Architectural Patterns

Architectural Patterns

‘Notion’ of an <u>architectural</u> <u>pattern</u> is essential to good architectural design.

- We call these ***architectural patterns*** or ***architectural styles***.
- **Patterns** provide for flexible systems using <u>***components***</u>
  - <u>Components</u> are as <u>independent</u> as possible. (Hunks of executable components…)
- <u>Some architectural patterns are better</u> **far better** for some applications than for others.
- Let’s look at a few important ones.

1. The Multi-Layer Architectural Pattern

Layered architecture:

—layers communicate down!

—Normally immediately below with few ‘skips’

—Is the classical approach.

—The higher layer sees the lower layers as a set of ***services***.

—This notion is **fundamental to good design.**

—Often, a layer communicates **ONLY** with the layer below it
not always - but normally.

.

Multi-Layered Architectural Pattern…

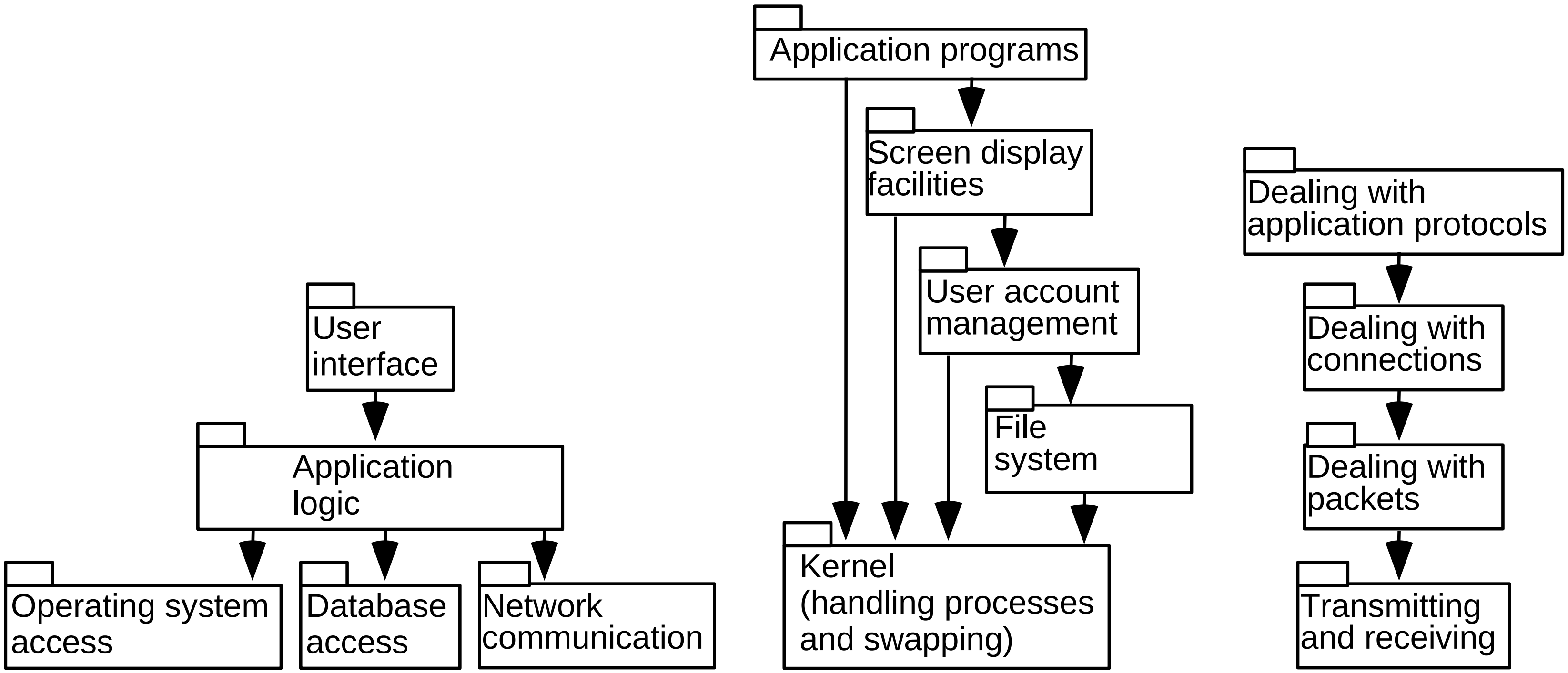
- Built with layers at **increasing levels of abstraction.**
  - 1. **User Interface layer -** normally first for **presentation**
  - 2. **Application Layer is usually** <u>immediately below</u> UI layer and **typically provides** the **<u>application functions</u>** determined by application use-cases. (application layer)
  - 3. **Domain Layer is usually next and** provides **general domain-level services** (business use-cases)
  - 4. **Services / Support** (**Bottom) layers** provide **general (but essential) services**.
    - » e.g. network communication, database access
    - » operating system services

Extremely Nice Feature of Layered Design

- Layers / layer services are <u>**replaceable**</u>
  - NO impacting to other layers and dependencies

    <u>**if**</u> the <u>**interfaces**</u> remains unchanged.
  - Examples:
    - » User Interface layer when porting to a different platform or for different environments.
    - » Upgrading / enhancing / optimizing services…
  - We have **clear separation of concerns**
  - We have very good '**cohesion**' of services…

Example Of Multi-layer Systems (Seen Before…)



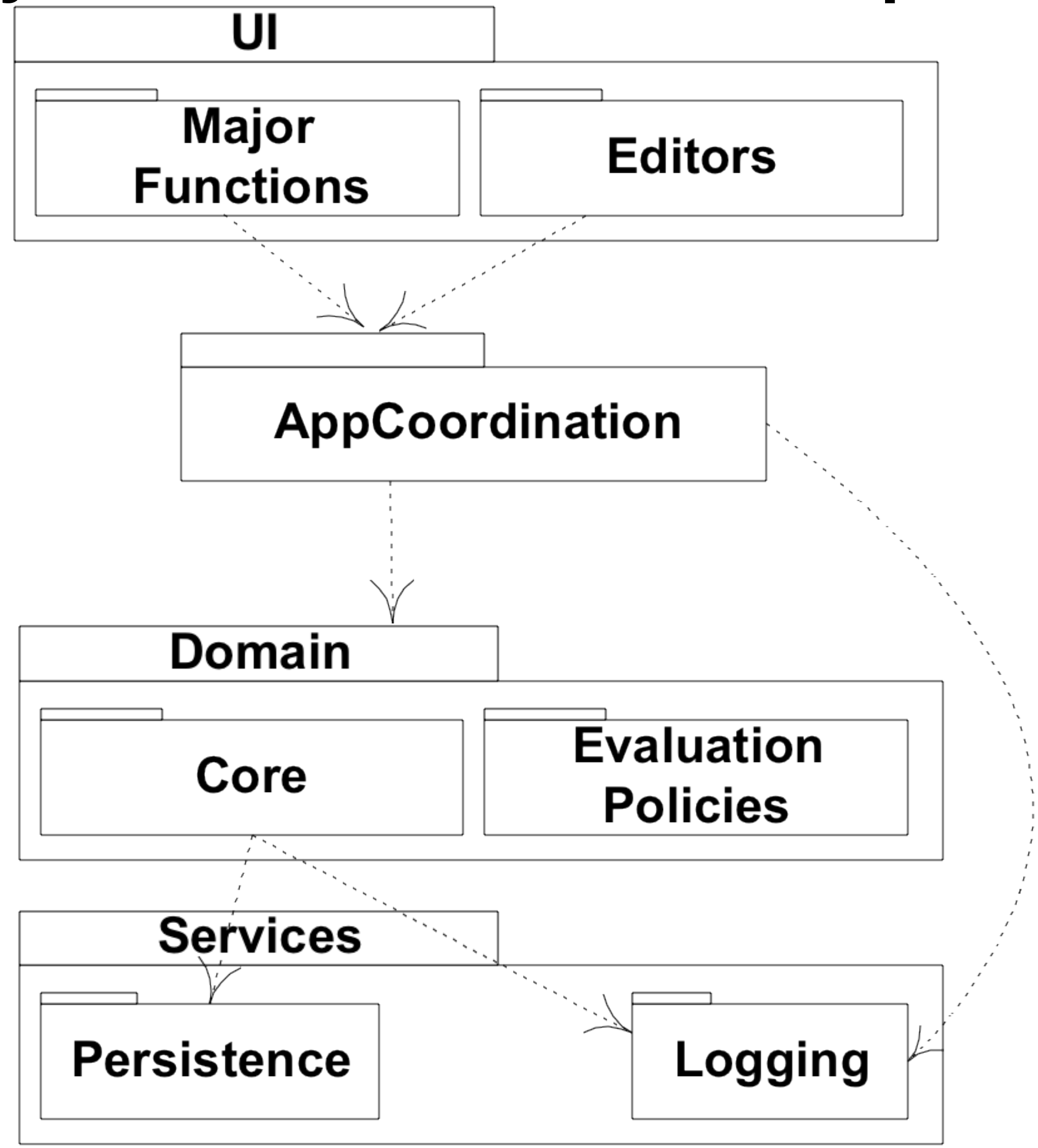
a) Typical layers in an application program

b) Typical layers in an operating system

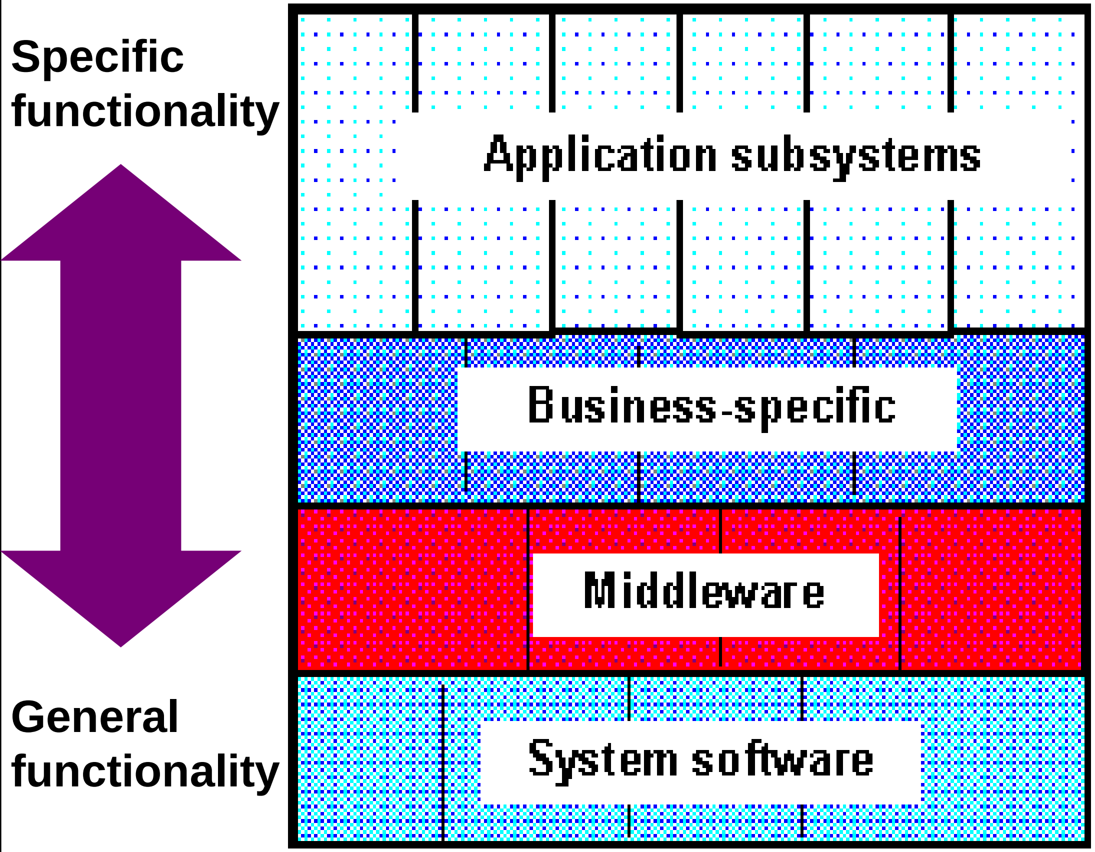
c) Simplified view of layers in a communication system

è Communications between layers: usually use procedure calls.
Upper layers become clients; lower layers become servers.

Multi-Tier Layered Architecture - Example



Multi-Tier Layered Architecture - Example Layering Approach



Distinct application subsystem that make up an application - contains the value adding software developed by the organization.

Business specific - contains a number of reusable sybsystems specific to the type of business.

Middleware - offers subsystems for utility classes and platform-independent services for distributed object computing in heterogeneous environments and so on.

System software - contains the software for the actual infrastructure such as operating systems, interfaces to specific hardware, device drivers and so on.

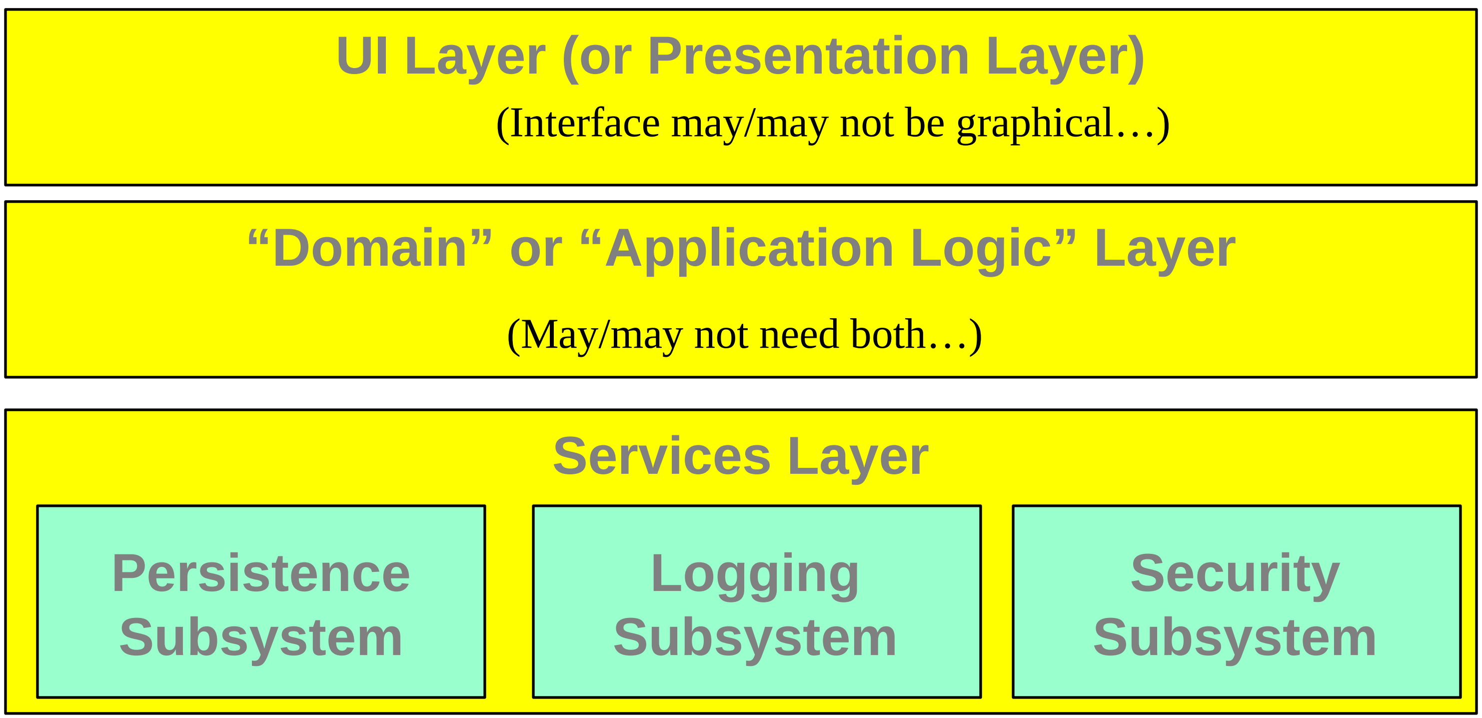
This is a <u>very broad generalization</u>. in practice, things will be considerably different and <u>application dependent</u> in many cases.

Note: this is also a very <u>general view</u>; may/may not include a GUI layer.

Multi-Tier Layered Architecture - Example

<u>Separate</u> presentation and application logic, and other areas of concern.

Consider: Different names (in some cases). Can see the main idea!



Multi-layered Architecture And Design Principle Satisfy Eleven Architectural Design Principles)

1. *Divide and conquer*: layers can be independently designed.

2. *Increase cohesion*: Well-designed layers have layer cohesion.

3. *Reduce coupling*: Well-designed lower layers **do not know about the higher layers** and the only connection between layers is through the API.

4. *Increase abstraction*: you **do not need to know** the details of how the lower layers are implemented.

5. *Increase reusability*: The lower layers can often be designed generically. (e.g. those that handle database access, persistency, etc.) (different databases….)

Multi-layered Architecture And Design Principle Satisfy Eleven Architectural Design Principles)*

6. *Increase reuse*: You can often **reuse layers** built by others that provide the services you need. (think: Domain Layer)

7. Increase flexibility: you can **add** new facilities built on lower-level services, or replace higher-level layers.

8. *Anticipate obsolescence*: By isolating components in separate layers, the <u>system becomes more resistant to obsolescence</u>.

9. *Design for portability*: All the <u>**dependent** facilities can be isolated</u> in one of the lower layers.

As we know, some things tend to change over time; more than some other aspects of an application.

10. *Design for testability*: Layers can be tested independently through the interfaces exercising layer responsibilities.

11. *Design defensively*: The APIs of layers are natural places to build in rigorous **assertion-checking**.

2. Client-server And Other Distributed Architectural Patterns

- At least **one** component has the role of ***server:***
  - —waiting for and then handling connections.
- At least **one** component that has the role of ***client***, initiating connections to obtain some service.

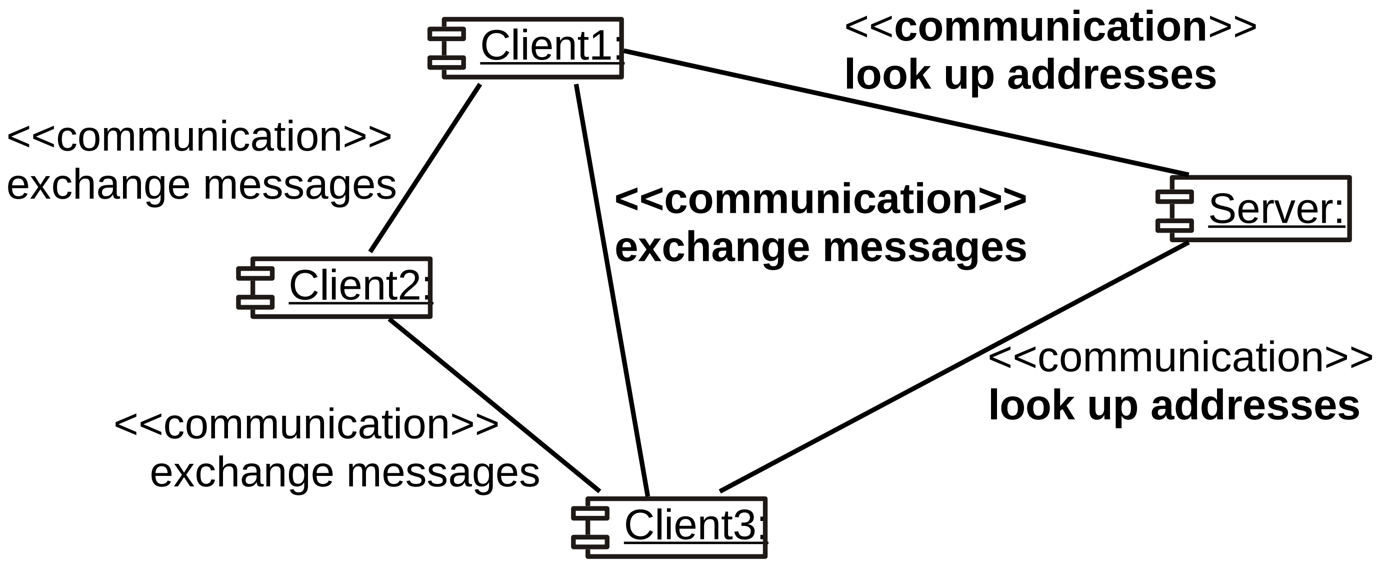
- **Three-tier model** for web-based client-sever applications:
  - —Server 'in the middle'
    - server to client (web-based or not; likely communicating via the Internet)
    - client to a database server (usually / often via an **intranet**)

Other Distributed Client-server Architectural Patterns

- **Peer-to-Peer pattern**. A system where:
  - —various software components **distributed** over several hosts.
  - —Hosts: both clients and servers (to each other)
  - —**Any two components** can set up a communications channel through which communications is accomplished.

- **Variation:**
  - —Sometimes peers need to be able to find each other; need for a server containing **location information**

An Example Of A Distributed System



How Does The Client-server Architectural Pattern Subscribe To Principles Of Good Architectural Design?

1. *Divide and conquer*: Dividing the system into client and server processes is a **strong** way to divide the system.

—Each can be separately developed.

2. *Increase cohesion*: Server can provide **cohesive services** to clients.

3. *Reduce coupling*: There is usually only **one** communication channel exchanging simple messages.

4. *Increase abstraction*: Separate distributed components are often good abstractions. **What does this sentence mean to you?**

6. *Increase reuse*: It is often possible to find suitable **frameworks** on which to build good distributed systems

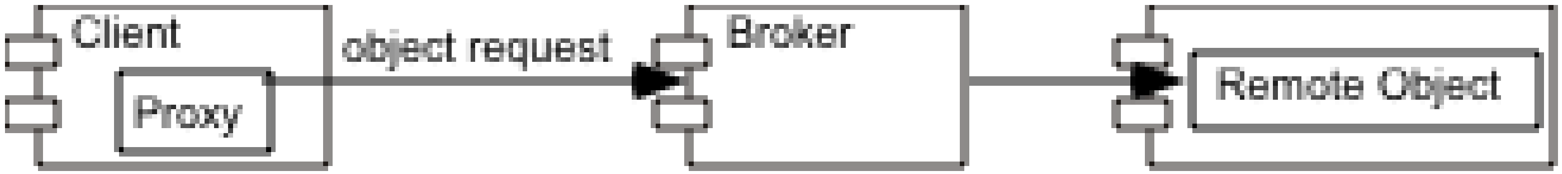
However, client-server systems are often very application specific.

3. The Broker Architectural Pattern

- Here, we **transparently** **distribute** **aspects** of the software system to different nodes
  - —Objects call method's other objects **w/o knowing** object is remotely located.
  - —Client does not '**care**' where the remote object is.
  - —**CORBA:** well-known open standard allowing you to build this kind of architecture.
    - (Common Object Request Broker Architecture)
    - (Microsoft has its own architecture: COM, DCOM (old))

    - **'Proxy design pattern'** can be used such that a **proxy object** calls the broker, which determines where the desired object is located.

Example of a Broker system



Note that all these architectural patterns are illustrated using ‘components.’

Broker Architecture And How This Architectural Design Pattern Subscribes To Design Principles

1. *Divide and conquer*: The remote objects can be independently designed.

5. *Increase reusability*: It is usually possible to **design the remote objects** so that other systems can use them too.

7. *Design for flexibility*: The brokers can be updated as required, **<u>or</u>** the proxy can communicate with a different remote object.

9. *Design for portability*: You can write clients for new platforms while still accessing brokers and remote objects on other platforms.

11. *Design defensively*: You can provide careful assertion checking in the remote objects.

4. Transaction-Processing Architectural Pattern

A process reads a series of inputs one by one.

- Each input describes a *transaction* – a command that typically (for example) might **change** some data stored by the system

- Normally transactions come in **one-by-one**. generally **atomic**.

- A ***transaction dispatcher*** briefly processes a transaction and ‘hands’ that transaction to a specific ***transaction handler*** designed to specifically ‘handle’ that kind of transaction**.**

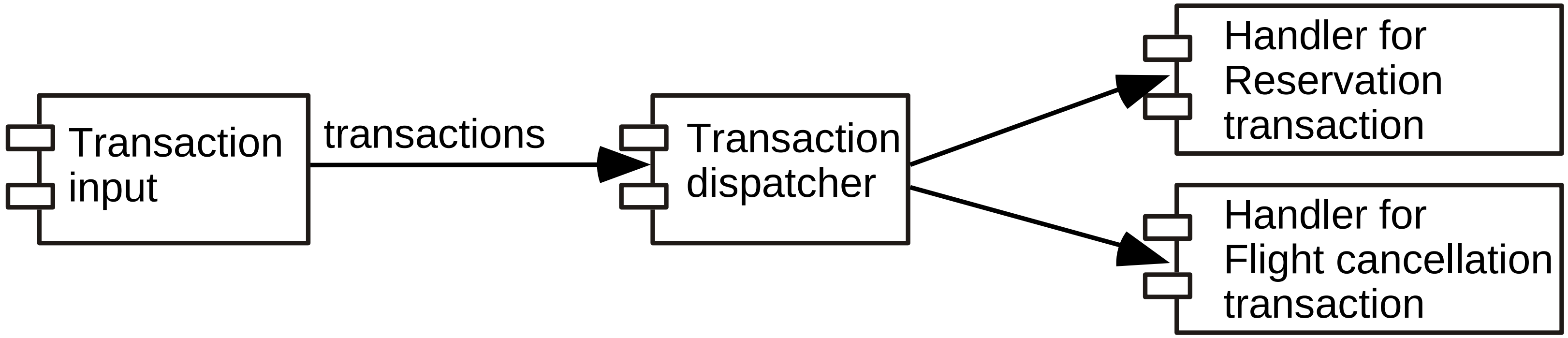
- The ***transaction*** ***handle***r is specifically designed and implemented to handle ‘a’ specific type of transaction.

Transaction Dispatchers – Some Complexity

- Comments:
  - In a threaded environment, where many transactions may be ‘in process,’ data to be modified must be *locked* and *released* as appropriate. Additional complexity.

  - Particularly complicated when an application needs to *perform a query* ***prior*** to an update transaction all the while ensuring that the data is not changed…
  - See database books on the details.

Example of a Transaction-Processing System



Recognize that these 'components' might exist on a <u>local device or remotely</u>.

The 'component' may simply be an <u>implementation</u> of some design subsystem.

The Transaction-Processing Architecture And Design Principles (As Usual, These Are <u>Very</u> Good..)

1. *Divide and conquer*: The **transaction handlers** are <u>**suitable system divisions**</u> that can be given to <u>separate</u> software engineers for detailed design and development.

2. *Increase cohesion*: Transaction handlers <u>are</u> **naturally** cohesive units.

   A 'hander' accommodates <u>only 'that' transaction.'</u>

3. *Reduce coupling*: Separating the dispatcher from the handlers clearly <u>**reduces coupling**</u>.

7. *Design for flexibility*: One may <u>**readily add new**</u> transaction handlers to handle additional transactions.

11. *Design defensively*: One may add <u>**assertion**</u> <u>**checking**</u> in each transaction handler and/or in the dispatcher.

5. The Pipe-and-Filter Architectural Pattern

Streams of data, in a relatively simple format, passed through <u>series of processes</u>

- Data constantly fed into a pipeline; Each component transforms the data in some way.

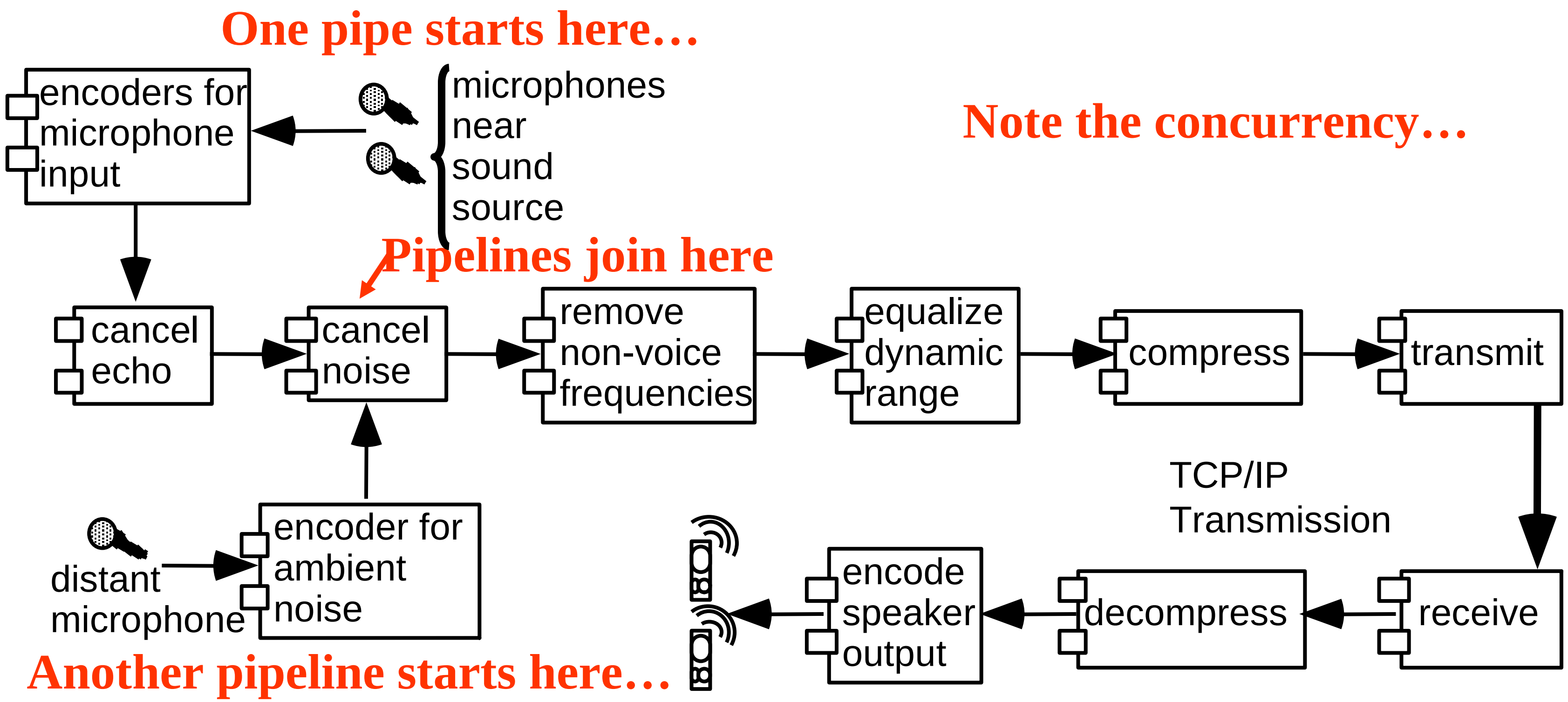
- The processes work ***<u>concurrently</u>***. Constant data in and coming out.

- Very flexible architecture.
  - —Almost all the components could be **removed**.
  - —Components may **added**, **changed**, **deleted**, **reordered**…

- Very flexible particularly (for example) as in <u>converting</u> **<u>data</u>** or <u>filtering</u> out (removing) **<u>characters</u>** or '**<u>features</u>**', etc.

- Sometimes (oftentimes) data might undergo a series of **<u>transformations</u>**…
- Can also split pipelines or join pipelines together.

Example Of A Pipe-and-filter System



Think in terms of manufacturing processes, process control applications or a GPS system.

Used more frequently in scientific/engineering systems than in information systems applications.

Pipe-and-Filter Architecture: Design Principles

1. *Divide and conquer*: The separate processes can be **independently** designed.

2. *Increase cohesion*: The processes have ***functional cohesion***. (single input; single output; no side effects…)

3. *Reduce coupling*: The processes have only ***one*** input and ***one*** output.

4. *Increase abstraction*: The pipeline components are often good abstractions, ***hiding*** their internal details.

5. *Increase reusability*: The processes can often be used in **many** different contexts.

6. *Increase reuse*: It is often possible to find reusable components to insert into a pipeline.

6. The Model-View-Controller (MVC) Architectural Pattern

Architectural pattern to help **separate *user interface layer* from other parts of the system**

Great way to have **layered cohesion**, as interfaces or controlled.

Coupling reduced between UI layer and rest of system.

- **THE MVC pattern separates the**
  - **Model:** the functional layer (business entities, 'key
  - abstractions,' the objects, relations, ...) from the
  - **View**: the user interface and the
  - **Controller;** the **director / sequencer** of the activities in response to the user.

MVC Architectural Pattern

- The ***model*** contains the underlying **classes** whose instances **(objects)** are to be viewed and manipulated
  - —Model will likely contain classes from the **domain** that may be general and form the **application** itself, which may be unique or specialized to the application.
  - —These may be very complicated software objects.

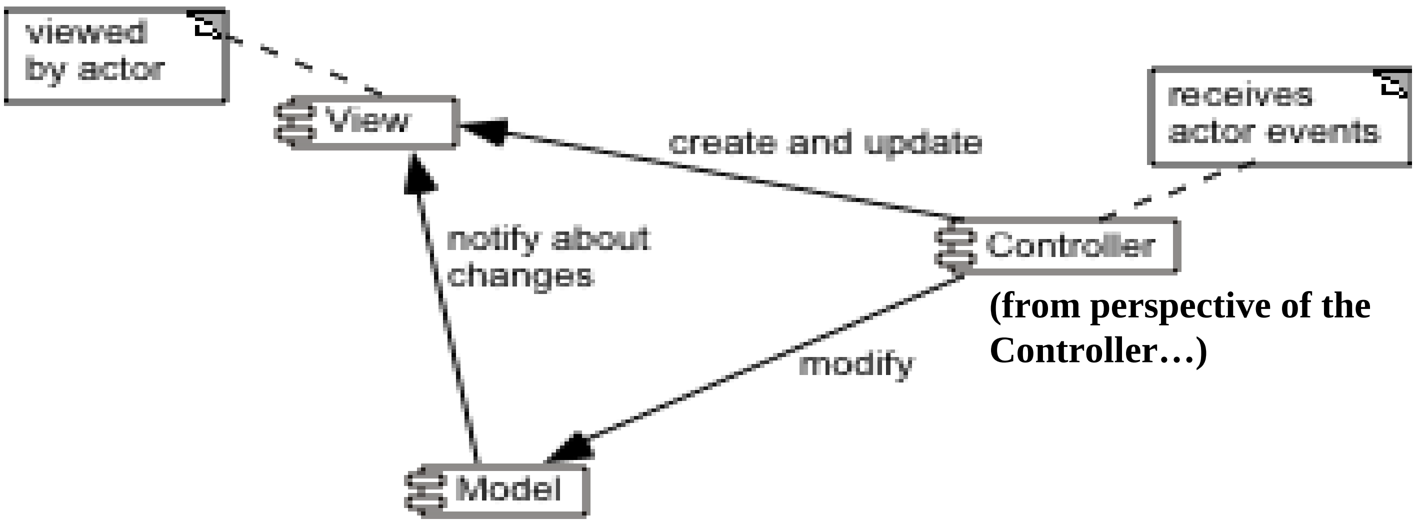
- The ***view*** contains objects used to **render the appearance** of the data from the model in the **user interface** and the **controls** with which an actor can interact.

- The ***controller*** contains the objects that **control** and **handle** the user's interaction with the view and the model.
  - —Controller contains business logic…and response to events.

(The **Observable design pattern** is normally used to separate the model from the view (later))

Example of the MVC architecture for the UI

- MVC exhibits **<u>layer cohesion</u>**, as the model **has no idea** what view and controller are attached to it (doesn't care!).
- **Model** is '<u>passive</u>' in this respect.
- The **View** (UI), business services (controller), and **model** (business entities / core abstractions) will reside in **<u>different architectural layers</u>**.



(from perspective of the Controller…)

There may be special cases when no controller component is created, but the separation of the model from the view is still essential.

The MVC Architecture and Design Principles

Know These!

1. *Divide and conquer*: Three components can be somewhat independently designed.

2. *Increase cohesion*: Components have **stronger** layer cohesion than if the view and controller were together in a single UI layer.

3. *Reduce coupling*: **Minimal** communication channels among the three components.

6. *Increase reuse*: The **view** and **controller** normally make ***extensive*** use of reusable components for various kinds of UI controls.

7. *Design for flexibility*: It is usually quite easy to change the UI by changing the **view**, the **controller**, or both.

10. *Design for **testability***: Can *test* application separately from the UI.