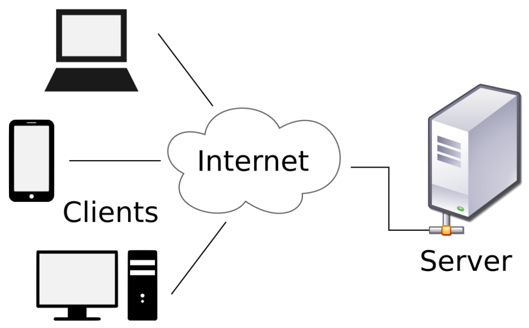
Lecture Notes
September 10, 2020

Software engineers need version control because software changes over time (in addition to needing undo).

Non-trivial software is not implemented all-at-once, and software with real users changes over long periods of time to fix bugs and add features. Adding new features sometimes requires redesigning parts of the software.

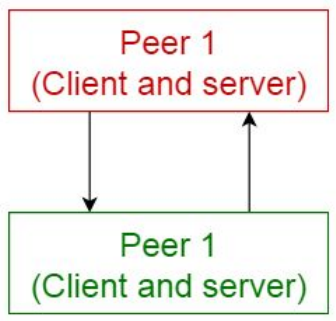
Architecture is <u>fundamental</u> system structure that rarely if ever changes, even when many new features are added.

Example: Client/Server architecture, server listens for client requests and provides services to multiple clients

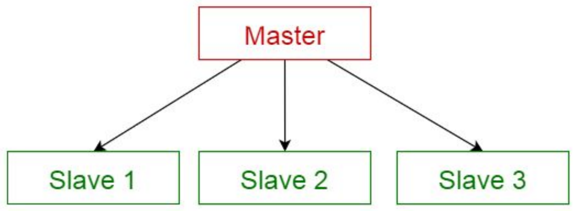


For Web applications, what internet-level architecture would you use instead?

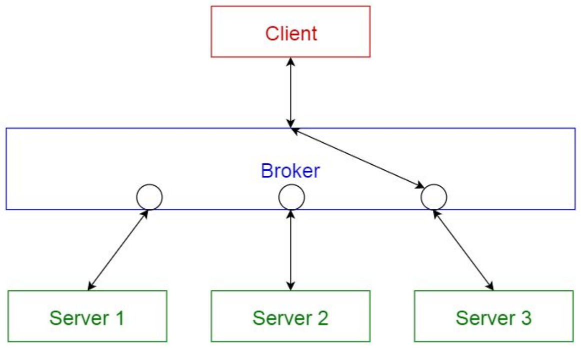
Potential alternatives to client/server … maybe ….



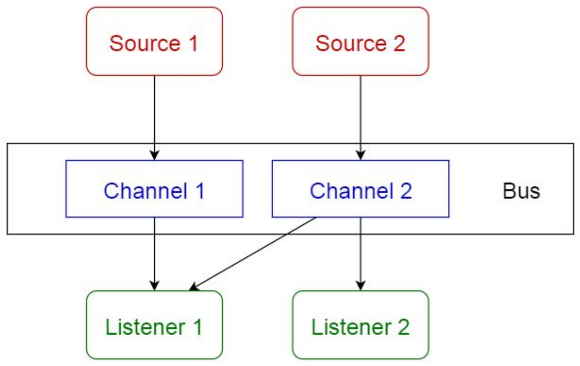
Peer-to-Peer



Master-Slave



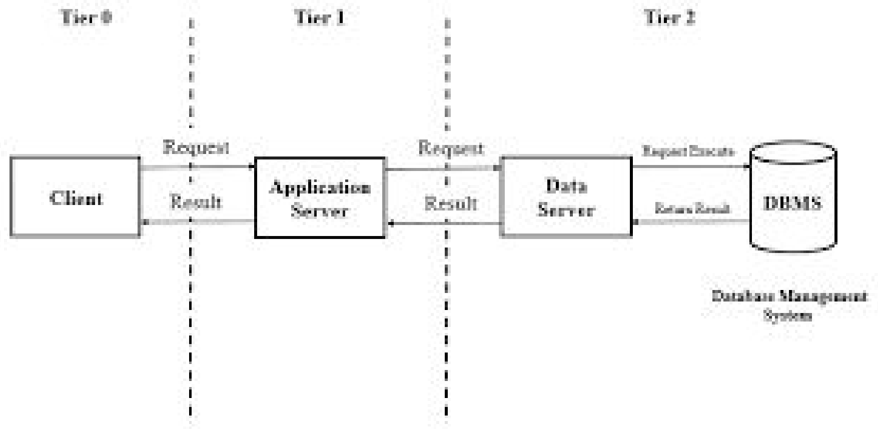
Broker



Publish-Subscribe

What about these?

3-Tier (or N-Tier)



REST

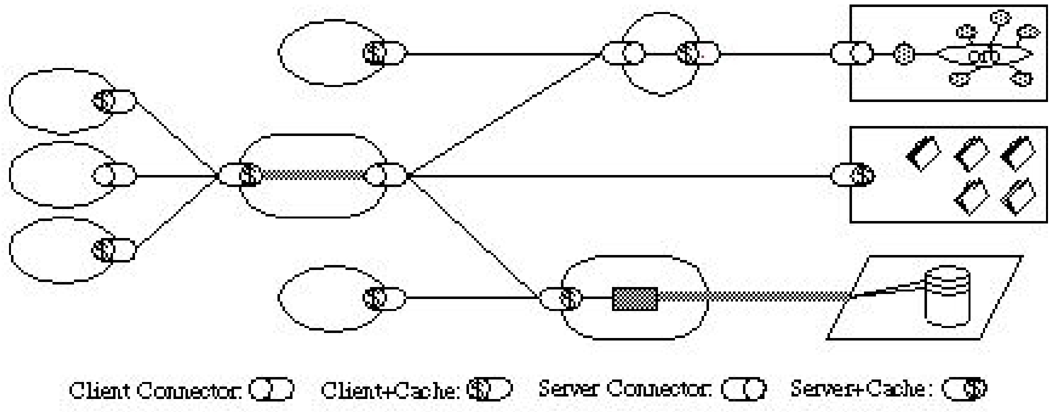


Figure 5-7. Uniform-Layered-Client-Cache-Stateless-Server

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

- 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

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
- *Originality is overrated, imitation is the sincerest form of not being stupid*
- Our (not yet written) software almost certainly corresponds to some well-known “pattern”

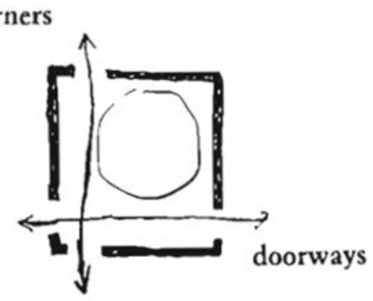
Before there were patterns in software, before there was any software, there were patterns in buildings, in cities

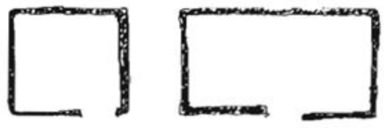
Christopher Alexander codified what architects and civil engineers had long known in his books about patterns

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*

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 this solve our problem of where to put doors?

Addendum added after class:

If we were designing buildings instead of software, would this solve our problem of where to put doors?

As one student noted in a comment, this does not directly address the problem of which direction the doors should swing. I hope Alexander's book also includes patterns for these issues, but I cannot find out without buying the book or a paid membership on patternlanguage.com. The excerpt above was part of a freely available sample.

However, I was able to find this example of someone using Alexander's patterns book to design a house: https://www.houzz.com/discussions/3278091/a-small-house-attempting-to-use-a-pattern-language. The author says *"I decided to sit down one day and go through the book and try to apply as many patterns as possible"*. One of the comments points out that the door to the porch in the 1870 sqft version of the design will hit against the washer/dryer every time it's opened, and suggests fixes. So maybe there isn't any such pattern in the book.

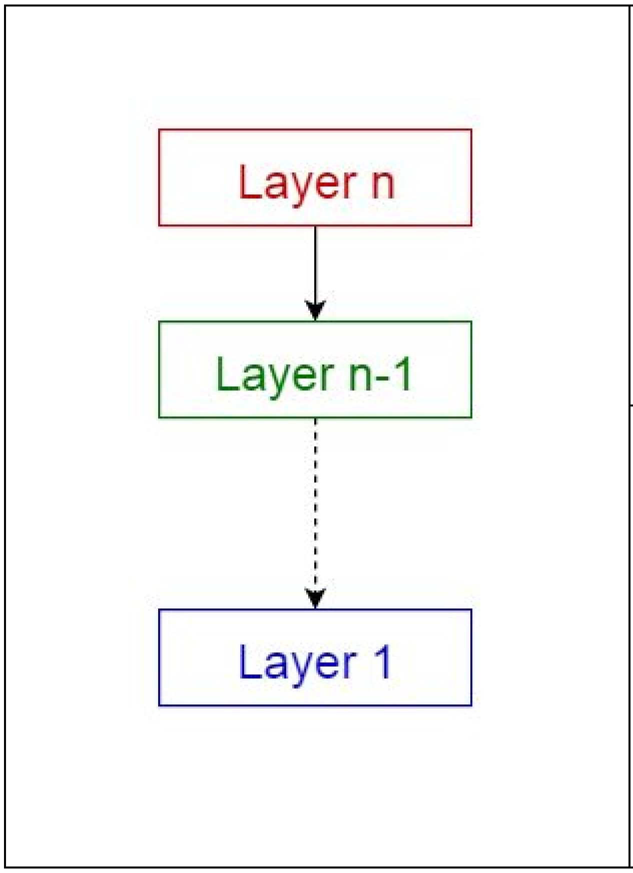
Alexander identified four elements to describe an architectural pattern:

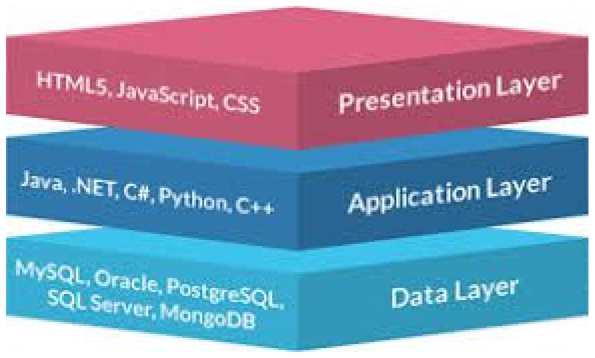
1. Name of the pattern
2. Purpose of the pattern = what problem it solves
3. How to use the pattern to solve the problem
4. Constraints to consider in solution

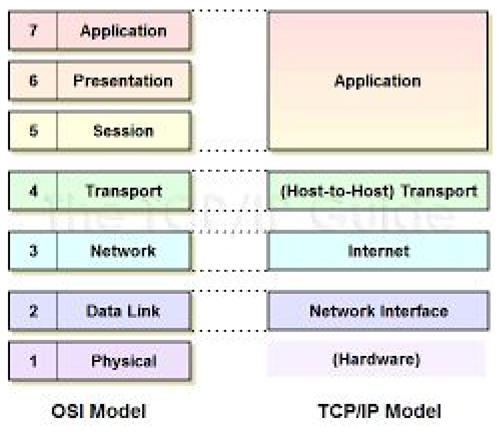
Benefits of patterns in software engineering

- 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

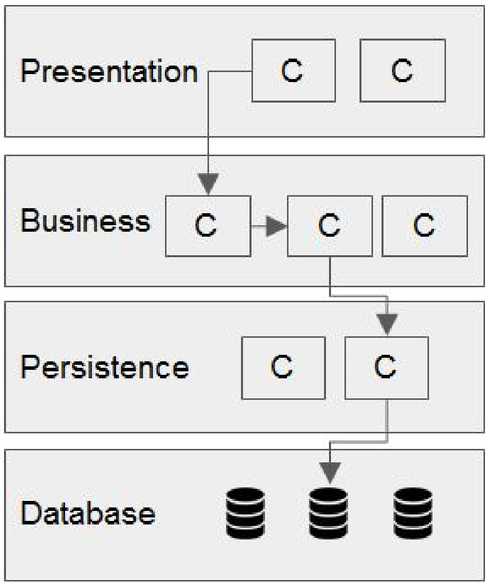
Layered architecture = each layer provides services to the next higher layer and implements those services using the next lower layer



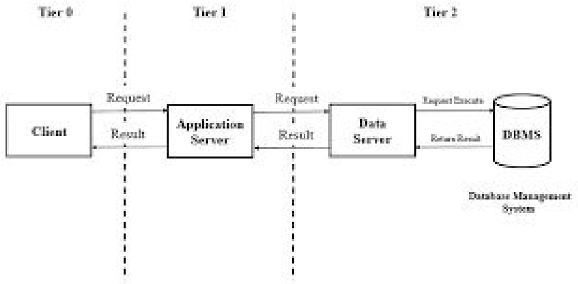

HTML5, JavaScript, CSS
Presentation Layer
Java, .NET, C#, Python, C++
Application Layer
MySQL, Oracle, PostgreSQL, SQL Server, MongoDB
Data Layer

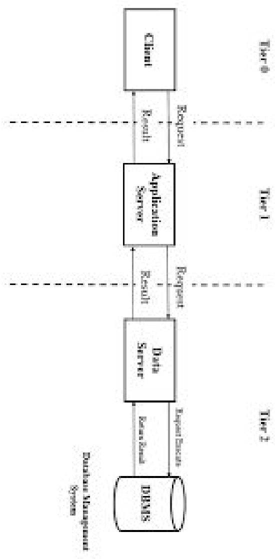


Layers do not need to be located on the same machine, when distributed across multiple machines the layers are often called tiers ➪ N-tier Architecture



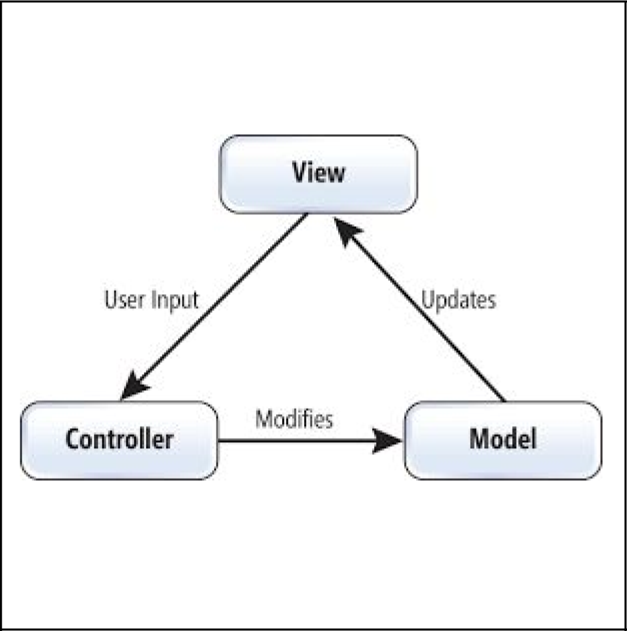
- Each ‘tier’ has a defined responsibility
- Requests can not skip layers, communicate only with immediately lower layer (and respond only to immediately upper layer)
- Higher layer treats lower layer as set of services
- Each layer “replaceable”
- No statement on physical deployment

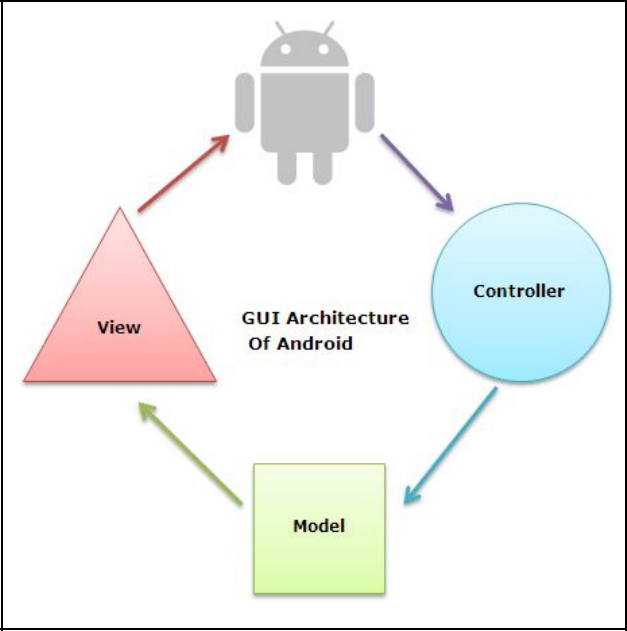


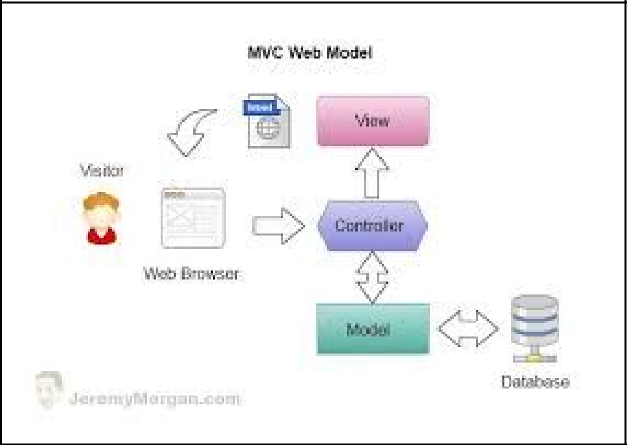


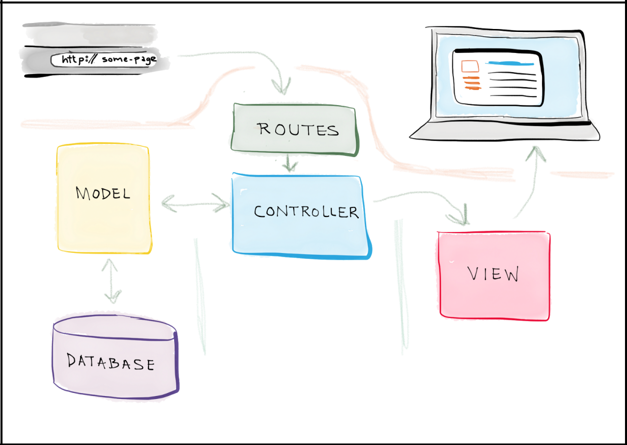
How to structure a GUI application is a very common problem but ‘‘Presentation” does not tell us enough, so there is a standard pattern

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









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

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 (methods) 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 = 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 and mobile versions of a web application
- 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 even as intermediary between user and view (renders/displays view for user to see)
- May be multiple controllers for same model: web vs. mobile

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

Separation of concerns:

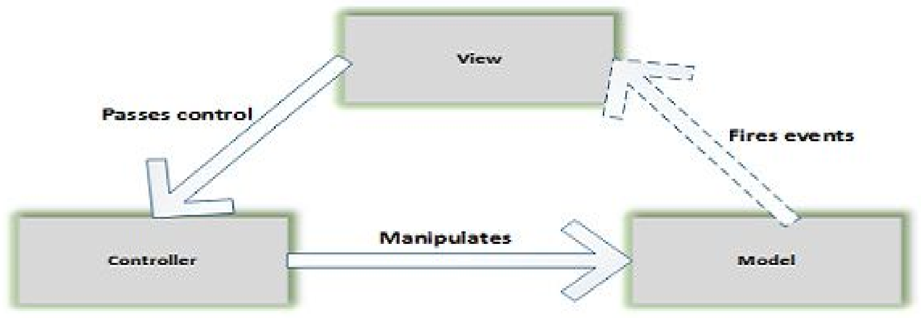
- Model is loosely coupled with the view and controller, knows nothing about either ⇨ *modularity*
- View and controller may be tightly coupled
- In OO languages usually implemented as two or three different classes that can be developed separately

Example: iTunes or similar music player

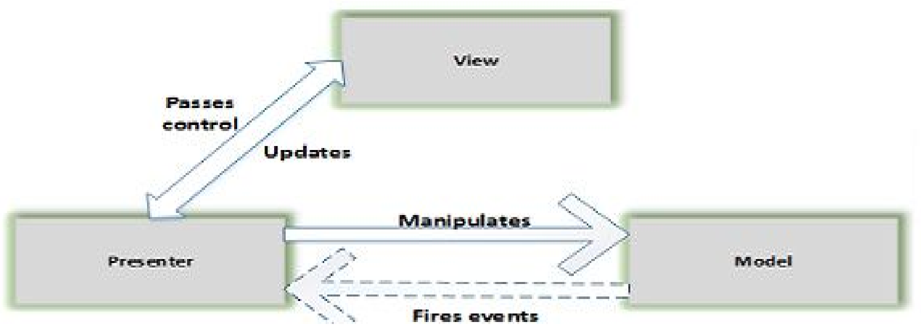
- View - playlists, current song, graphics, sound, video, etc., with some elements that accept user selection, action, data entry
- Controller - interprets the user actions applied to view and tells model what to do
- Model - stores content, knows how to play songs, shuffle, rip, and so on

Controversy as to what belongs in the model vs. in the controller, e.g., the model might be *only* the data (with possibly some bookkeeping methods) and the business logic resides in the controller, but then neither the model nor the controller is a full application by itself

Model View Controller



Model View Presenter



Minimize *Coupling* = interdependencies among program units, “need to know”

Maximize *Cohesion* = degree to which program unit exhibits a single purpose

Addendum added after class:

We briefly discussed in class when it might make sense to put the business (application) logic in the controller rather than in the model. I mentioned then, but did not write down until later:

I think the business logic belongs in the model when the data, and thus the model, is only part of this one application.

But when the data is shared across many applications, which is often the case for “enterprise” applications, then the model should only include bookkeeping operations. The application-specific business logic would then be in the controller, with a different controller (and presumably different view or set of views) for each such application.

Initial assignments for this course:

“Assignment I0”: https://courseworks2.columbia.edu/courses/104335/assignments/472886. It’s labeled 0 because it’s graded complete/non-complete and should be very easy. It’s labeled I for individual, a later series of assignments will be labeled T for team.

To do this assignment, you need to learn to use github (or already use github). Both the individual and team projects require submitting your code in a github repository. Many students include their github userid in their linkedin page and job applications so potential employers can see their work.

This assignment includes watching a video that explains the basics of version control using github, plus my survey indicated that most of you already use github, so I gave only a very brief introduction in Tuesday’s lecture.

“Assignment I1”:
https://courseworks2.columbia.edu/courses/104335/assignments/482592

To do this assignment, you need to code a simple web application in Java (a tic-tac-toe game). You have to use Java because the assignment provides skeleton code, in Java, for you to fill in - we provide the UI. The skeleton employs the model-view-controller (MVC) architecture.

The list of download/installs may seem overwhelming, so there will be an in-class tutorial going through all of that on Tuesday.

There will be two more individual programming assignments after this one, which will use JUnit to test your tic-tac-toe code and then add a database for saving moves during the game (and test it). There will also be in-class tutorials for the additional tools needed, each a week prior to the assignment deadline.

“Assignment T0”: https://courseworks2.columbia.edu/courses/104335/assignments/486855 This is the initial team assignment, to form a team and choose which programming language your team will use (C/C++, Java, Javascript, Python).

It’s not due until October 15, after the individual project and first assessment, but you can start forming teams any time.

There is a “search for teammates” thread in piazza that you can use to look for teammates https://courseworks2.columbia.edu/courses/104335/external_tools/1456

If time permits, at the end of each of the next few class sessions I will try to put everyone in breakout groups so you can introduce yourself and meet each other. Today (or next class if no time today) there is a short exercise to work on in your breakout groups, and then volunteers will report their group’s ideas to the class:

Design the functionality (not code) for a small drone that flies around an in-person classroom, just below the ceiling, and uses its ***sensors*** to determine which students are "paying attention" and which are not. It reports periodically to the instructor via a mobile app.

Its ***sensors*** include:

- High-resolution camera with very advanced image/video processing and face recognition
- Sensitive microphone with very advanced speech/sound processing and understanding
- Wifi monitoring - it can determine precise location of any devices in room that are actively sending/receiving data (but cannot read contents of encrypted data streams),
- Anything else you can think of that could realistically fit on a drone.

The advanced signal processing, NLP, ML, etc. occurs on backend servers, not the drone itself, and is super fast. Do not consider flight control, collision avoidance, energy management, bandwidth constraints, etc., just how it determines which students are not paying attention.

Ten minutes, then volunteer reports