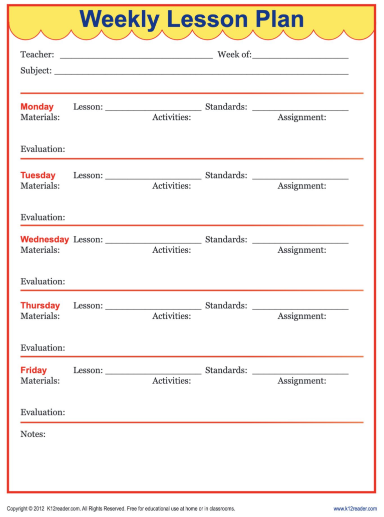
COMS W4156 Advanced Software Engineering (ASE)

November 15, 2022

Agenda

1. Software Architecture
2. MVC



Weekly Lesson Plan

Teacher: ____________ Week of: ____________
Subject: ____________

Monday Lesson: ____________ Standards: ____________
Materials: Activities: Assignment:

Evaluation:

Tuesday Lesson: ____________ Standards: ____________
Materials: Activities: Assignment:

Evaluation:

Wednesday Lesson: ____________ Standards: ____________
Materials: Activities: Assignment:

Evaluation:

Thursday Lesson: ____________ Standards: ____________
Materials: Activities: Assignment:

Evaluation:

Friday Lesson: ____________ Standards: ____________
Materials: Activities: Assignment:

Evaluation:

Notes:

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. Refactoring along the way may substantially re-organize system structure

Design principles. such as the SOLID principles, are centered on change, trying to make the code more resilient to change - more flexible, extensible, and maintainable (micro level)

Architecture is fundamental system structure that rarely changes (macro level)

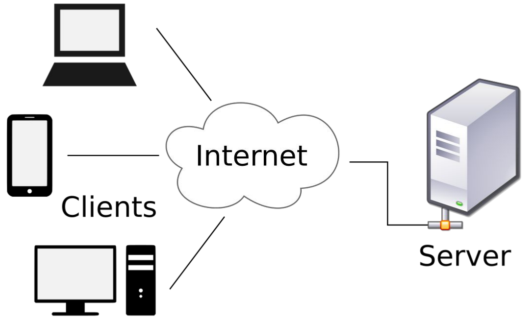
Client/Server Architecture

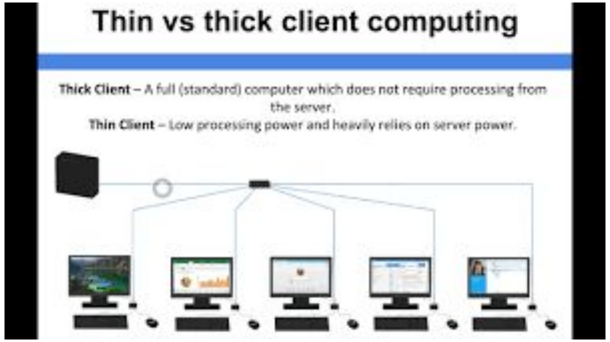
Network computing model where server listens for client requests and sends responses to multiple clients

Most web apps work this way, with the frontend running in a browser and the backend running in a web server or application server

“[Separation of concerns]” principle, in the simplest case separating UI from data storage:

- Improves portability of UI across multiple platforms (different web browsers on different operating systems)
- Improves scalability by simplifying server
- Client and server can evolve independently




Thin vs thick client computing

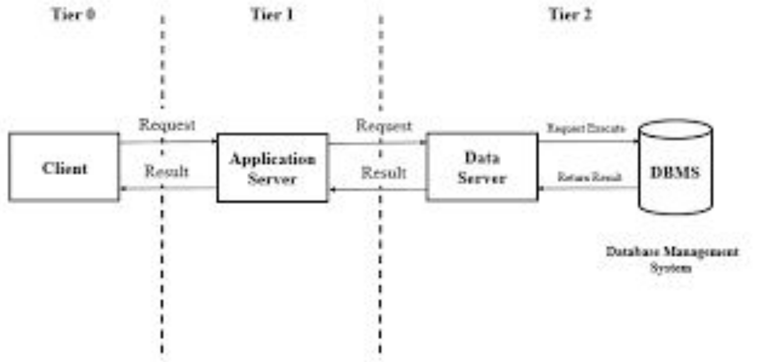
Thick Client – A full (standard) computer which does not require processing from the server.
Thin Client – Low processing power and heavily relies on server power.

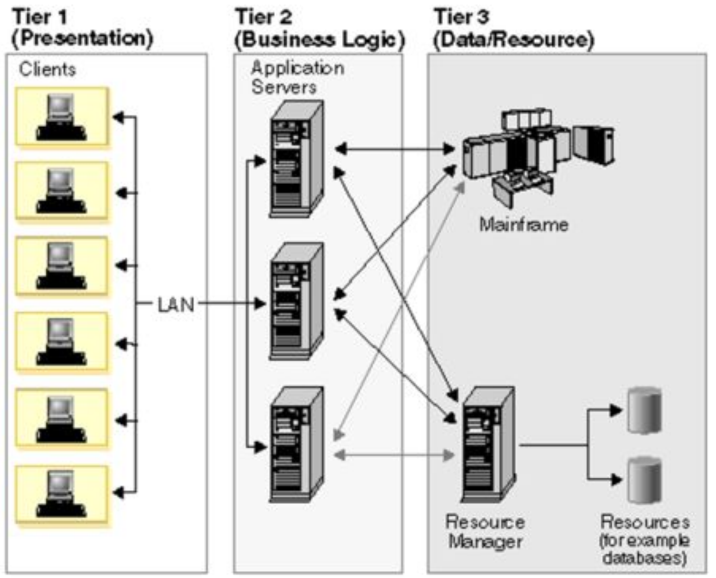
N-Tier

Client/Server is so basic that we take it for granted, but the pure multiple-client/single-server form is rarely used

“Client/Server” software is usually 3-Tier (or N-Tier)

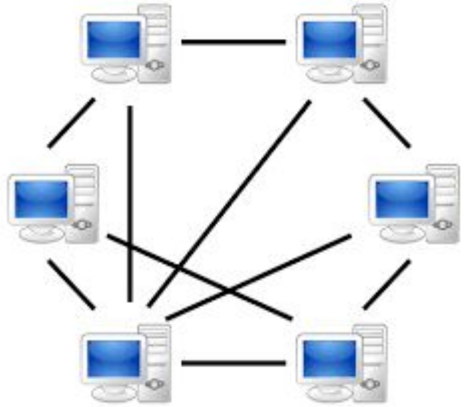
The number and purpose of tiers may change, e.g., adding caches and load balancers, but it’s still conceptually client/server



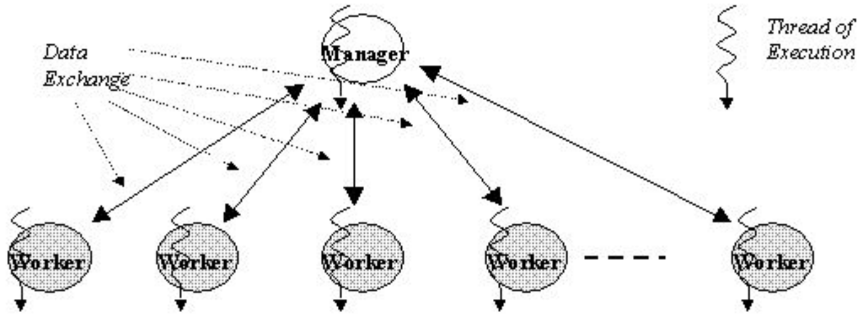


Alternatives to Client/Server for Web Apps?

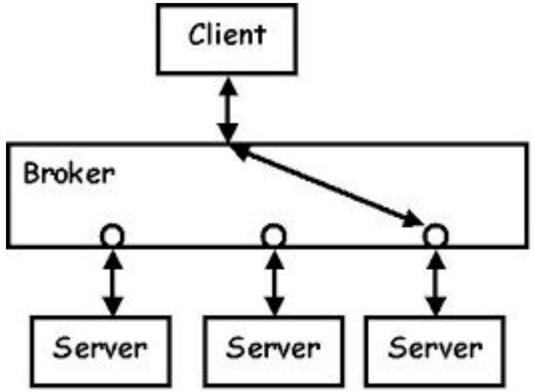
Peer to peer



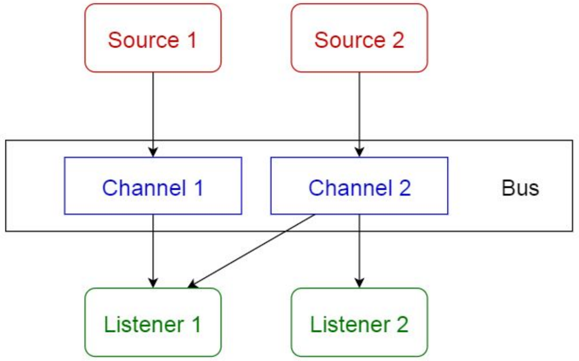
Manager/Worker



Broker



Event-Driven



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

For most projects, the developers invent their own design.
Should they also invent their own architecture?

How do we know when an architecture is “good”, is there an objective basis for such judgment?

- If we invent the architecture ourselves, we want to be able to judge whether it is good, whether it is the *right* architecture for our problem
- If we reuse a well-known architecture, someone else already judged whether it is good, but we still need to judge whether it is right architecture for *our* problem

How do we choose an architecture for a particular kind of software development problem? 2

Do we need to produce novel solutions for every problem? Is our system so unique in its architecture such that we need to design from the ground up?

➔ **NO,** someone somewhere has already solved our problem or a very similar 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

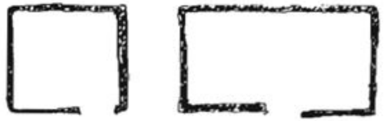
Christopher Alexander codified what architects and civil engineers had long known in his books about architectural patterns - architecture for buildings, not software

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

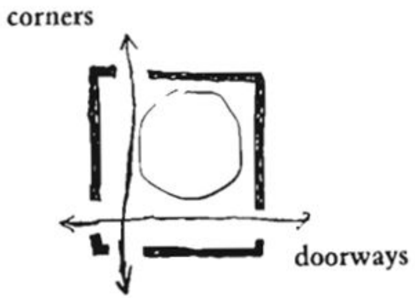
196 CORNER DOORS*

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.



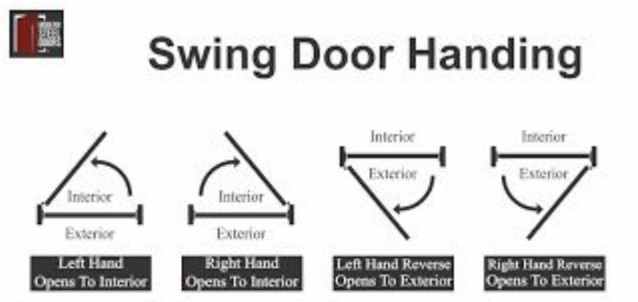
Rooms with one door.



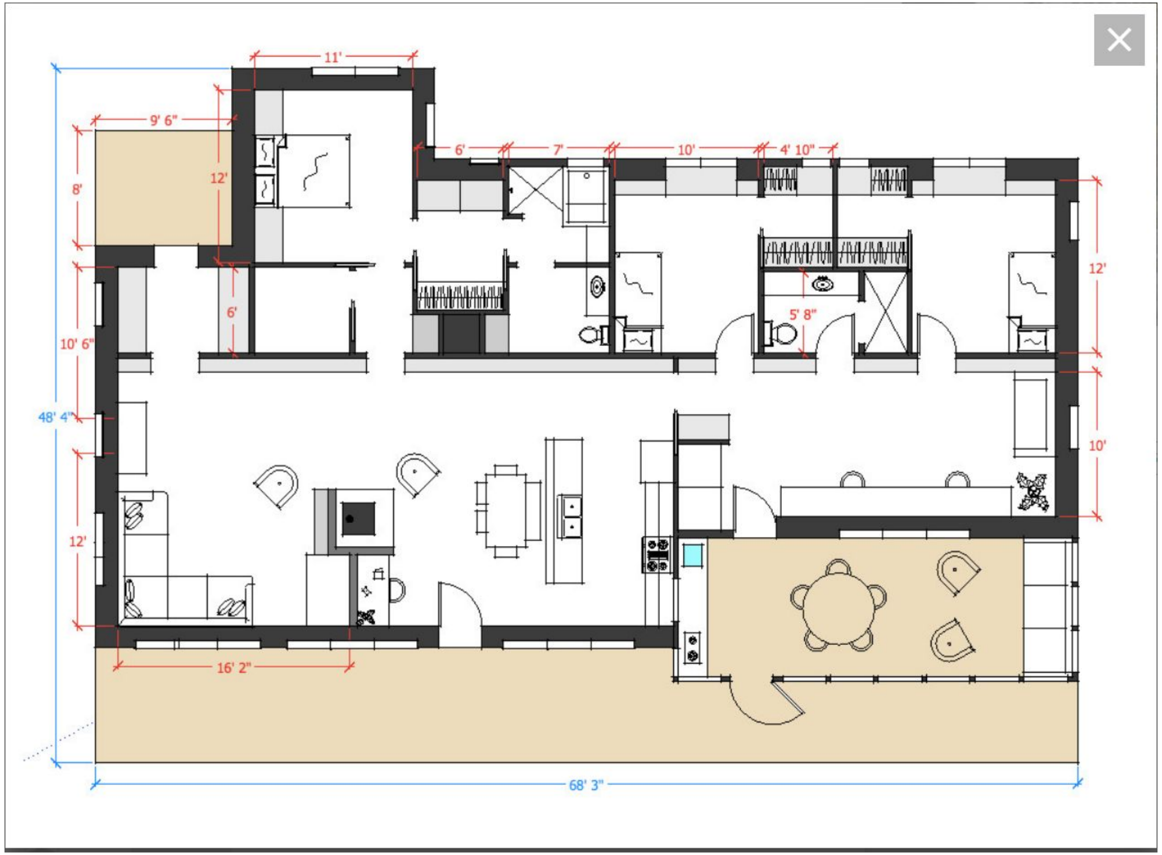
Except in very large rooms, a door only rarely makes sense in the middle of a wall. It does in an entrance room, for instance, because this room gets its character essentially from the door. But in most rooms, especially small ones, put the doors as near the corners of the room as possible. If the room 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?

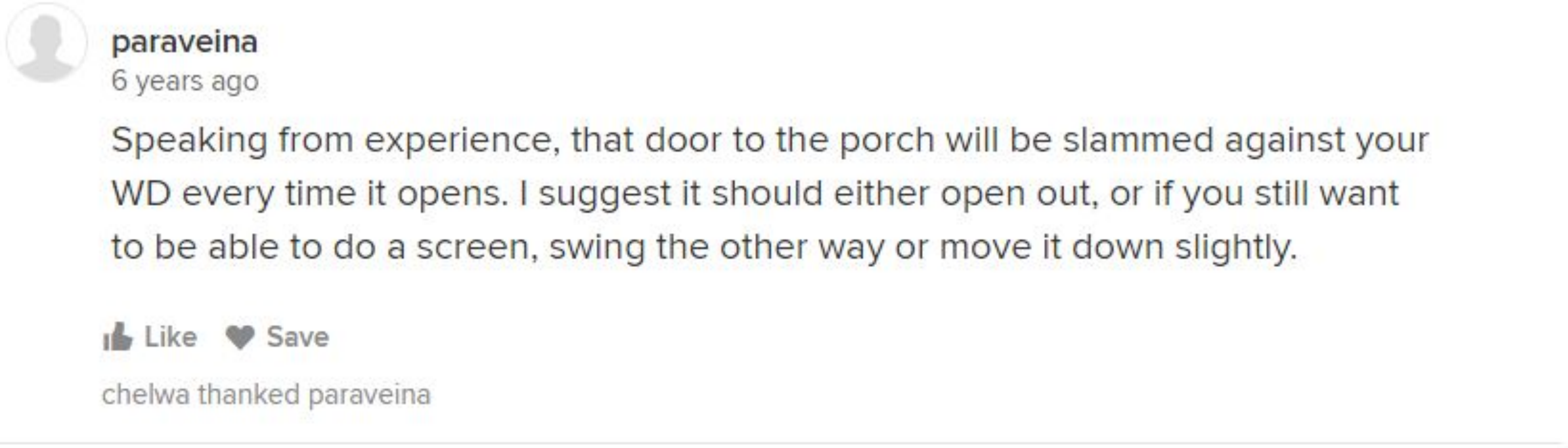
Not entirely - which way should the door(s) swing?


Swing Door Handing
Interior
Exterior
Left Hand Opens To Interior
Interior
Exterior
Right Hand Opens To Interior
Interior
Exterior
Left Hand Reverse Opens To Exterior
Interior
Exterior
Right Hand Reverse Opens To Exterior

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



paraveina
6 years ago

Speaking from experience, that door to the porch will be slammed against your WD every time it opens. I suggest it should either open out, or if you still want to be able to do a screen, swing the other way or move it down slightly.

chelwa thanked paraveina

Architectural Pattern Elements

Alexander identified four elements to describe an architectural pattern:

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

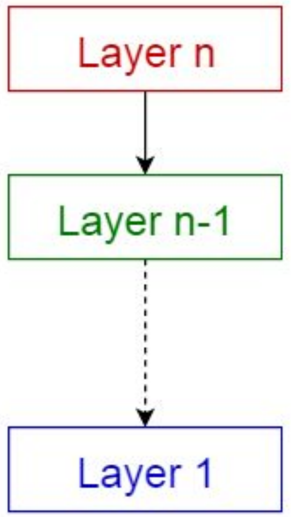
Benefits of patterns in software engineering

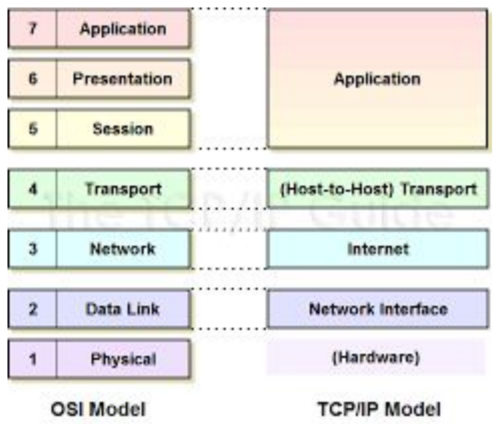
- Reuse well-understood and tested architectures (and designs - design patterns is a separate topic later)
- Save time and don't reinvent the wheel
- Communication language among software engineers

In software engineering context, usually called "architectural style" rather than "architectural pattern"

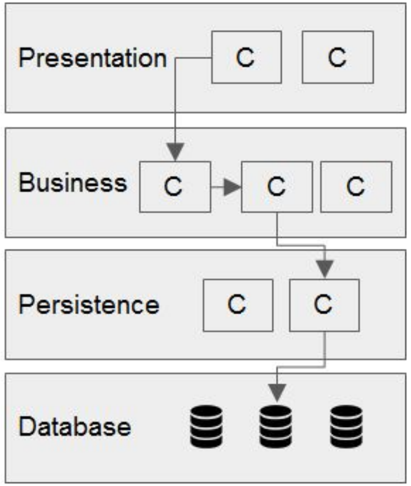
Example: Layered Architecture

- Each layer has a defined responsibility and provides services to the next higher layer and implements those services using the next lower layer
- Requests cannot 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



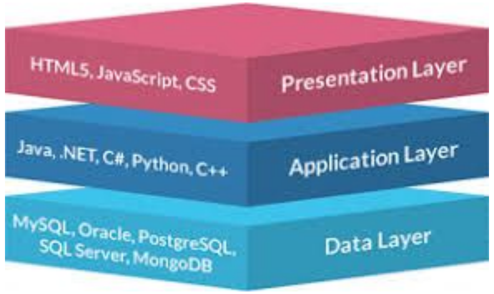


N-Tier can be viewed as a category of Layered Architecture

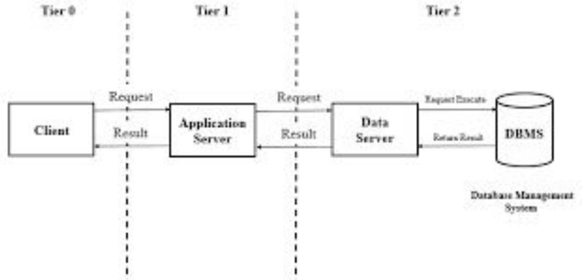


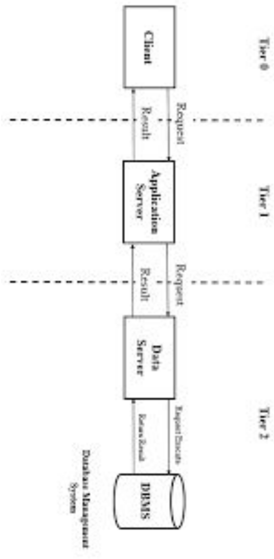
Layers are orthogonal to physical deployment and do not need to be located on the same machine

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


HTML5, JavaScript, CSS
Presentation Layer
Java, .NET, C#, Python, C++
Application Layer
MySQL, Oracle, PostgreSQL, SQL Server, MongoDB
Data Layer

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





Web Architecture

REST Architecture reflects web as it was 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

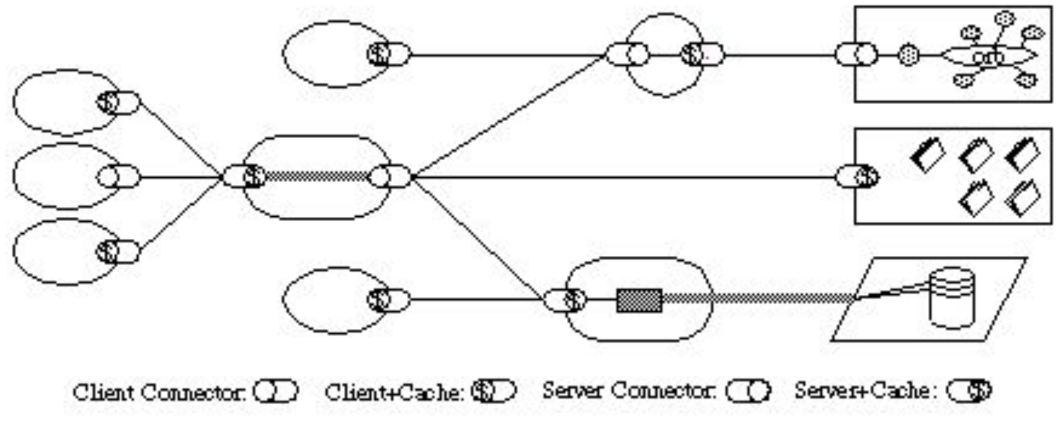
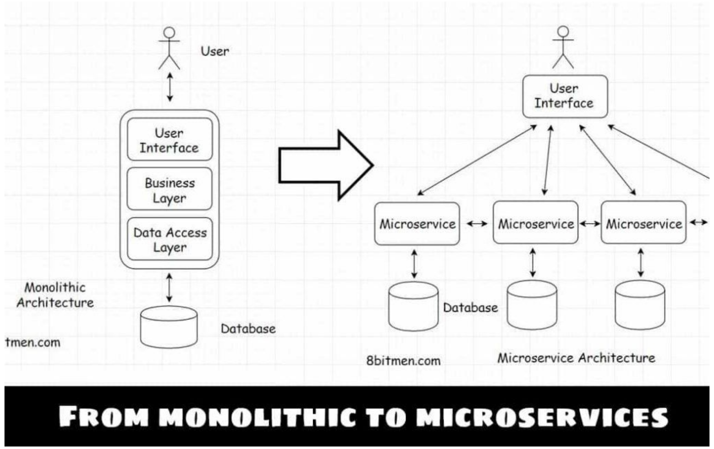


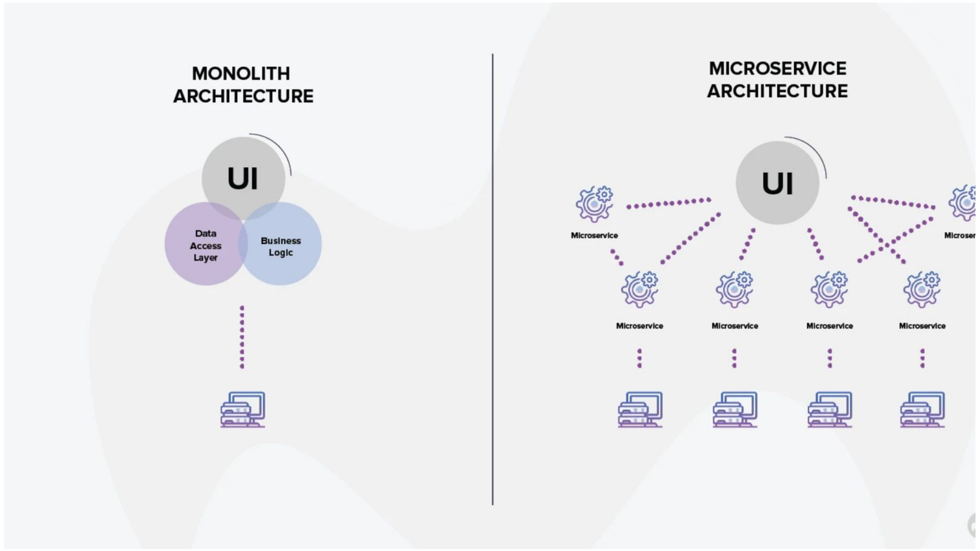
Figure 5-7. Uniform-Layered-Client-Cache-Stateless-Server

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

Microservices Architecture

Business layer tier reorganized from monolithic to microservices (trend starting ~2014)





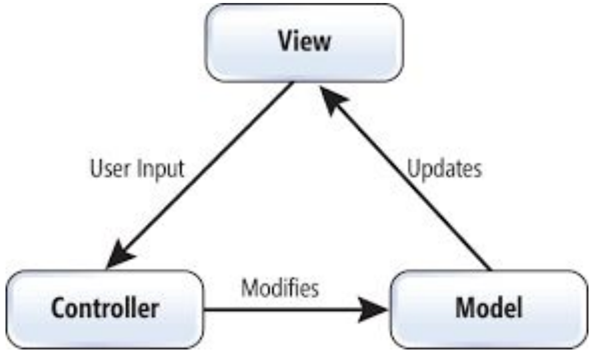
Agenda

1. Software Architecture
2. MVC

Model-View-Controller (MVC)

How to structure a GUI application is a very common architectural problem but “Presentation” does not tell us enough, so there is a standard architectural pattern

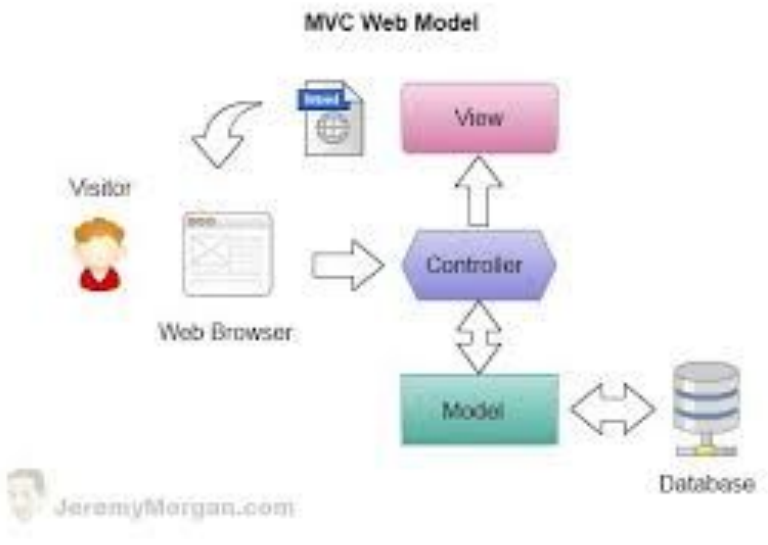
- Model contains the core functionality and data,
- View displays the information to the user,
- Controller handles user input entered through the view and directs the model to ‘do something’

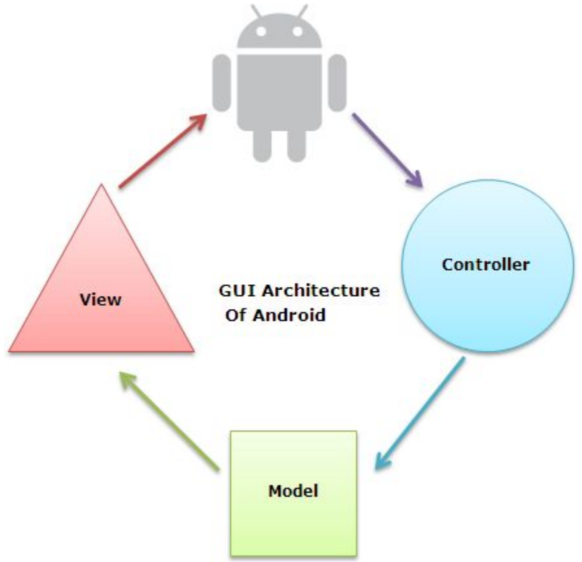


Is MVC truly an architecture or is it a design pattern?

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

“Something like it” = numerous variants and controversies





MVC Analogy

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

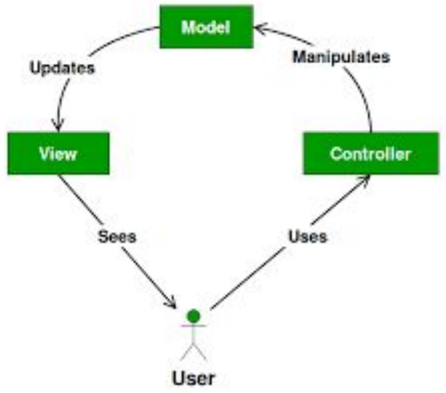
MVC Analogy

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

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

There can be multiple controllers and views for the same model

Model



Includes business logic that manipulates state and does most of the work as well as CRUD operations for persistent state (application or domain data). 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 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

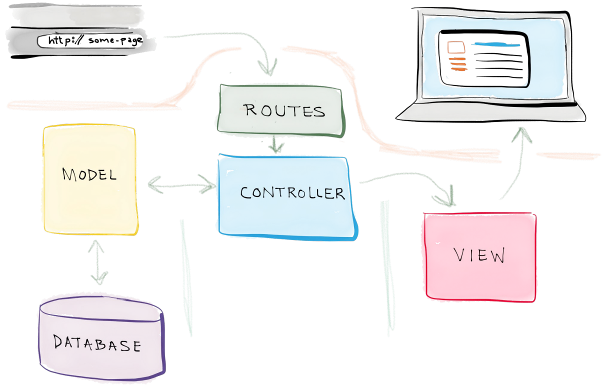
- May be multiple views of same data, such as web, tablet and phone versions of an app
- Push - view registers callbacks with model, model notifies of changes
- Pull - view polls model for changes

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

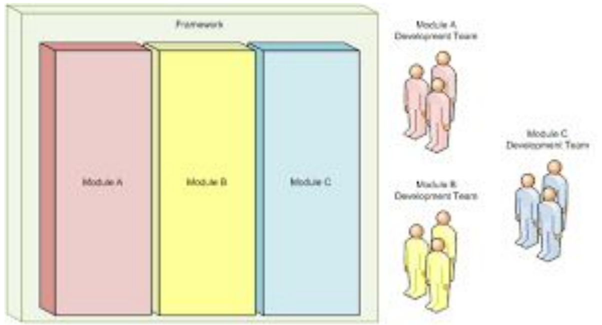
“Routes” = extension of MVC for REST APIs



“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


Framework
Module A
Module B
Module C
Module A Development Team
Module B Development Team
Module C Development Team

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

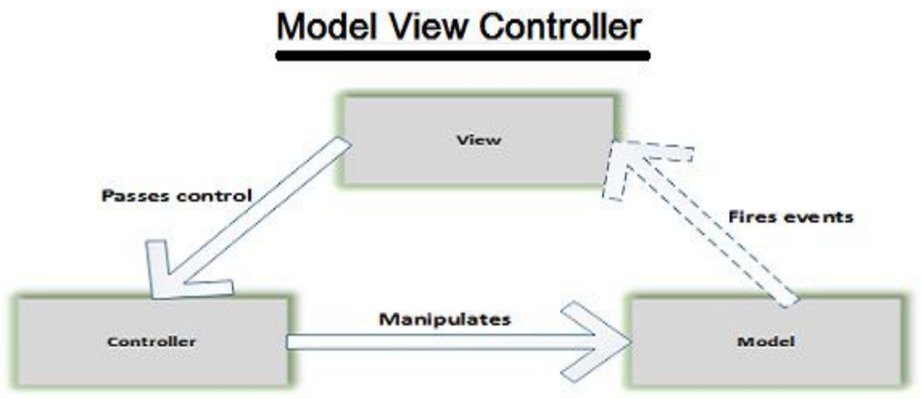
- View - playlists, current song, graphics, sound, video, etc., with some elements that accept user selection, action, data entry
- Controller - interprets user actions applied to view and tells model what to do
- Model - stores content, knows how to play songs, shuffle, rip, and so on

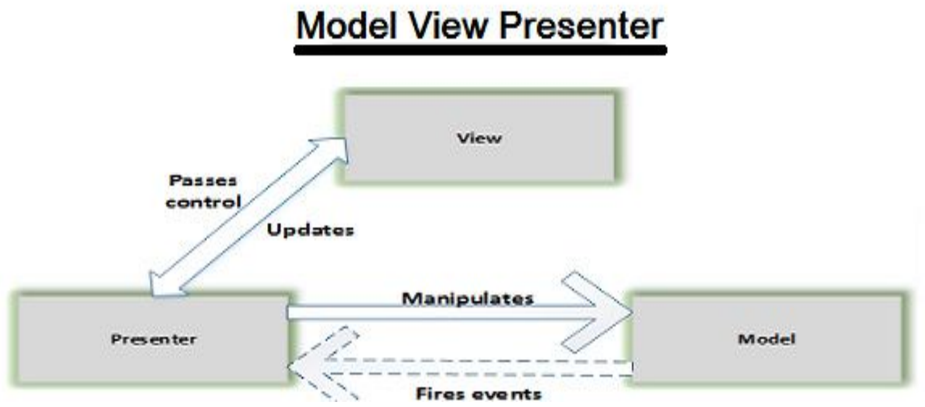
MVC Variants

[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

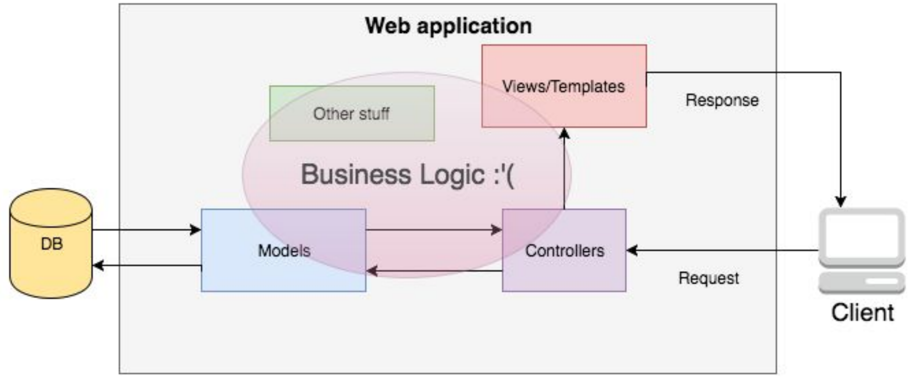
But then neither the model nor the controller is a full application by itself





How to decide?

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

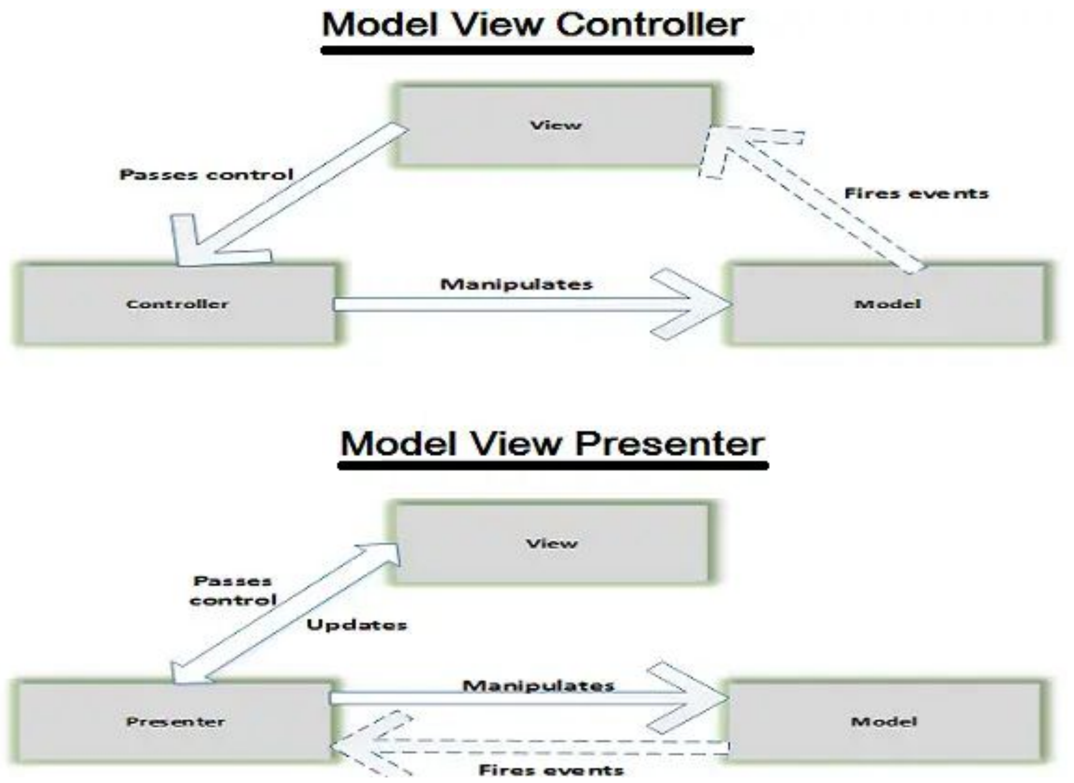


Business logic belongs in the controller when the data, and thus the model, is shared across many applications

Different controller (and presumably different view or set of views) for each application, enables replacing model while retaining controller (and views)

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

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



MVC - uses Observer Pattern to passively update view

MVP - uses Observer Pattern to passively update presenter, which in turn actively updates the view

(There are other variants)

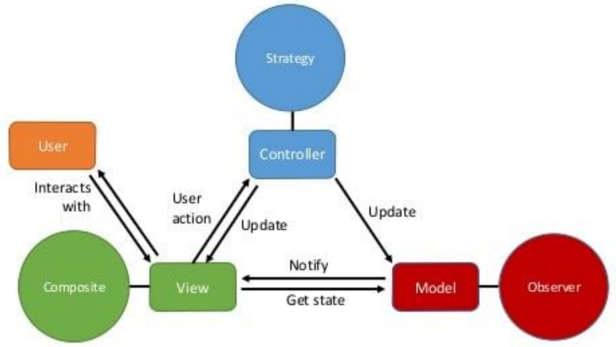
MVC/P is an Architecture <u>and</u> a *Compound* Design Pattern

Observer Pattern + Strategy Pattern

Observer Pattern (ultimately) updates the view to display the model, possibly multiple different views for the same model, e.g., desktop vs. mobile browser

Strategy Pattern supports different ways ("strategies") to leverage the model, e.g., multiple apps using the same service

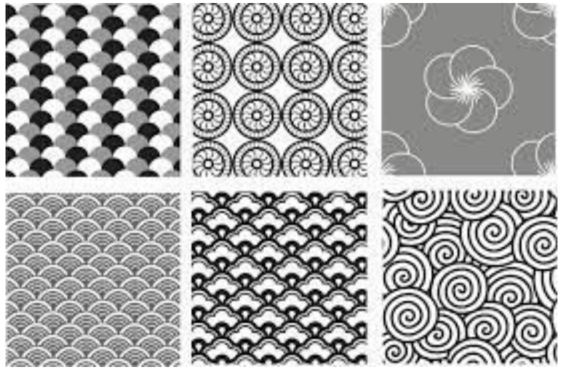
Traditional MVC & associated design patterns



1. R. Buencamino. Learning the Model-View-Controller Design Patterns in iOS. Lynda.com, 2015

Next Class

Design Patterns



Upcoming Assignments

Second Iteration due November 28 (note this is the Monday after Thanksgiving)

Second Iteration demo due December 5

Ask Me Anything