include the process-abstraction-controller pattern.

- MVC has been successfully adapted as an architecture for the World Wide Web, e.g., see:
  - ✓ JavaScriptMVC
  - ✓ Backbone

HIERARCHICAL SYSTEMS

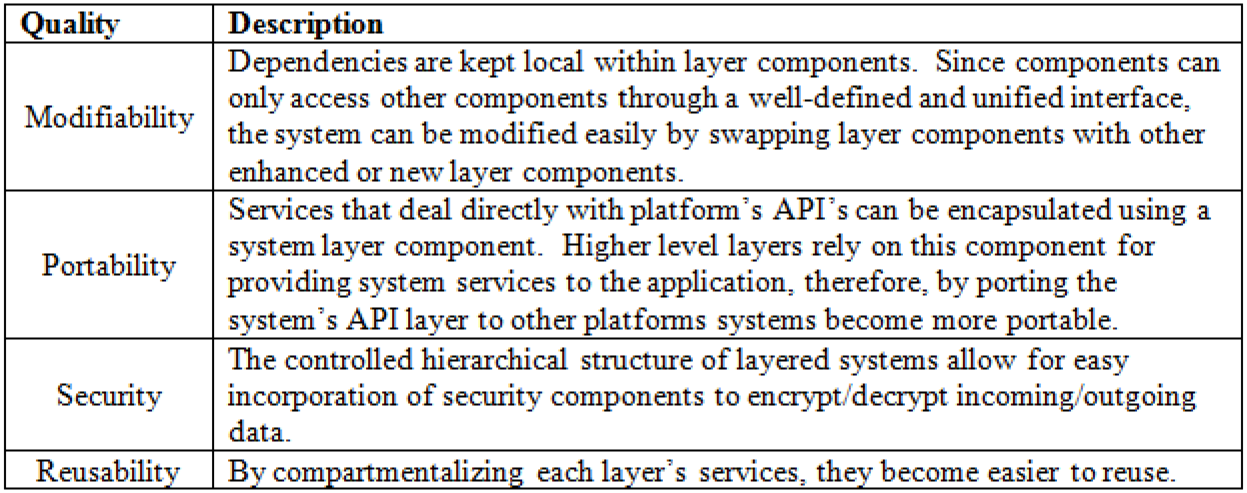
- Hierarchical systems can be decomposed and structured in hierarchical fashion. Two common architectural patterns for hierarchical systems are:
  - ✓ Main program and subroutine
  - ✓ Layered

- Quality properties of the Main Program and Subroutine architectural pattern include:



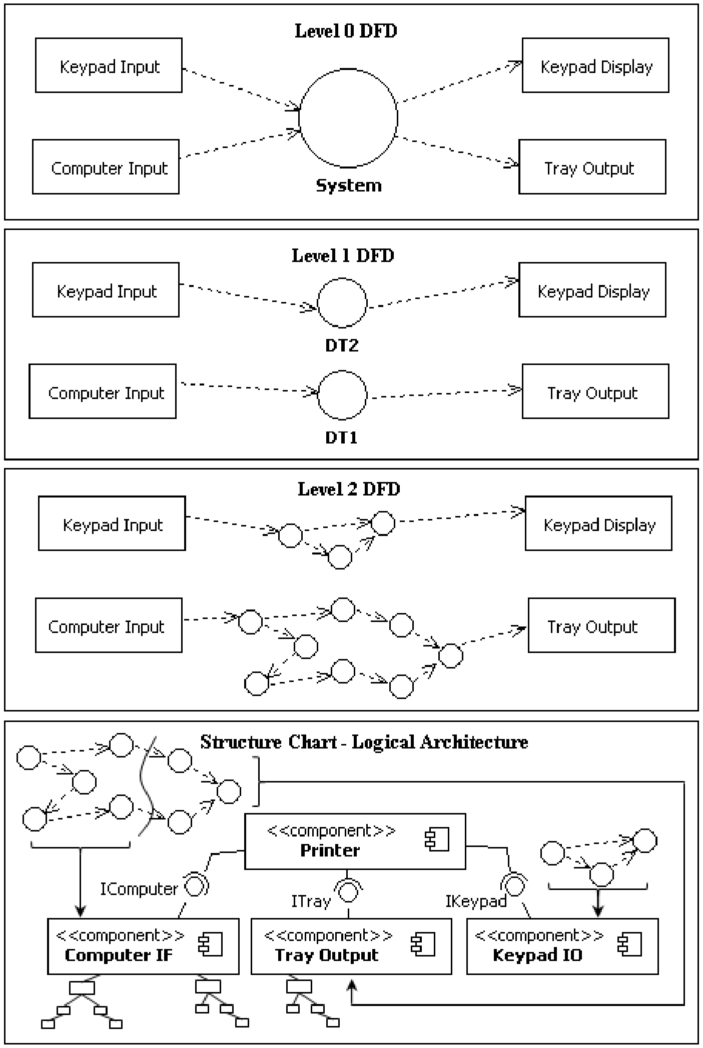
Quality	Description
Modifiability	By decomposing the system into independent, single purpose components, each component becomes easier to understand and manage.
Reusability	Independent, finer-grained components can be reused in other systems.

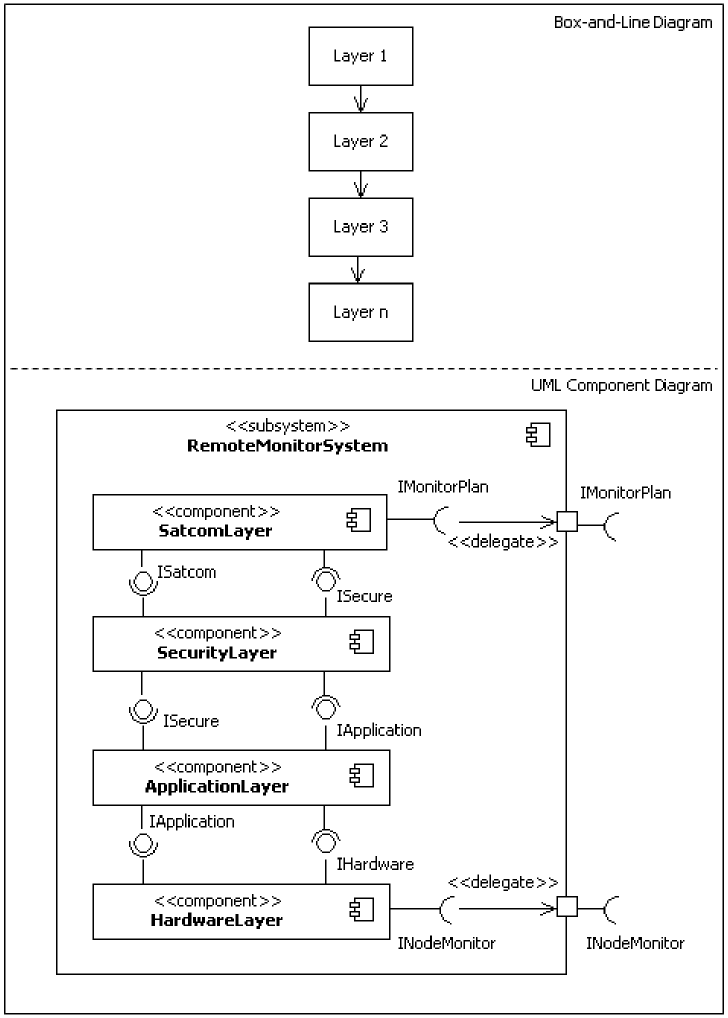
- Quality properties of the Layered architectural pattern include the ones specified below.



Quality	Description
Modifiability	Dependencies are kept local within layer components. Since components can only access other components through a well-defined and unified interface, the system can be modified easily by swapping layer components with other enhanced or new layer components.
Portability	Services that deal directly with platform's API's can be encapsulated using a system layer component. Higher level layers rely on this component for providing system services to the application, therefore, by porting the system's API layer to other platforms systems become more portable.
Security	The controlled hierarchical structure of layered systems allow for easy incorporation of security components to encrypt/decrypt incoming/outgoing data.
Reusability	By compartmentalizing each layer's services, they become easier to reuse.

MAIN PROGRAM AND SUBROUTINE AND LAYERED PATTERNS





WHAT'S NEXT…

- In this session, we continued the discussion on distributed systems and presented fundamentals concepts of interactive and hierarchical systems, together with essential architectural patterns for these systems, including:
    - ✓ Broker
    - ✓ MVC
    - ✓ Layered
    - ✓ Main Program and Subroutine

- This finalizes our coverage of architectural patterns. In the next module, we start the discussion on detailed design, which is the next activity in the design process. Detailed design begins once the architecture of the software is sufficiently complete.