COMS W4156 Advanced Software Engineering (ASE)

September 14, 2021

Lecture Notes linked on Courseworks Calendar

These are “live” links, so possibly changing up to and even during class

Questions and Comments during Class

We will start with Piazza Live Q&A but may switch to something else

Software Architecture

Software engineers need version control (like git) because software changes over time, possibly very long periods of time

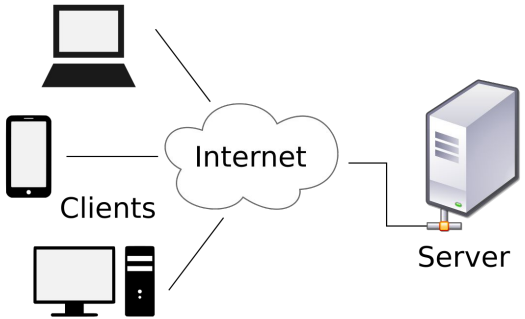
Non-trivial software is not implemented all-at-once, and software with real users changes over long periods of time to fix bugs, add features, and adapt to changing environment and context

Architecture is fundamental system structure that rarely changes (Uber's conversion from monolithic to microservices is still essentially client/server)

Example: Client/Server Architecture

Server listens for client requests and provides services to multiple clients - Most web and mobile apps work this way

“Separation of concerns” principle, here separating UI from data storage:

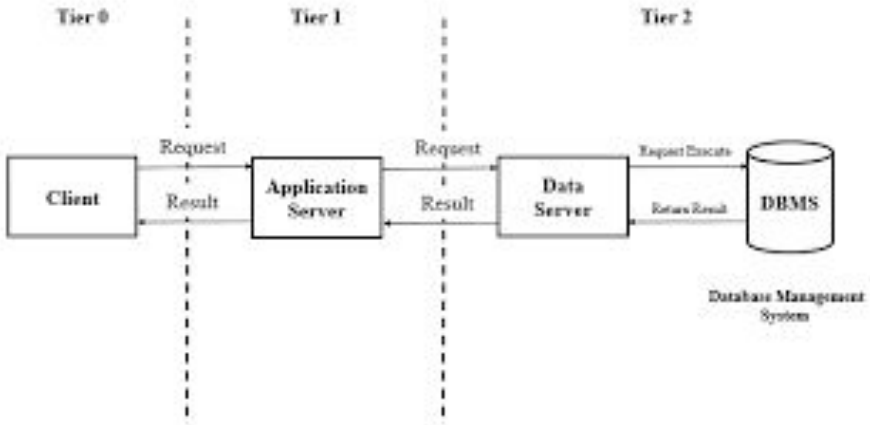


- Improves portability of UI across multiple platforms (e.g., different web browsers)
- Improves scalability by simplifying server
- Client and server can evolve independently

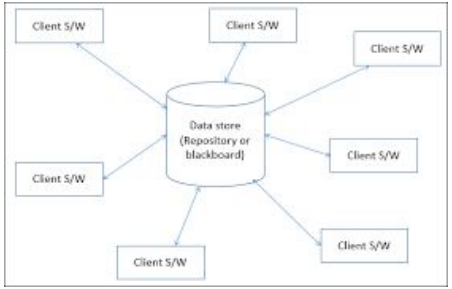
Example: Client/Server Architecture

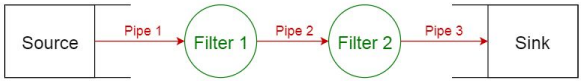
Many systems with client/server architecture are really “3-tier” (or “N-tier”)

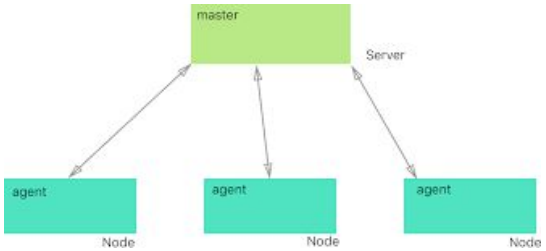
The number and purpose of tiers may change, e.g., adding load balancing, but the structure rarely does

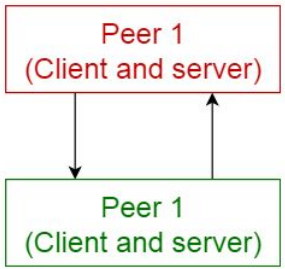


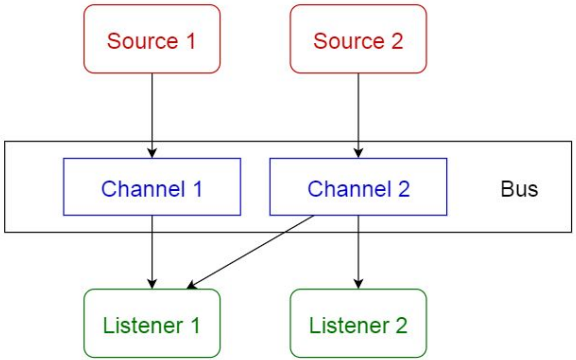
Lots More Architectures

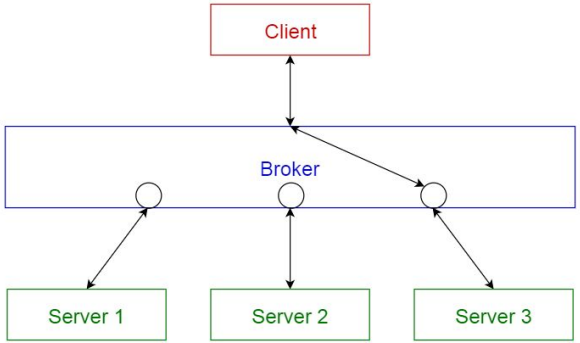












How do we choose an architecture for a particular kind of software development problem?

We look at what (nearly) everyone else uses for that kind of problem or similar problems

- *Originality is overrated*
- *Imitation is the sincerest form of not being stupid*

Our (not yet written) software almost certainly corresponds to some “pattern”

Architectural Patterns

Before there were patterns in software, before there was any software, there were patterns in cities and individual buildings

Christopher Alexander codified what architects and civil engineers had long known in his books about patterns

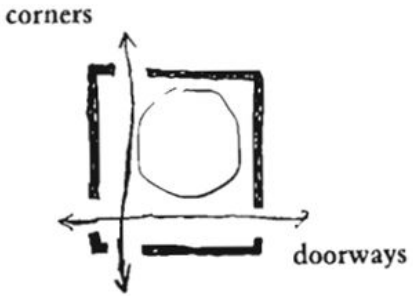
On building architecture: "*Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice.*" -- Christopher Alexander

196 CORNER DOORS*

The success of a room depends to a great extent on the position of the doors. If the doors create a pattern of movement which destroys the places in the room, the room will never allow people to be comfortable.

First there is the case of a room with a single door. In general, it is best if this door is in a corner. When it is in the middle of a wall, it almost always creates a pattern of movement which breaks the room in two, destroys the center, and leaves no single area which is large enough to use. The one common exception to this rule is the case of a room which is rather long and narrow. In this case it makes good sense to enter from the middle of one of the long sides, since this creates two areas, both roughly square, and therefore large enough to be useful. This kind of central door is especially useful when the room has two partly separate functions, which fall naturally into its two halves.

Rooms with one door.

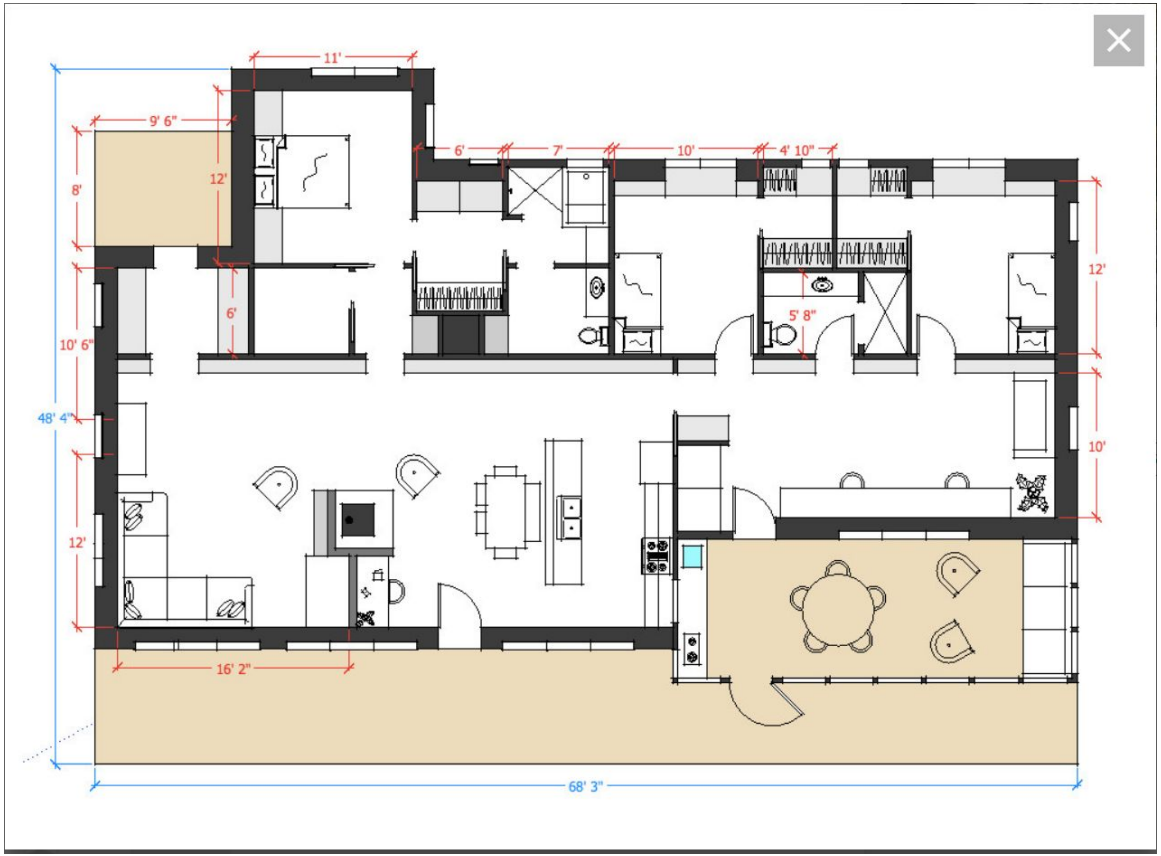


Except in very large rooms, a door only rarely makes sense in the middle of a wall. It does in an entrance room, for instance, because this room gets its character essentially from the door. But in most rooms, especially small ones, put the doors as near the corners of the room as possible. If the room has two doors, and people move through it, keep both doors at one end of the room.

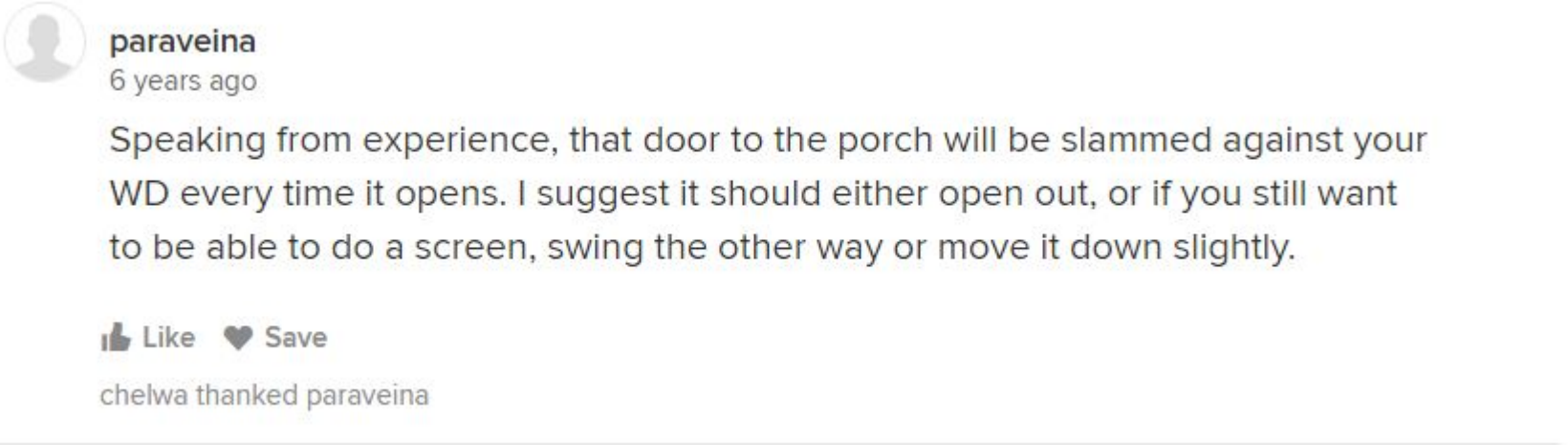
If we were designing buildings instead of software, would Professor Alexander’s pattern solve our problem of where to put doors?

Not entirely - which way should the door(s) swing?

A small house attempting to use Alexander’s "A Pattern Language"

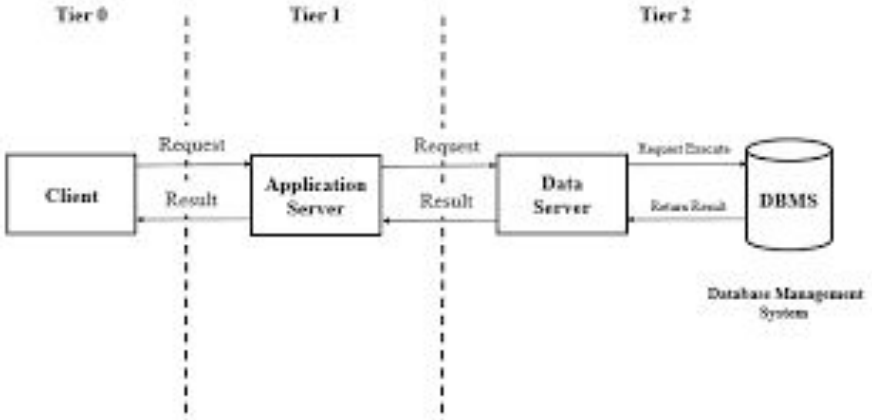


As in software architecture, building architecture patterns do not provide solutions to every aspect of the problem



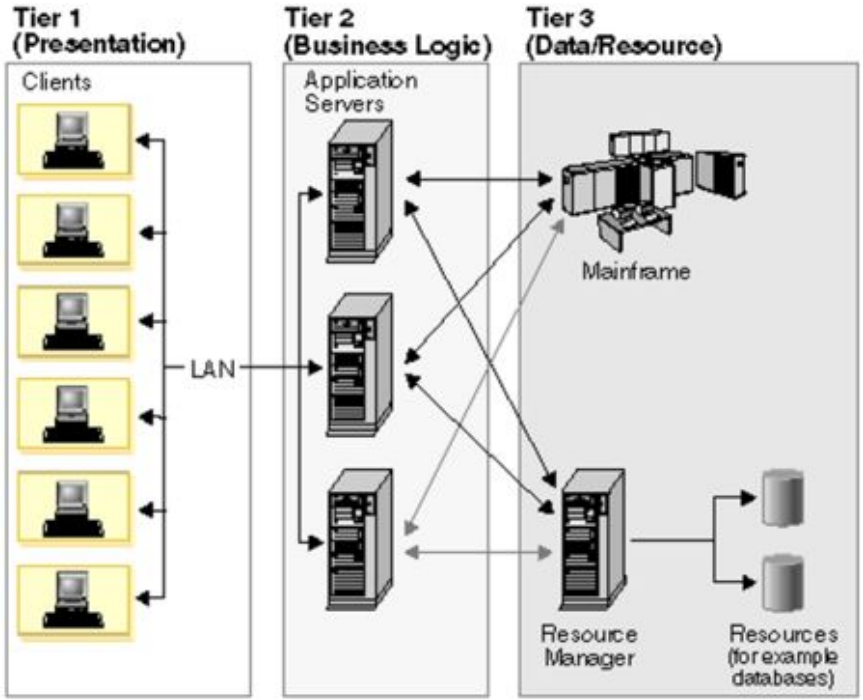
Revisit Example: N-Tier Architecture

Very generic



Revisit Example: N-Tier Architecture

A bit more detail



Revisit Example: N-Tier Architecture

Refined into a lot more detail: REST Architecture (Web before ~2000 = mostly static documents)

Emphasis on *uniform interfaces* and *caching*:

- Uniform interface decouples services provided from their implementation, encouraging independent evolvability and scalability, may degrade efficiency
- Caching improves efficiency and scalability, reduces average latency, may decrease reliability

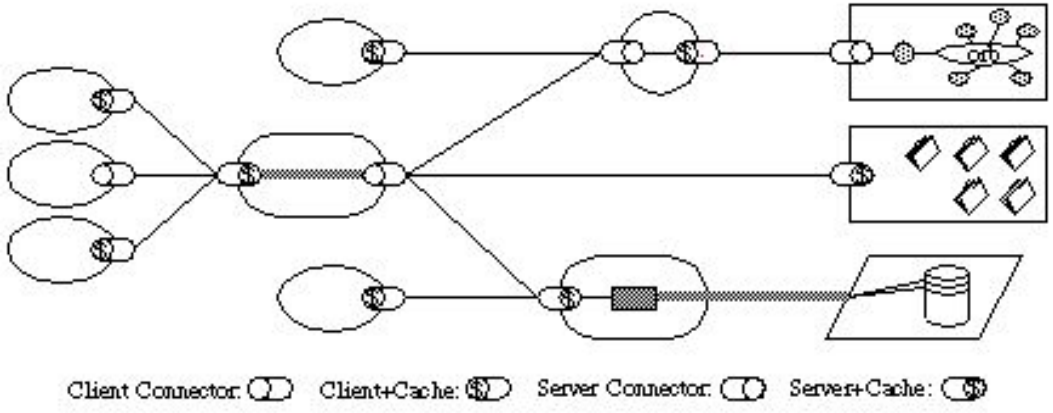
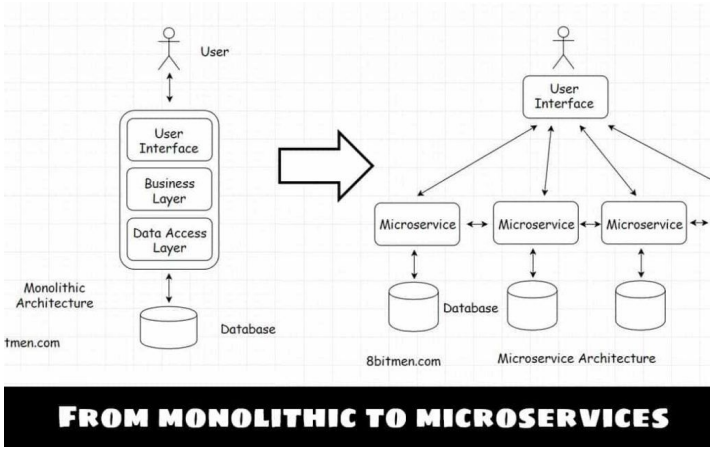
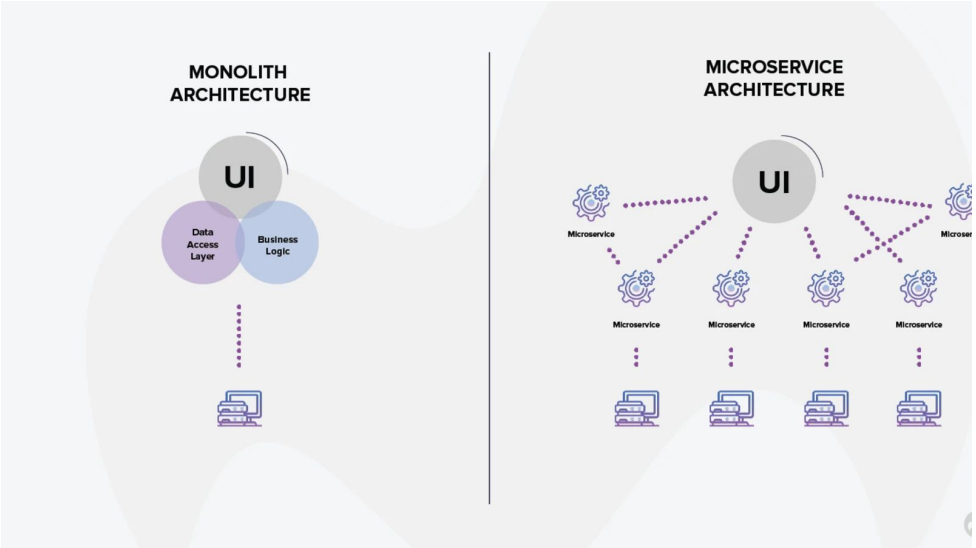


Figure 5-7. Uniform-Layered-Client-Cache-Stateless-Server

Revisit Example: N-Tier Architecture

Rearranging the detail from
[monolithic to microservices](#)
(trend starting ~2015)





Architectural Pattern Elements

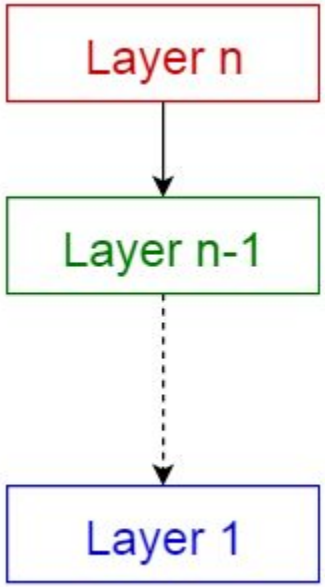
1. Name of the pattern
2. Purpose of the pattern = what problem it solves
3. How to use the pattern to solve the problem
4. Constraints to consider in solution

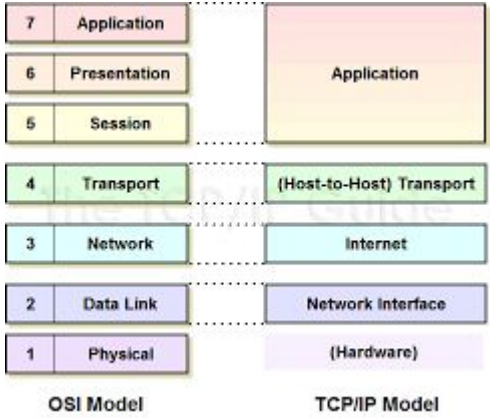
These elements work just as well for software engineering as for buildings

- Reuse well-understood and tested architectures (and designs - design patterns is a separate topic)
- Save time and don’t reinvent the wheel
- Communication language among software engineers

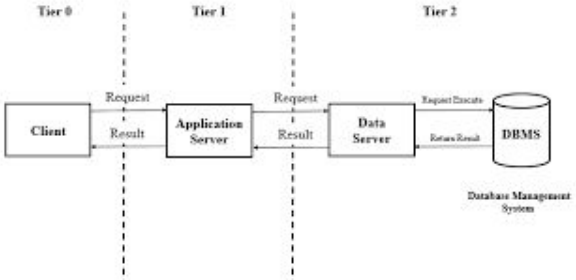
Example: Layered Architecture

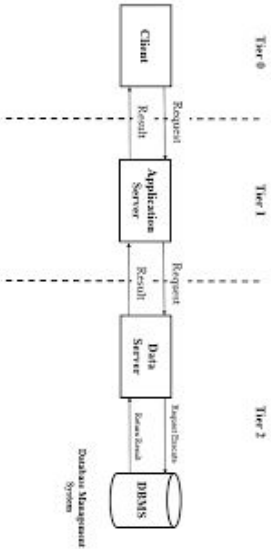
- Each layer has a defined responsibility
- Requests can not skip layers, communicate only with immediately lower layer and respond only to immediately upper layer
- Places bound on overall system complexity but adds overhead and latency
- Each layer “replaceable”: Enables encapsulation of legacy services and protects new services from legacy clients
- Security policies can be enforced at layer boundaries



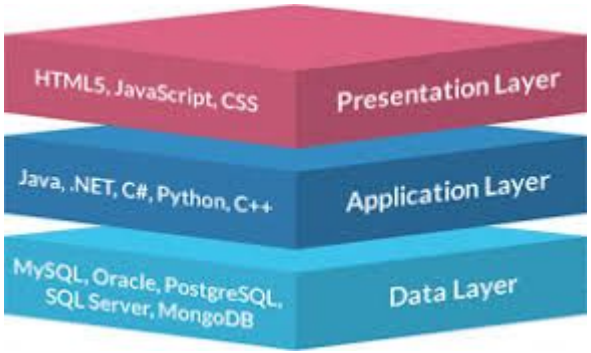


N-Tier can be viewed as a category of Layered Architecture





N-Tier can be viewed as a category of Layered Architecture



Layers are orthogonal to physical deployment

- If all in one address space, typically called layers
- If distributed across processes or machines, typically called tiers

Web Architecture

The N-tiered architecture is invisible to most developers of web apps

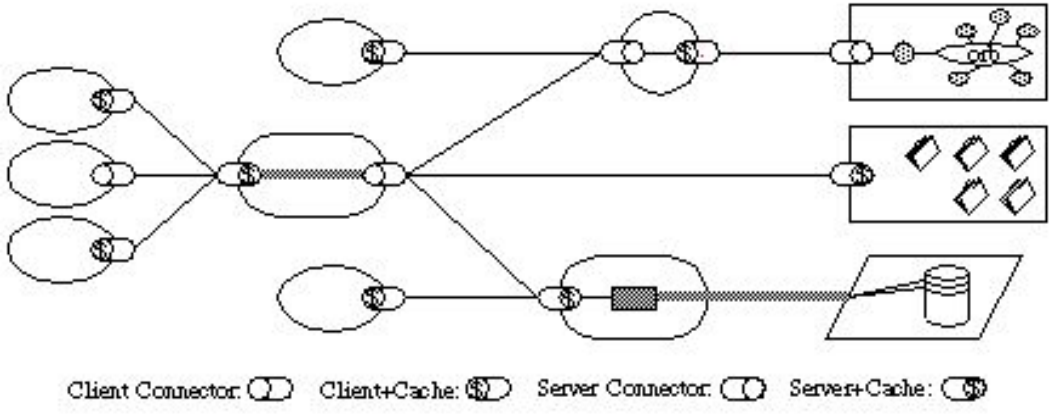
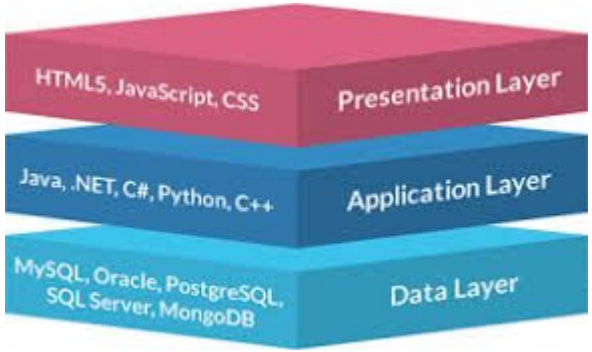


Figure 5-7. Uniform-Layered-Client-Cache-Stateless-Server

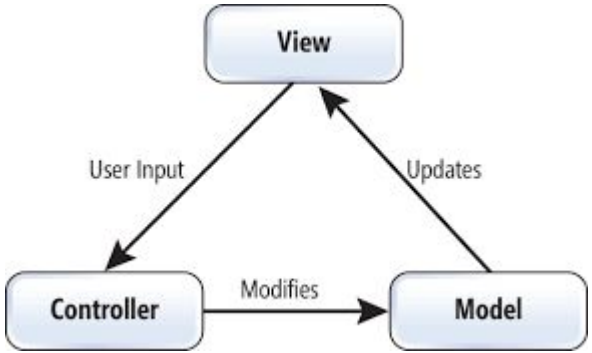
Web Architecture

Application developers think about the *internal* architecture of their applications, not just how they are deployed



Welcome Model-View-Controller (MVC)

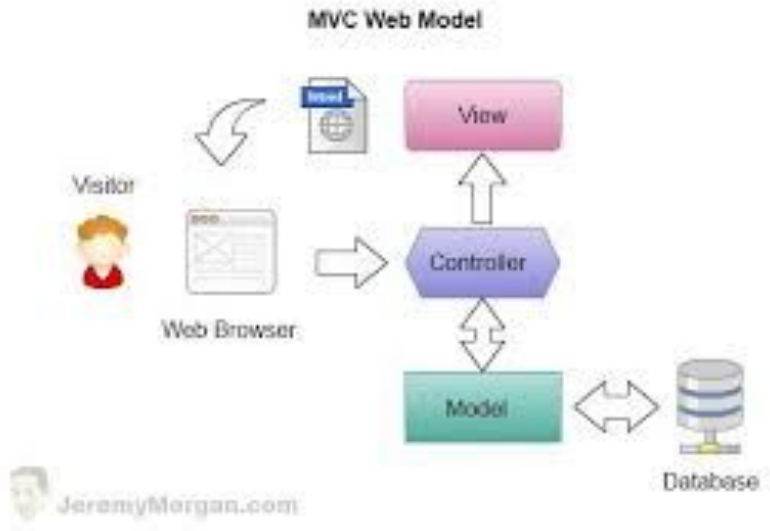
- Model contains the core functionality and data,
- View displays the information to the user,
- Controller handles user input and directs the model

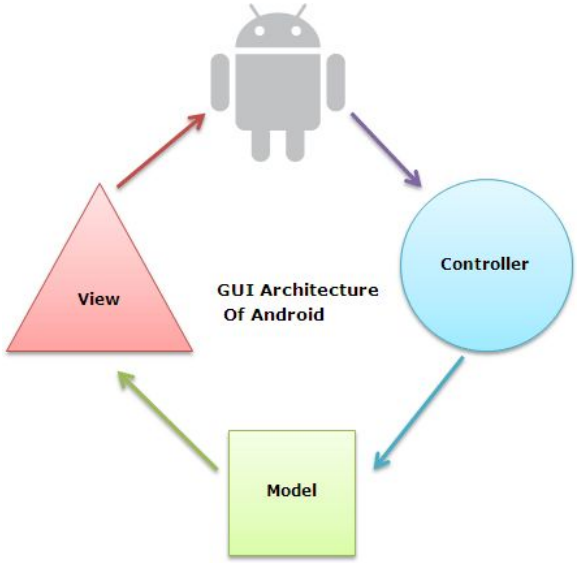


Is MVC truly an architecture or is it a design pattern?

Who cares? It works and everyone uses it or something like it

“Something like it” = numerous variants and controversies (will return to later)





MVC Analogy

In a restaurant, the Chef is the Model. It's job is to take orders (requests) from the Waiter and create food for the Waiter to deliver to the Customer

MVC Analogy

A Waiter is a Controller. Waiters take in orders (requests), take them back to the Chef (Model), and then deliver the proper meal to Customers (Users)

The Table is the View. It just holds whatever meal the Waiter brings back from the Chef

Model

Application or domain data, includes business logic that manipulates state and does most of the work - ideally a fully functional application program (with no GUI)

- Not possible to alter data except through the model
- Provides APIs to views and controllers that enable the outside world to communicate with it
- Responds to requests for information about its state (usually from the view, possibly indirectly through the controller)
- Responds to instructions to do something or change state (from the controller)

View and/or Controller may include some persistent state, such as saved layout and preferences, but the Model *owns* the domain data

View

GUI output: all CSS & HTML code goes here, updates when model changes, acts as passive observer that does not affect the model

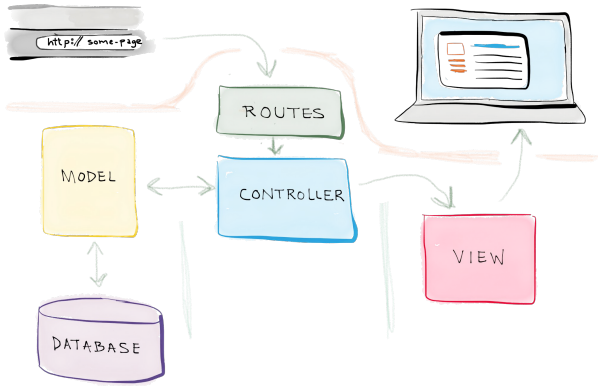
- May be multiple views of same data, such as web, tablet and phone versions of an app
- Push - view registers callbacks with model, model notifies of changes
- Pull - view polls model for changes

Controller

GUI input: accepts inputs from user, translates user actions on view into operations on model, decides what model should do

- May be structured as intermediary between view and model or as intermediary between user and view (renders/displays view for user to see)
- May be multiple controllers for same model

“Routes” = extension of MVC for REST APIs (covered in a later class)



“Separation of Concerns”

Important software engineering concept that you will keep hearing about

- Model is loosely coupled with the view and controller, knows nothing about either ⇨ *modularity*
- View and controller may be tightly coupled
- In OO languages usually implemented as two or three different classes that can be developed separately

Example: iTunes (aka Apple Music) or similar music player

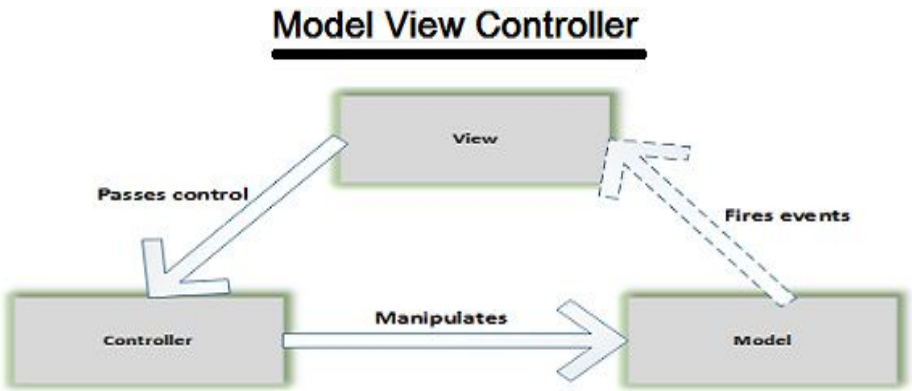
- View - playlists, current song, graphics, sound, video, etc., with some elements that accept user selection, action, data entry
- Controller - interprets the user actions applied to view and tells model what to do
- Model - stores content, knows how to play songs, shuffle, rip, and so on

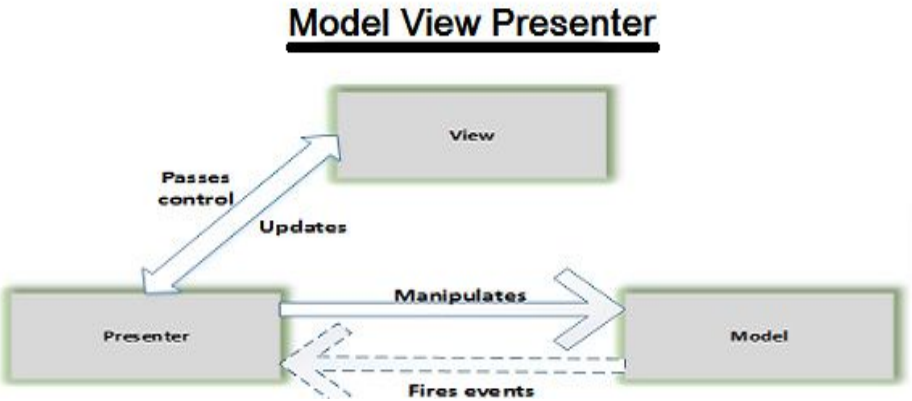
MVC Variants

Controversy over what belongs in the model

The model might be *only* the data (with possibly some bookkeeping methods) and the business logic resides in the controller, which better matches N-tier architecture

But then neither the model nor the controller is a full application by itself





How to decide?

Business logic belongs in the model when the data, and thus the model, is only part of one application

But the developer may not know in advance whether the model might eventually be used for other applications (software sustainability)

Business logic belongs in the controller when the data, and thus the model, is shared across many applications

Different controller (and presumably different view or set of views) for each application

Enables replacing model while retaining controller (and views)

Reminder: “Homework Zero”

https://courseworks2.columbia.edu/courses/135182/assignments/680926

Due tomorrow - or due six days after you joined the class if you were not enrolled on the first day of class

Individual Mini-Project

Three parts:

1. Implementing a simple game (intentionally due September 22, the day after program change period ends)
2. Testing the game (due September 29)
3. Saving game state (due October 6)

Fill in / expand provided Python skeleton OR redesign all three assignments for another language (preferably C++ or Java)

😎 It's probably easier to learn Python

See Connect Four