

# CptS 322- Software Engineering Principles I

## Software Design and Architecture Part-2

**Instructor: Sakire Arslan Ay**  
**Fall 2023**

# Architectural Patterns

- The notion of patterns can be applied to software architecture.
  - These are called *architectural patterns* or *architectural styles*.
  - Each allows you to design flexible systems using components
    - The components are as independent of each other as possible.

# Architectural Patterns

1.

2.

3.

4.

5.

# Architectural Patterns

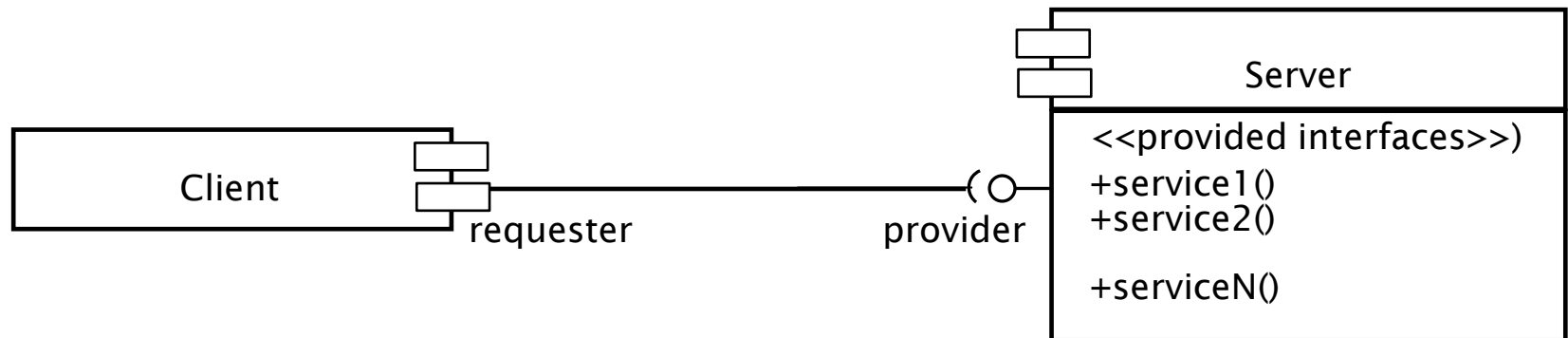
1. The Client-Server Architectural Pattern
2. Peer-to-Peer Architectural Pattern
3. Model-View-Controller (MVC) Architecture
4. The Multi-Layer Architectural Pattern
5. The Pipe-and-Filter Architectural Pattern

# The Client-Server Architectural Pattern

- There is at least one subsystem that has the role of **server**, waiting for and then handling connections.
- There is at least one subsystem that has the role of **client**, initiating connections in order to obtain some service.

# The Client-Server Architectural Pattern

- Each client calls on the server, which performs some service and returns the result
  - The clients know the interface of the server
  - The server does not need to know the interface of the client
- The response in general is immediate
- End users interact only with the client.



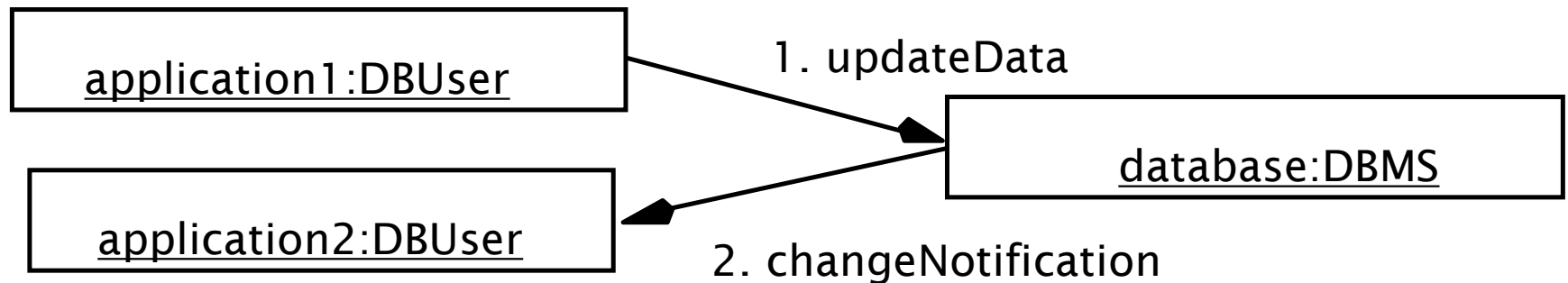
Client/server Architectural Design (UML Component Diagram)

# The Client-Server Architectural Pattern

- Often used in the design of database systems
  - Front-end: User application (client)
  - Back-end: Database access and manipulation (server)
  - Functions performed by client:
    - Input from the user (Customized user interface)
    - Front-end processing of input data
  - Functions performed by the database server:
    - Centralized data management
    - Data integrity and database consistency
    - Database security

# Limitation of Client/Server Architectures

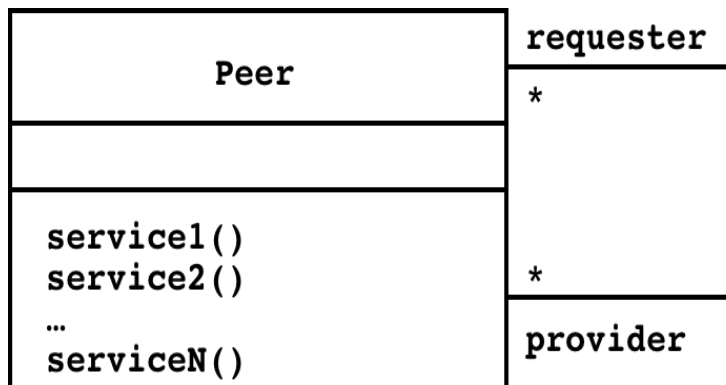
- Client/Server systems do not provide peer-to-peer communication
- Example:
  - Database must process queries from application and should be able to send notifications to the application when data have changed





# Peer-to-Peer Architectural Pattern

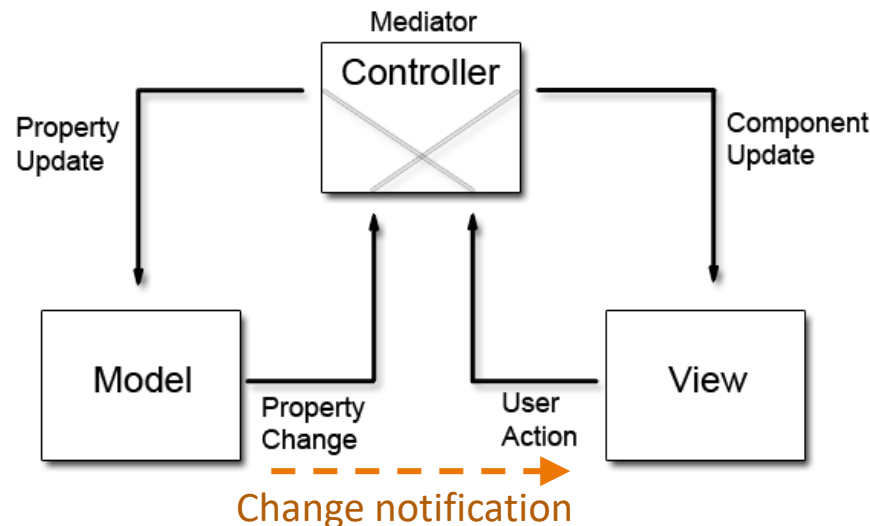
- Generalization of Client/Server Architectural Pattern
  - Introduction a new abstraction: Peer
  - “Clients and servers can be both peers”



“A peer can be a client as well as a server”.

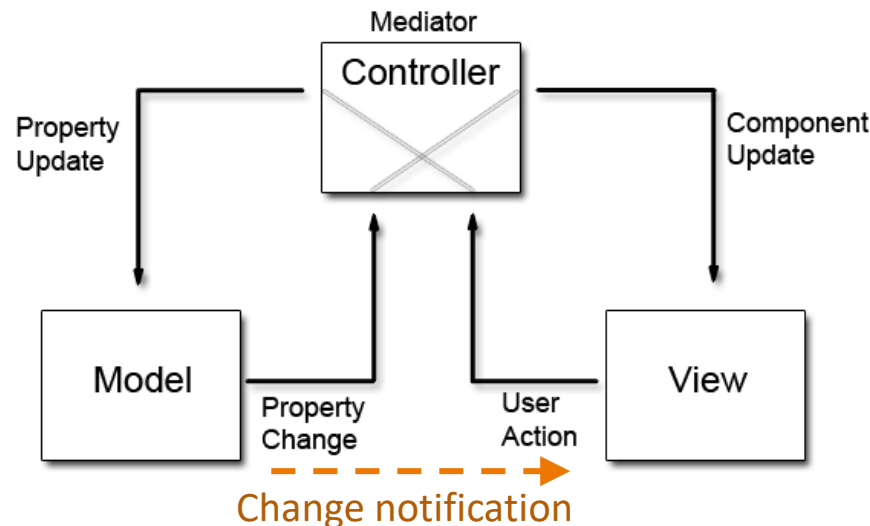
# Model-View-Controller (MVC) Architecture

- Many applications follow this pattern:
  - iOS, Android, Web, Desktop applications
    - Receive commands/instructions
    - Read/compute/update state
    - Pick/update the presentation of the state
- A time-tested architectural principle:
  - Separate model (data and state management) from presentation (view)



# Why MVC?

- The presentation and the data logic have different rate/risk for change
  - Isolate change
- Use different presentations for the same state
  - E.g., Google Drive presentations: browser, mobile, API
- Presentation is hard to test automatically
  - We want to at least test the state management easily

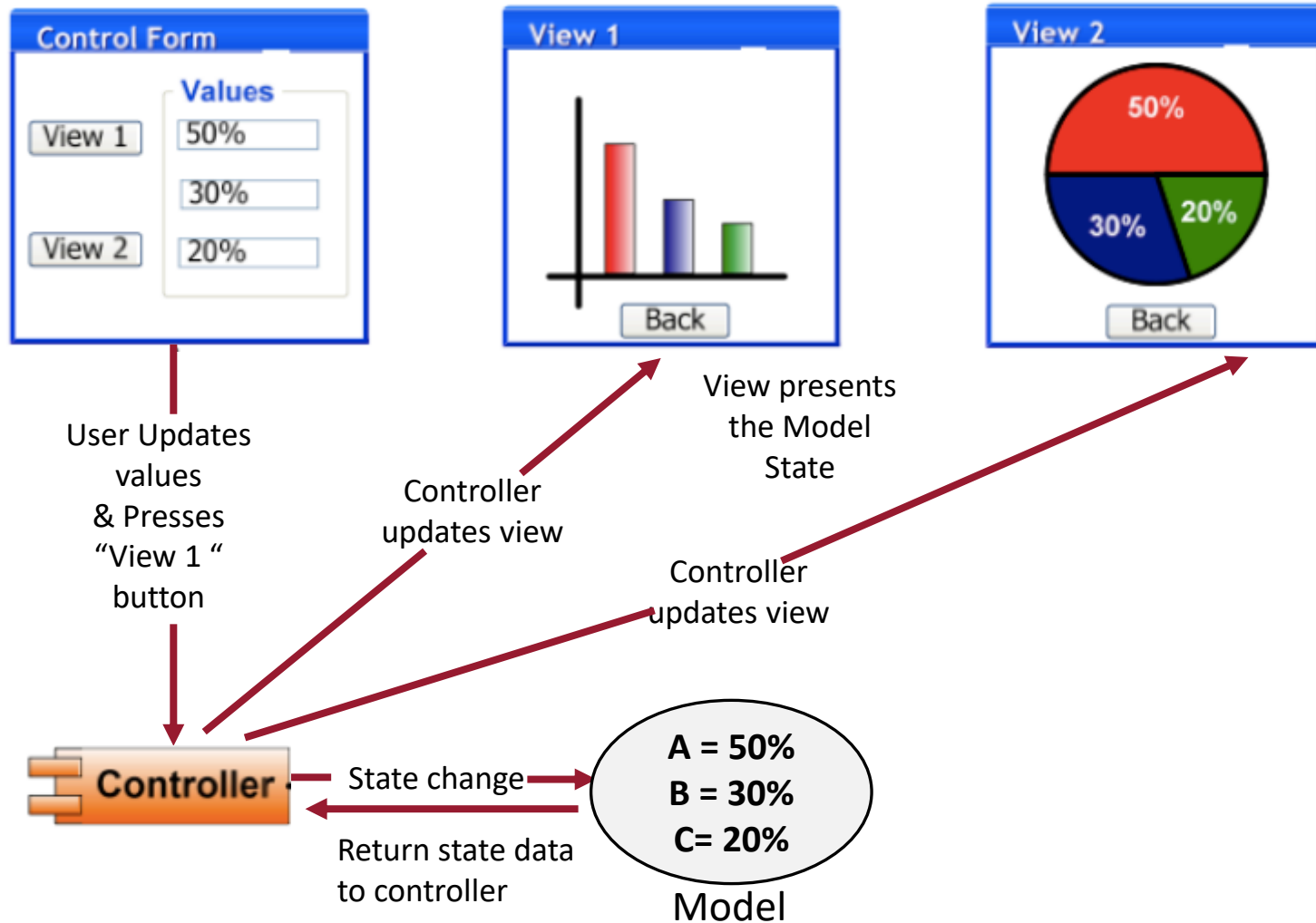


# Model-View-Controller (MVC)

## Architectural Pattern

- An architectural pattern used to help separate the user interface from other parts of the system.
  - *Model* :
    - The state and data
    - Methods for accessing and modifying state and data
  - *View* :
    - Render the appearance of the state and data from the model in the user interface
    - When model changes, view must be updated
  - *Controller* :
    - Translates user actions (i.e. interactions with view) into operations on the model
    - Example user actions: button clicks, menu selections
    - When model changes controller should update the view

# Example of MVC for GUI App



# MVC Example

## Step 1: Create the Model

```
public class MyData {  
    private float A;  
    private float B;  
    private float C;  
  
    public float getA() {  
        return A;  
    }  
    public float setA(float A) {  
        this.A = A;  
    }  
    public float getB() {  
        return B;  
    }  
    public float setB(float B) {  
        this.B = B;  
    }  
    public float getC() {  
        return C;  
    }  
    public float setC(float A) {  
        this.C = C;  
    }  
}
```

## Step 2: Create the View

```
public class MyDataView {  
  
    public void displayBarGraph(float A, float B, float C) {  
        g = barGraph(A,B,C);  
    }  
  
    public void displayPieGraph(float A, float B, float C) {  
        g = pieGraph(A,B,C);  
    }  
  
    public void printData(float A, float B, float C) {  
        System.out.println("A: %f" + A);  
        System.out.println("B: %f" + B);  
        System.out.println("C: %f" + C);  
    }  
}
```

### Step 3: Create the Controller

```
public class MyDataController {
    private MyData model;
    private MyDataView view;

    public MyDataController( MyData model, MyDataView view){
        this.model = model;
        this.view = view;
    }
    public void setDataA(float A){
        model.setA(A);
        updateView("bar");
    }
    public void setDataB(float B){
        model.setB(B);
        updateView("bar");
    }
    public void setDataC(float C){
        model.setC(C);
        updateView("bar");
    }
    public float getDataA(){
        return model.getA();
    }
    public float getDataB(){
        return model.getB();
    }
    public float getDataC(){
        return model.getC();
    }
    public void updateView(String type){
        if (type=="bar") view.displayBarGraph(model.getA(), model.getB(), model.getC());
        else if (type=="pie") view.displayPieGraph(model.getA(), model.getB(), model.getC());
        else view.printData(model.getA(), model.getB(), model.getC());
    }
}
```



## Step 4: Create the main program

```
public class MVCPatternDemo {  
    public static void main(String[] args) {  
  
        //Create the model  
        MyData model = new MyData();  
        model.setA(0.5);  
        model.setB(0.3);  
        model.setC(0.2);  
  
        //Create a view : to display data on graph  
        MyDataView view = new MyDataView();  
  
        MyDataController controller = new MyDataController(model, view);  
  
        controller.updateView("bar");  
  
        //update model data  
        controller.setA(0.6);  
        controller.updateView("pie");  
    }  
}
```

# The Model

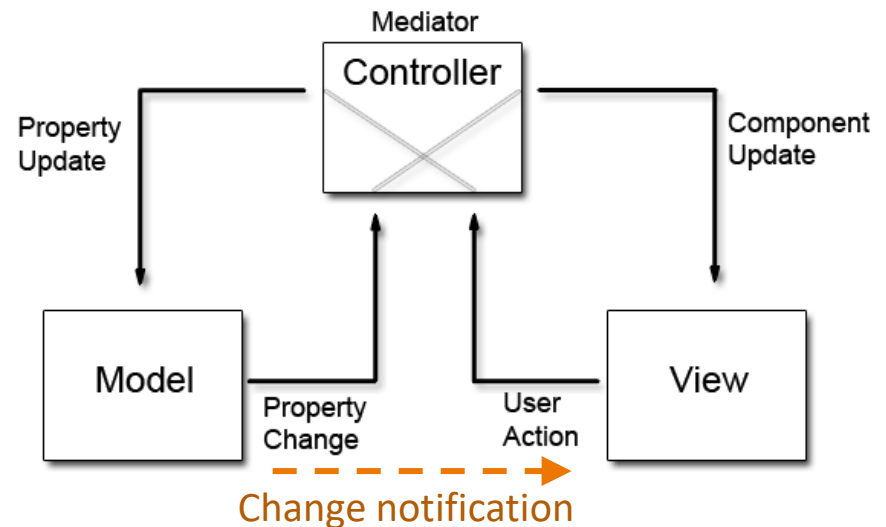
- The main goal of MVC is to separate the model from the presentation (view)
- Model is the functional core of the application:
  - Sometimes called business logic because this is the business layer in enterprise applications
  - Sometimes called domain logic
  - Encapsulates the state of the application
    - E.g., databases and data structures would be managed by the model
- Allows the controller to access application functionality encapsulated by the Model
- **Notifies views when the application state changes**

# The View

- Renders the contents of a model
- Forwards user gestures to the controller

MVC variants:

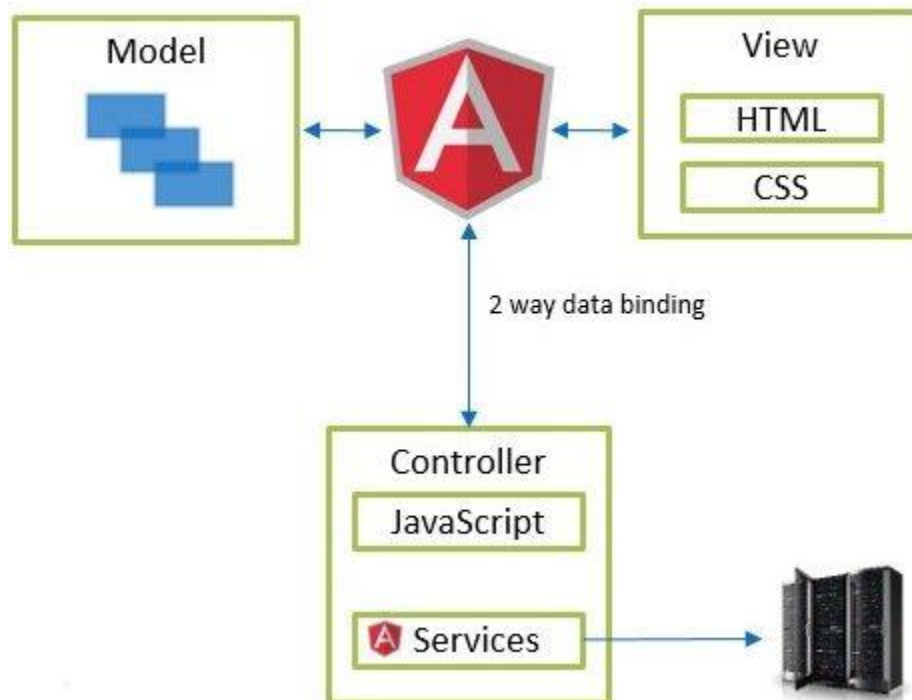
- When the model changes, the view must update its presentation
  - push model:
    - the view registers itself with the model for change notifications
  - pull model
    - the view is responsible for calling the model when it needs to retrieve the most current data



# The Controller

- Orchestrates interaction of models and views
- Defines application behavior
- Interprets user gestures and maps them into actions
  - For the model to perform
  - In selecting a different view
    - E.g., a web page of results to present back to the user

# Example of MVC for Angular JS

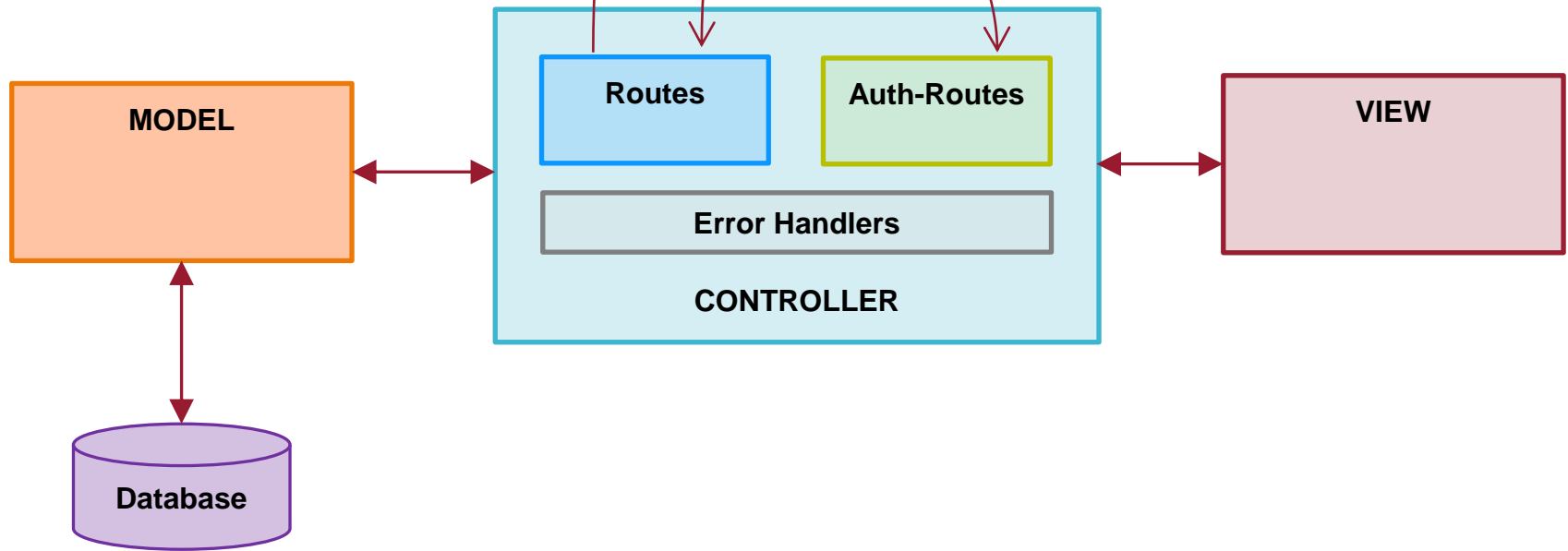
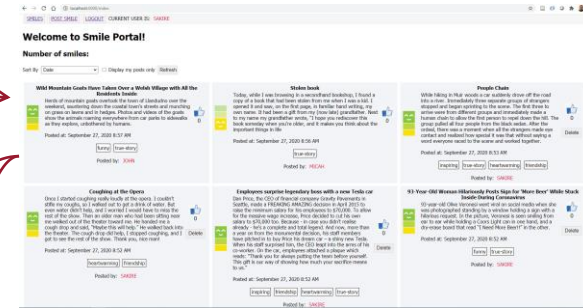


**2-way binding** – Angular.js keeps the data and presentation layer in sync.

- No need to write additional JavaScript code to keep the data in your HTML code and your data layer in sync. Angular.js will automatically do this for you.
- You just need to specify which control is bound to which part of your model.

- The Structure of the “Smile App”

# Smile Application



<https://selftaughtcoders.com/model-view-controller-mvc-web-application/>

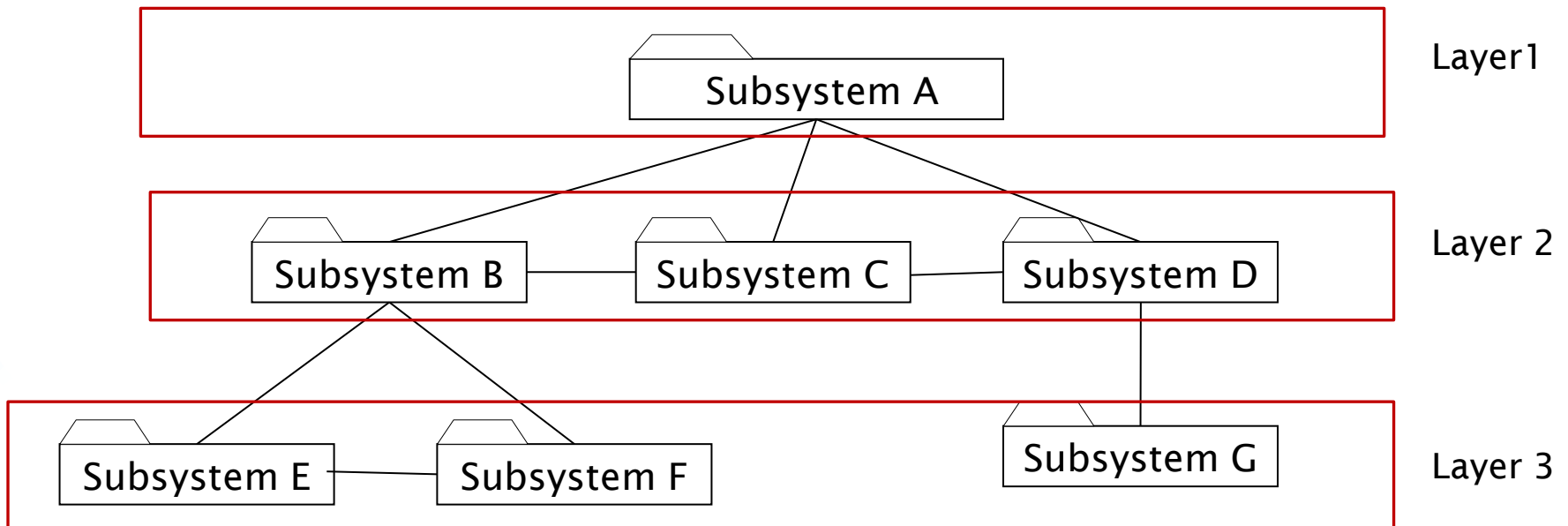
# Smile Application – UML Component Diagram

- We will draw this in class.



# The Multi-Layer Architectural Pattern

- Hierarchical decomposition of the system as an ordered set of **layers**.
- A **layer** is a subsystem that provides services to another layer:
  - A layer only depends on services from lower layers
  - A layer has no knowledge of higher layers

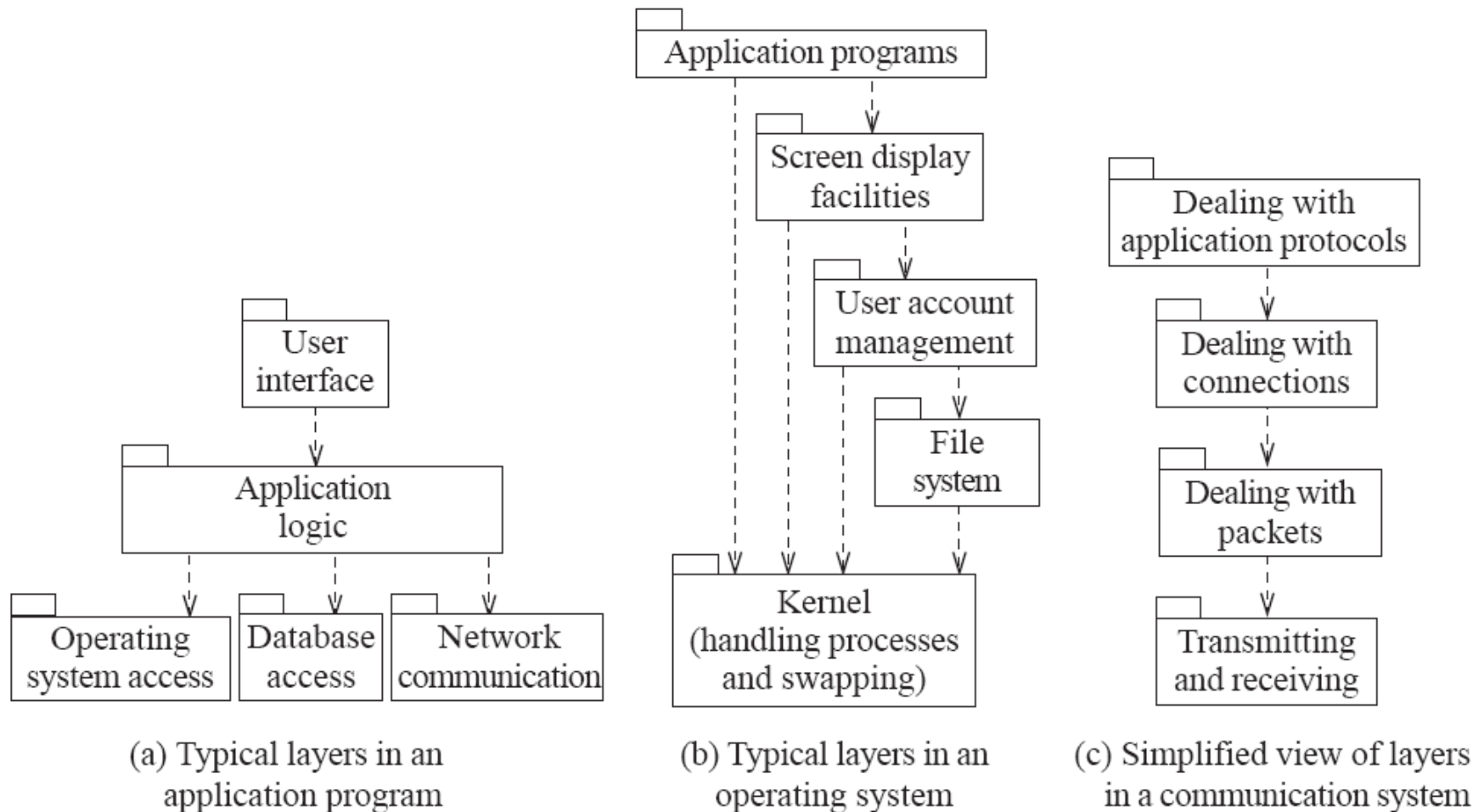




# The Multi-Layer Architectural Pattern

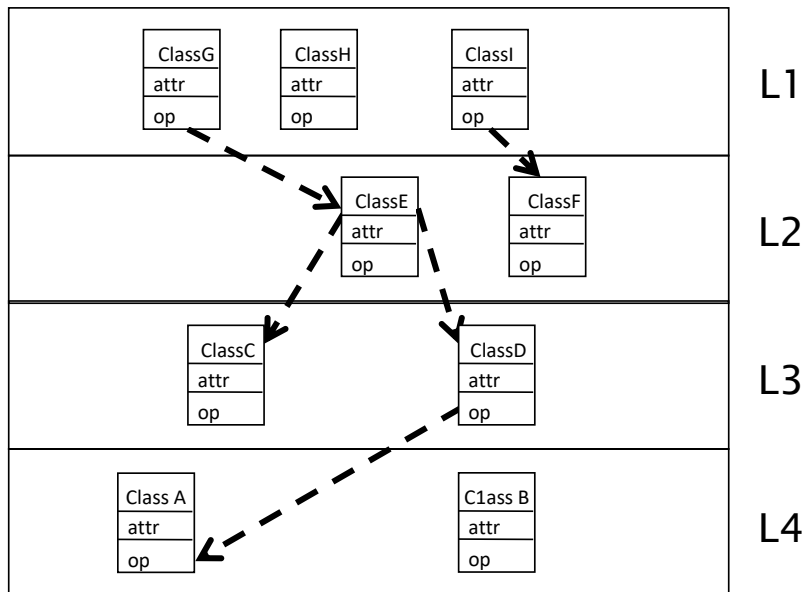
- Each layer has a well-defined interface used by the layers above.
  - The higher layers see the lower layers as a set of *services*.
- A complex system can be built by superposing layers at increasing levels of abstraction.
  - It is important to have a separate layer for the UI.
  - Layers immediately below the UI layer provide the application functions determined by the use-cases.
  - Bottom layers provide general services.
    - e.g. network communication, database access

# Example of Multi-layer Systems

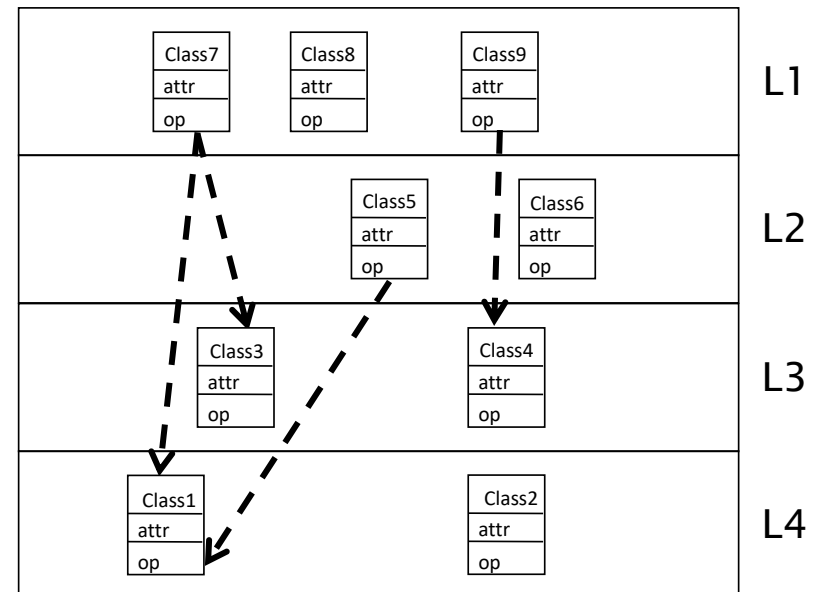


# Open vs Closed Layered Architecture

- **Closed architecture:** each layer communicates only with the layer immediately below it.
  - Design Goals: Maintainability, flexibility
- **Open architecture:** a layer can also access layers at deeper levels.
  - Design Goal: Runtime efficiency



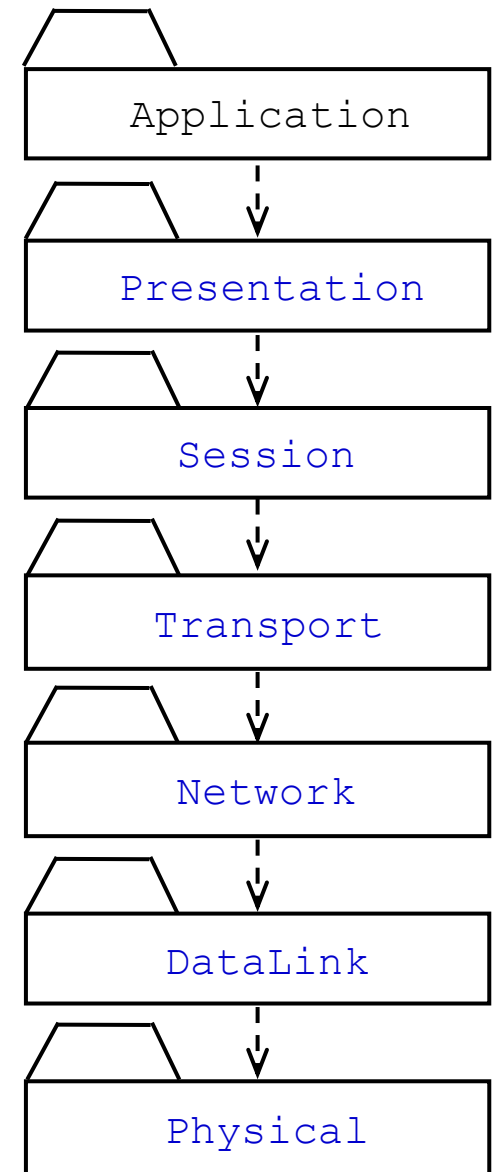
Closed Layered Architecture



Open Layered Architecture

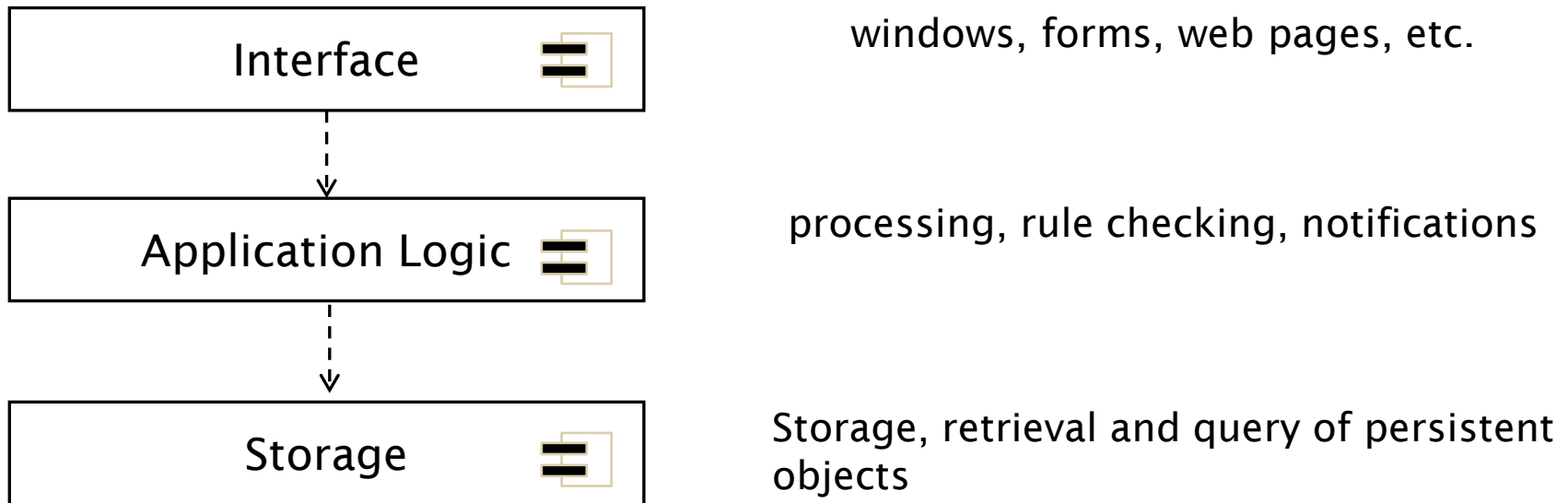
# Example of Closed Layered Architecture: OSI Model Layers and their Services

- Presentation Layer
  - Services: data transformation (encryption, byte swapping)
- Session Layer
  - Services: Initializing and authenticating a connection
- Transport layer
  - Services: Transmitting messages
- Network layer
  - Services: Transmit and route data within the network
- Datalink layer
  - Services: Transmit data frames without error
- Physical layer
  - Services: Transmit bits over communication channel



# Three-Tier Architectural Pattern

- An application consists of 3 hierarchically ordered subsystems
  - Interface layer: user interface
  - Application logic layer: middleware
  - Storage layer: database system



Usually the 3 layers are allocated on 3 separate hardware nodes

# Example of a Three-tier Architecture

- Three-tier Architectural pattern is often used for the development of Websites:
  1. The **Web Browser** implements the user interface
  2. The **Web Server** serves requests from the web browser
  3. The **Database** manages and provides access to the persistent data.

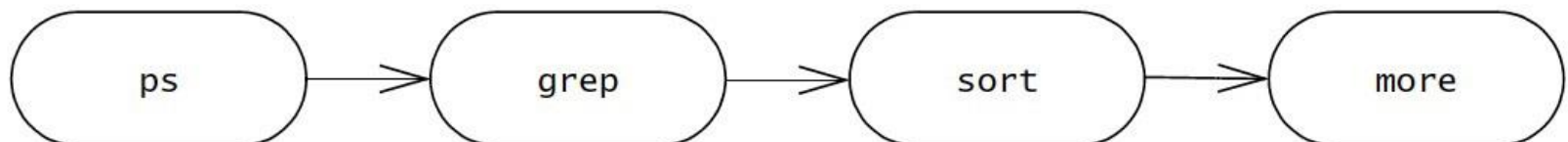


# The Pipe-and-Filter Architectural Pattern

- A **pipeline** consists of a chain of processing elements (processes, threads, etc.), arranged so that the output of one element is the input to the next element
- A stream of data, in a relatively simple format, is passed through these series of processes
- Each of which transforms it in some way.
- Data is constantly fed into the pipeline.
- The processes work concurrently.
- Example: Unix shell command

```
% ps auxwww | grep dutoit | sort | more
```

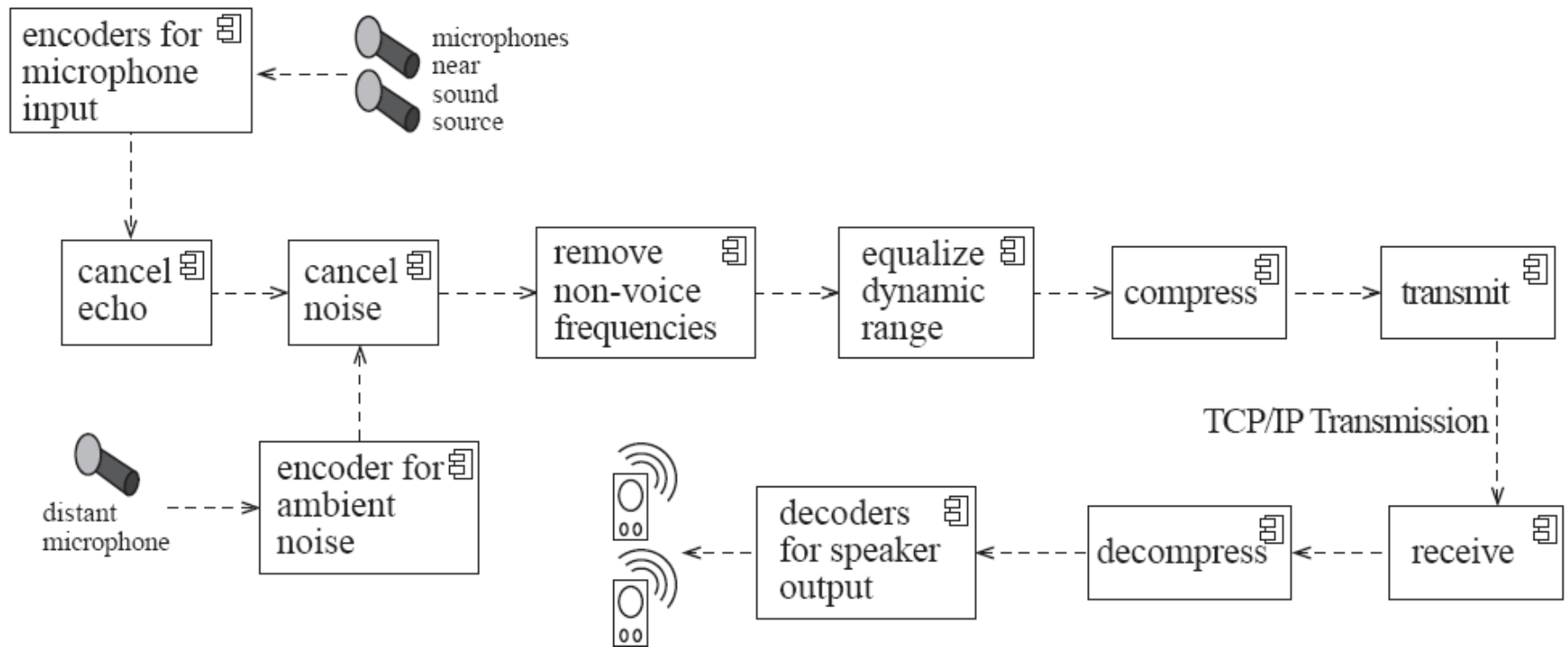
```
dutoit 19737 0.2 1.6 1908 1500 pts/6 0 15:24:36 0:00 -tcsh
dutoit 19858 0.2 0.7 816 580 pts/6 S 15:38:46 0:00 grep dutoit
dutoit 19859 0.2 0.6 812 540 pts/6 0 15:38:47 0:00 sort
```



# The Pipe-and-Filter Architectural Pattern

- The architecture consists of subsystems (called pipes) and filters
  - **Filter**: A subsystem that does a processing step
  - **Pipe**: A Pipe is a connection between two processing steps
- Each filter has an input pipe and an output pipe.
- The data from the input pipe are processed by the filter and then moved to the output pipe
- The architecture is very flexible.
  - Almost all the components could be removed.
    - Components could be replaced.
    - New components could be inserted.
    - Certain components could be reordered.

# Example of a Pipe-and-Filter System



# Contents of a Good Architectural Model

- A system's architecture will often be expressed in terms of several different *views*
  - The logical breakdown into subsystems/classes ; The interfaces among the subsystems/classes
    - UML component diagram
    - UML class diagram
  - The dynamics of the interaction among components at run time
    - UML sequence diagram
    - UML state diagram
  - The data that will be shared among the subsystems
    - UML ER diagram
  - The components that will exist at run time, and the machines or devices on which they will be located
    - UML deployment diagram

# Summary

- An architecture provides a highlevel framework to build and evolve a software system.
- Strive for modularity: strong cohesion and loose coupling.
- Consider using existing architectural styles or patterns.