Lecture Notes
November 27, 2018

Second Iteration due tonight

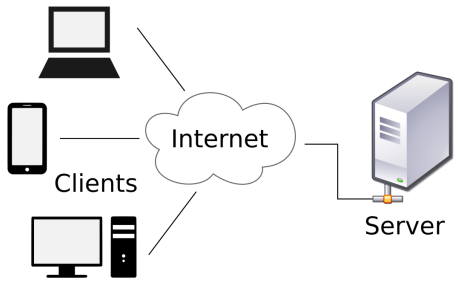
Second Iteration Demo due Tuesday December 4, 11:59pm

Design principles are centered on change, trying to make the code more resilient to change - more flexible, extensible, and maintainable

Requirements changes and refactoring may lead to changes in the *design*, hopefully abiding by design principles - micro level

Architecture is fundamental system structure that rarely changes - macro level

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

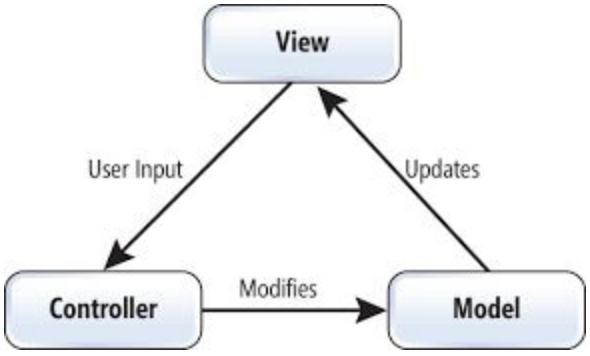


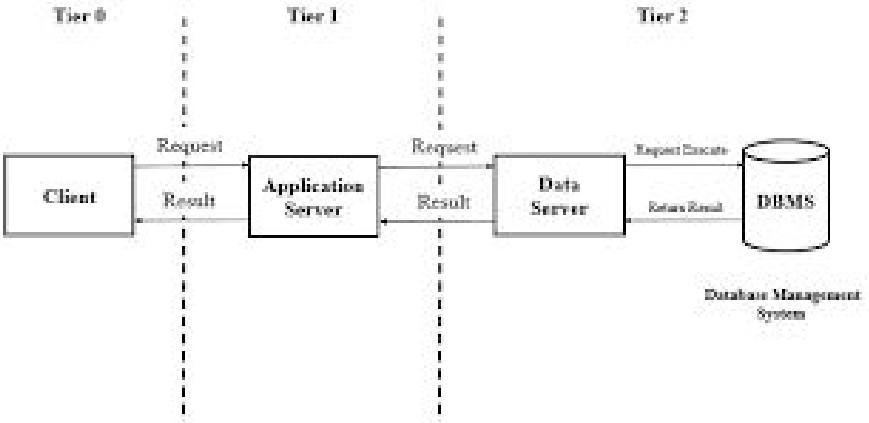
For most projects, the developers invent their own design. Should they also invent their own architecture?

Originality is overrated, imitation is the sincerest form of not being stupid

Almost all software adopts a well-known "pattern" at the architectural level - often called architectural style rather than architectural pattern

Which of these architectural styles do you think is better for web applications?





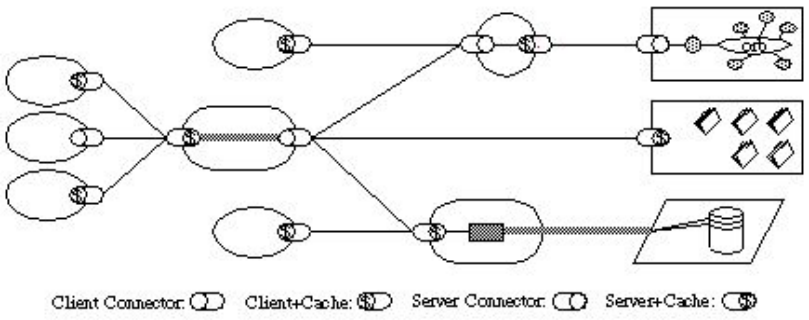


Figure 5-7. Uniform-Layered-Client-Cache-Stateless-Server

"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

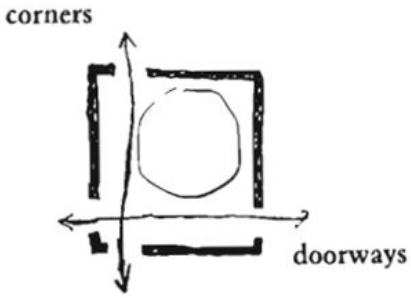
Christopher Alexander, who popularized the concept of architecture and design patterns, was an architect (as in buildings), not a computer scientist

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.

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?

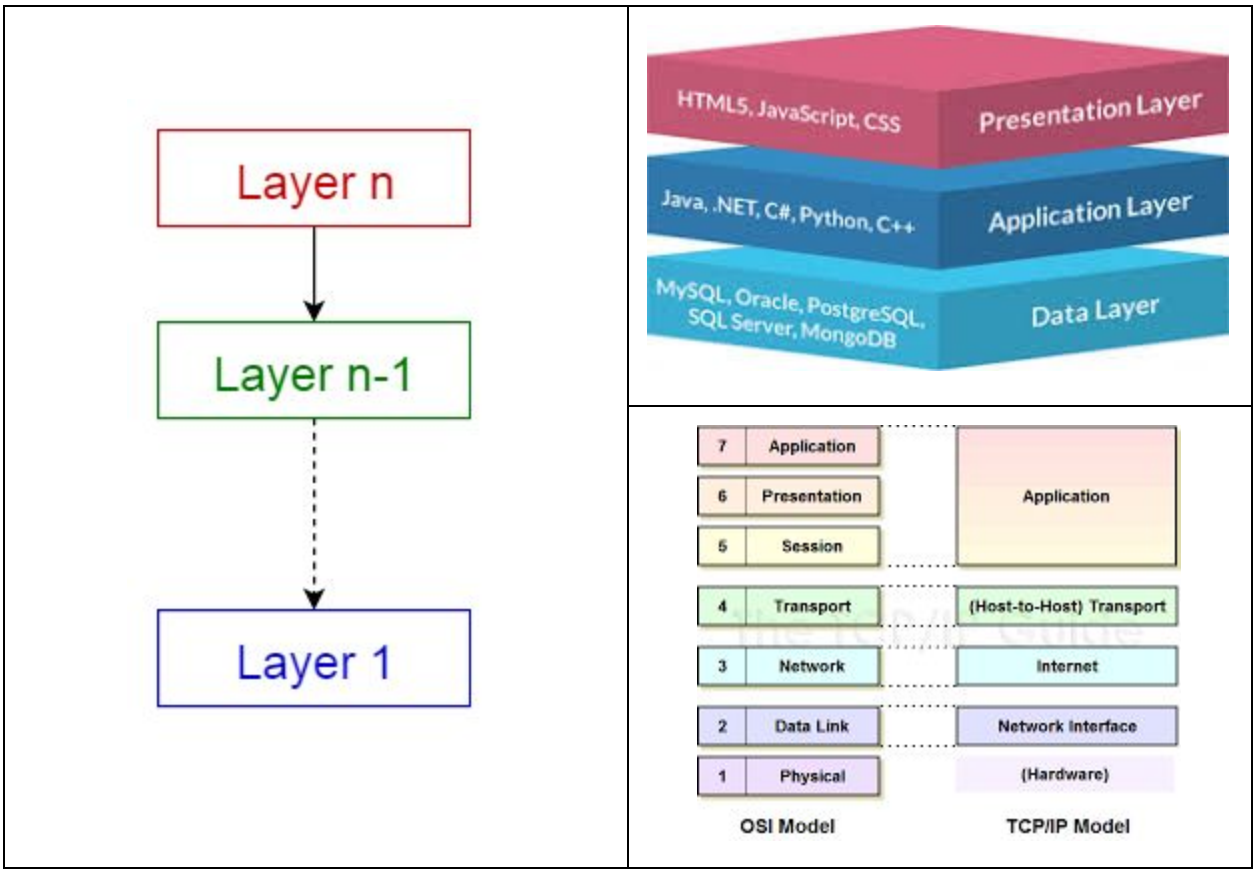
Alexander identified four elements to describe an architecture pattern:

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

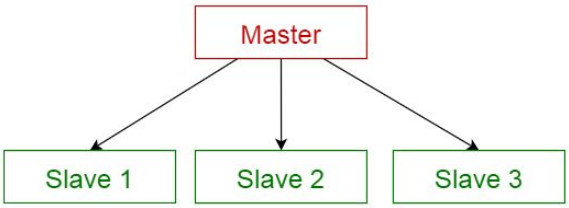
Benefits of patterns in software engineering

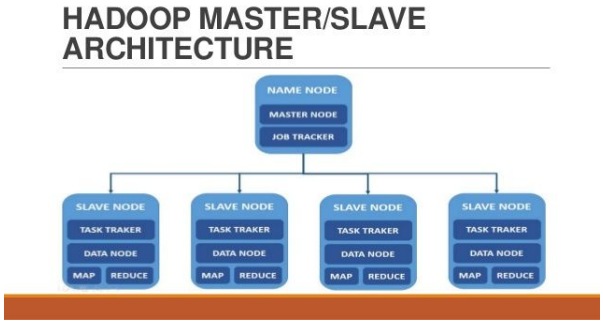
- Reuse well understood and tested architectures and designs
- Save time and don't reinvent the wheel
- Communication language among software engineers

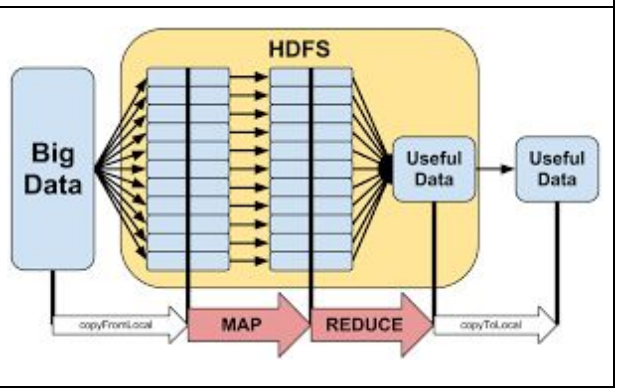
Layered pattern - each layer provides services to the next higher layer and implements those services using the next lower layer



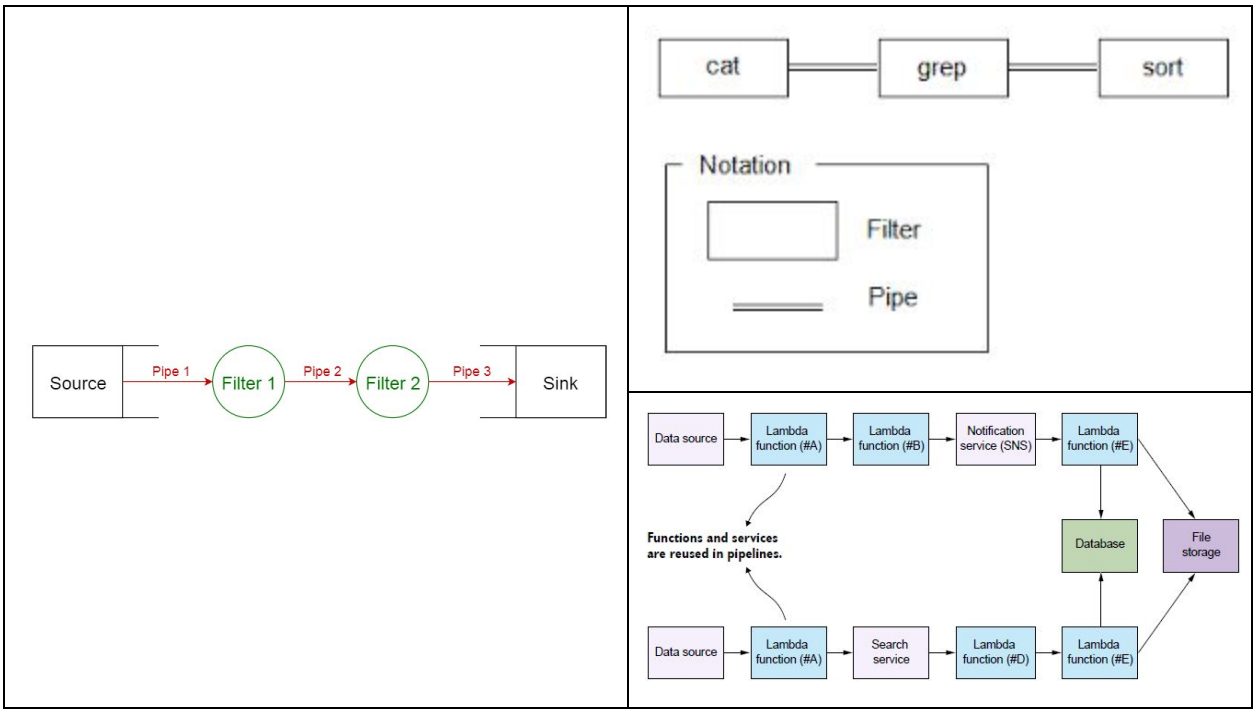
Master-slave pattern - Master divides work among many identical slaves, computes final results



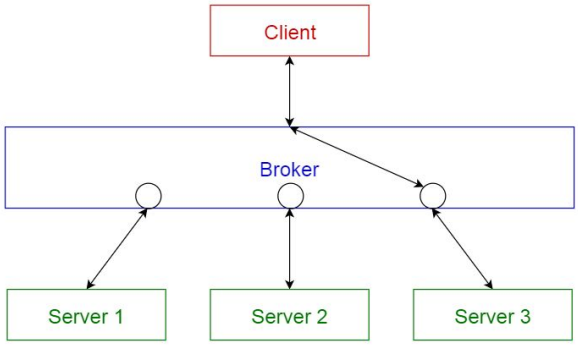


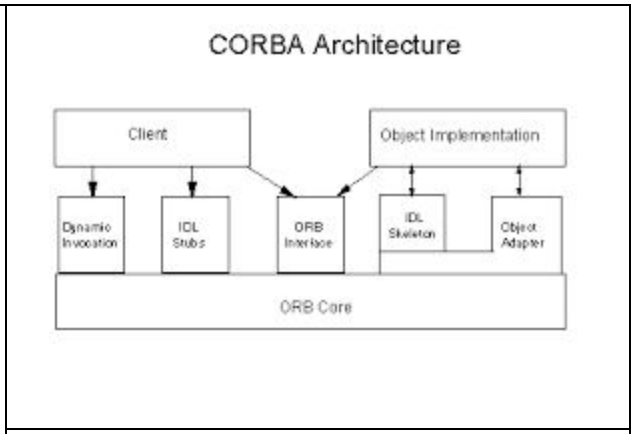


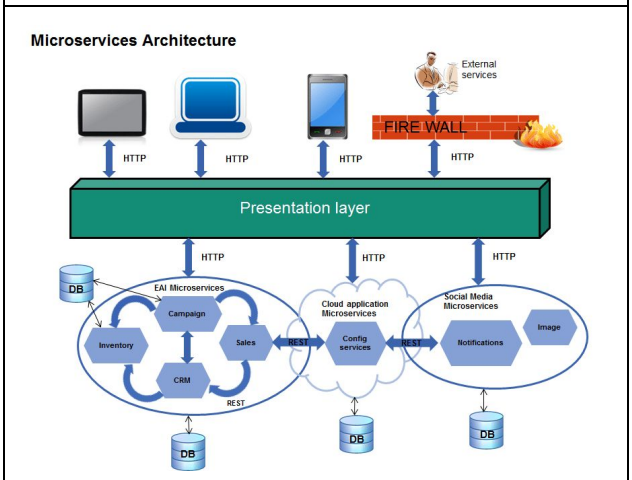
Pipe and Filter Pattern - processes stream of data, each processing step enclosed in filter, pipes buffer and synchronize data passing between filters



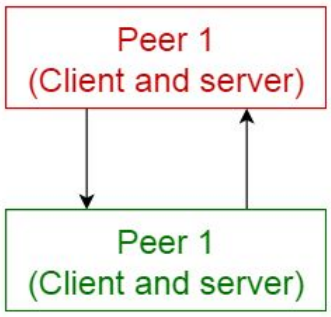
Broker pattern - Servers publish their capabilities to broker, Clients request service from broker, broker redirects client to suitable service and coordinates communication

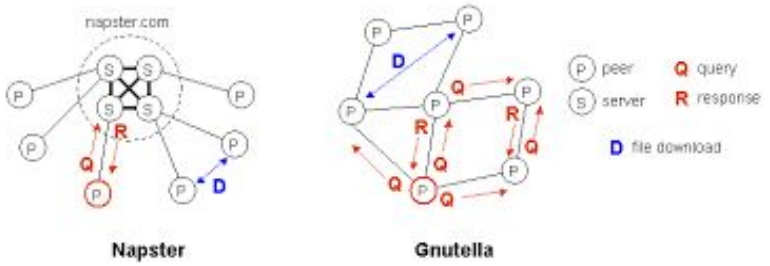


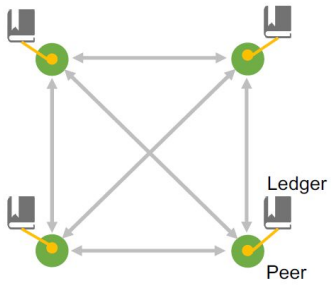




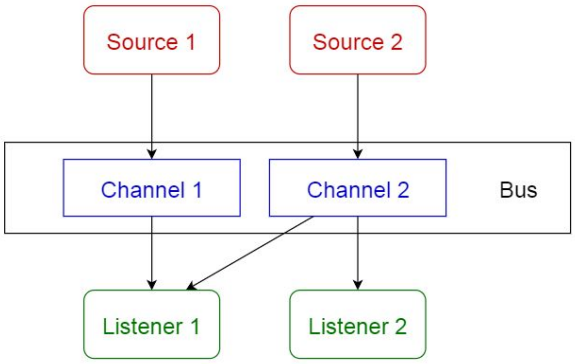
Peer to Peer pattern - Peers function both as client, requesting services from other peers, and as server, providing services to other peers

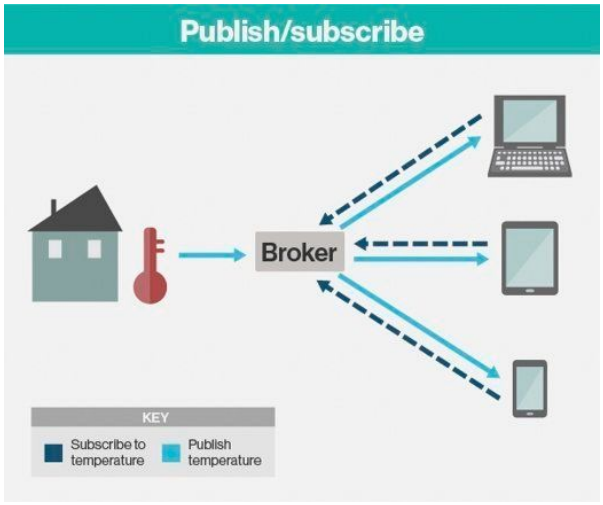


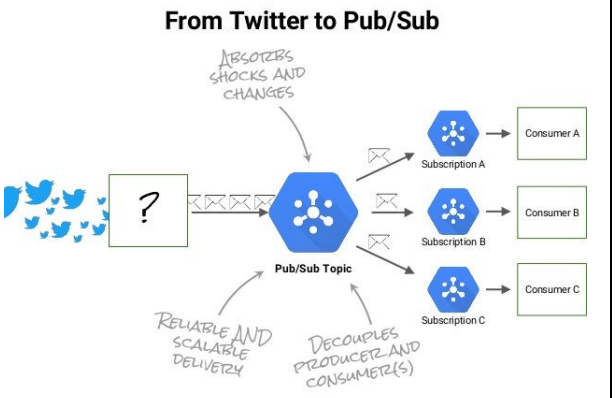




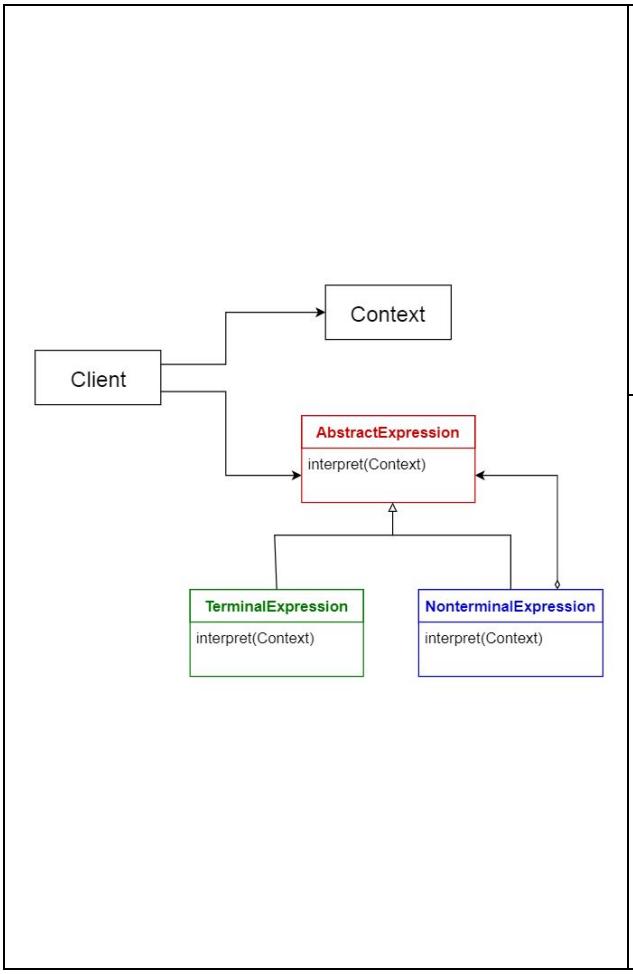
Event-bus pattern - Sources publish messages to particular channels on an event bus, Listeners subscribe to particular channels and are notified of messages that are published to that channel

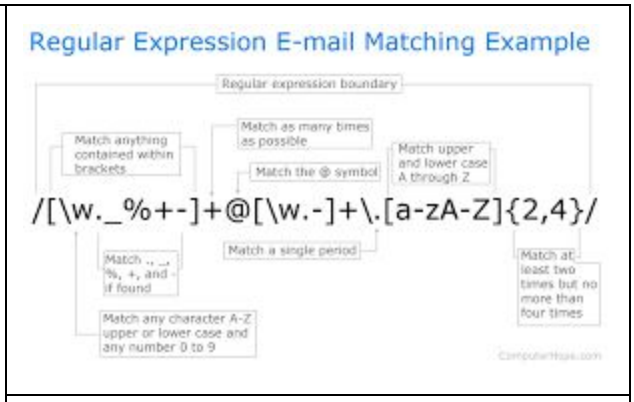


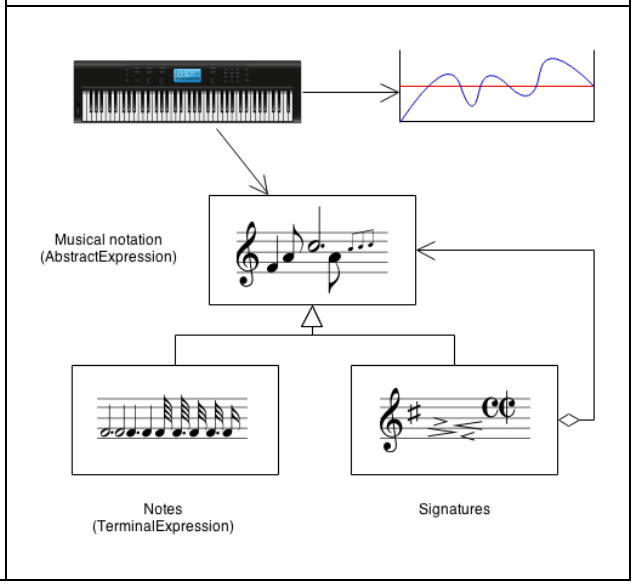




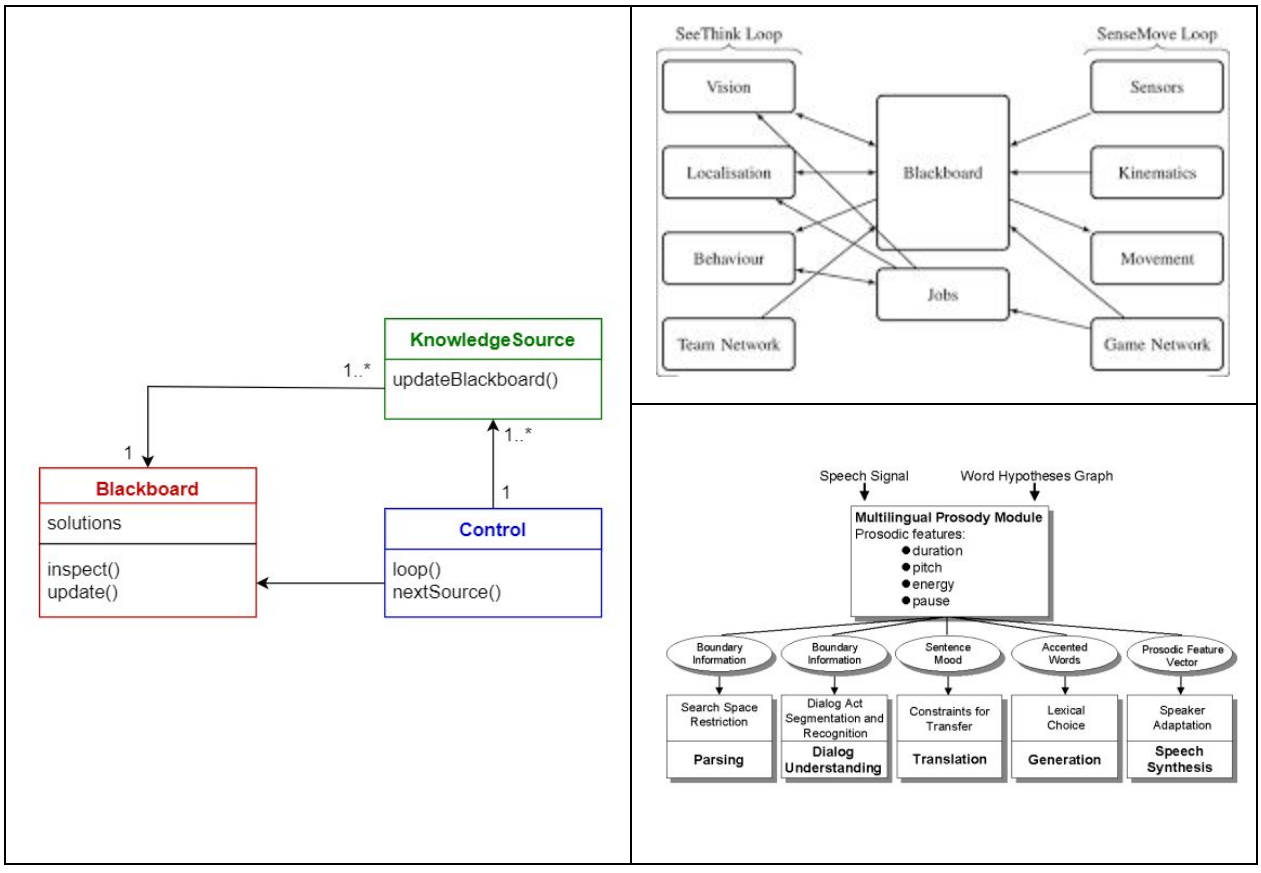
Interpreter pattern - Recursive-descent interpreter for domain-specific language (DSL). Every symbol in language corresponds to non-terminal, to be expanded, or terminal end-value



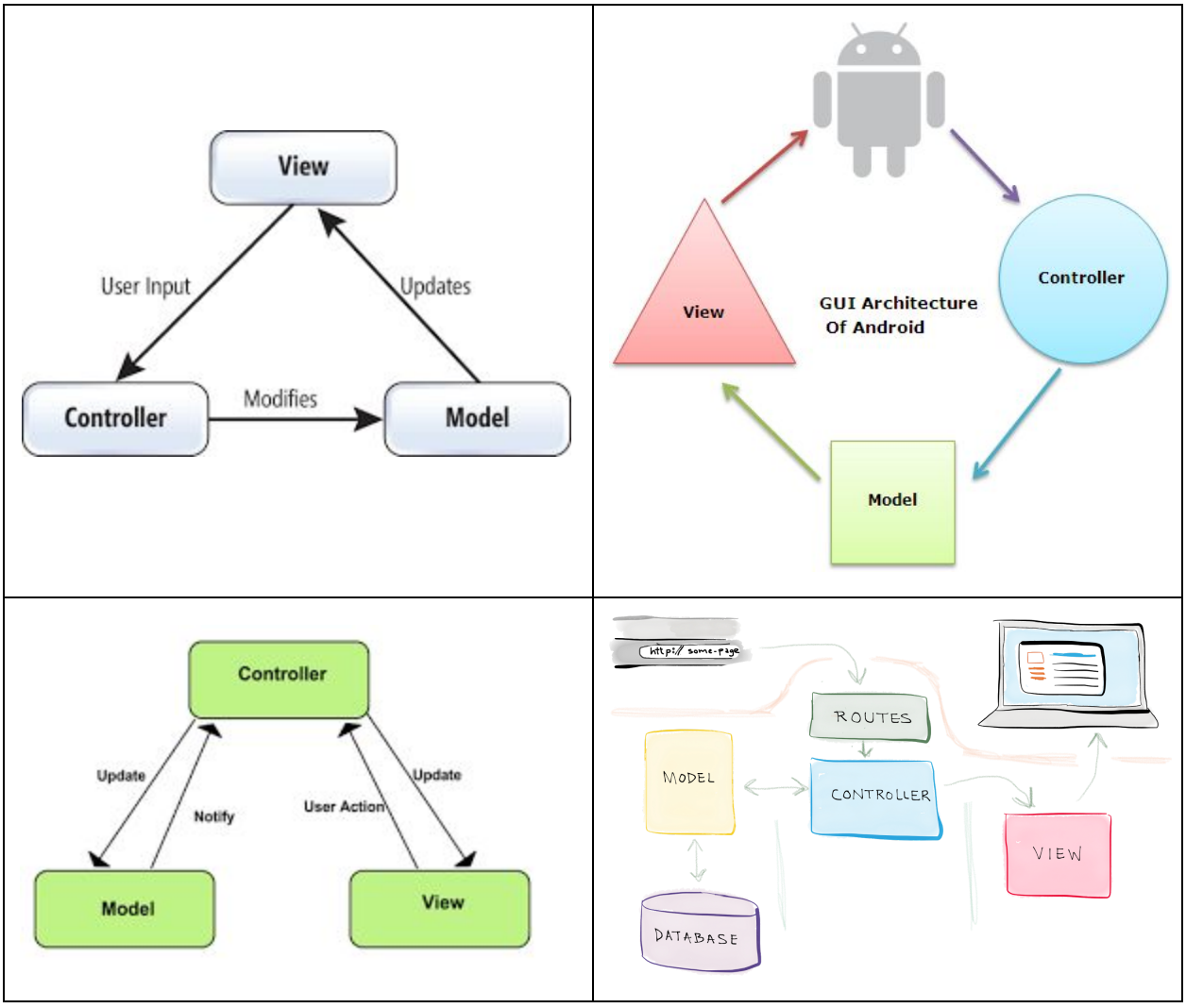




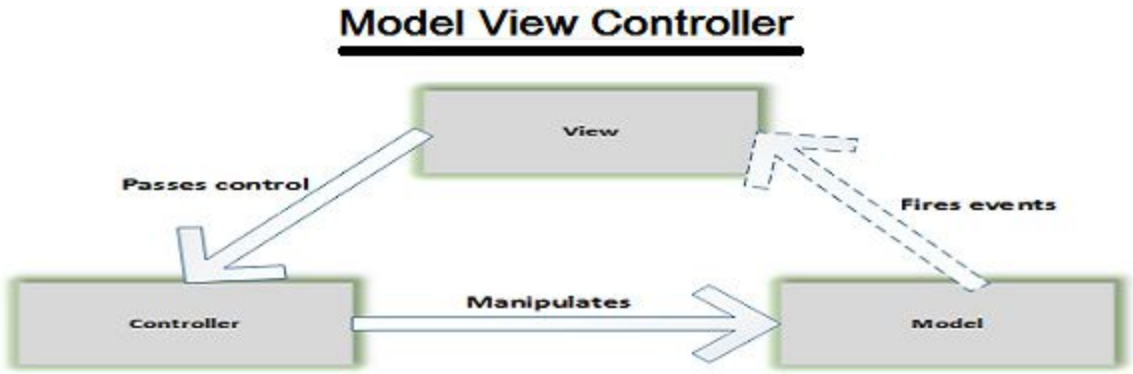
Blackboard or data-centered pattern - Blackboard (data) is a shared global memory, each Knowledge source is a specialized module that reads from and writes to global memory as part of their internal computations, Control decides which knowledge sources to consider and puts together results

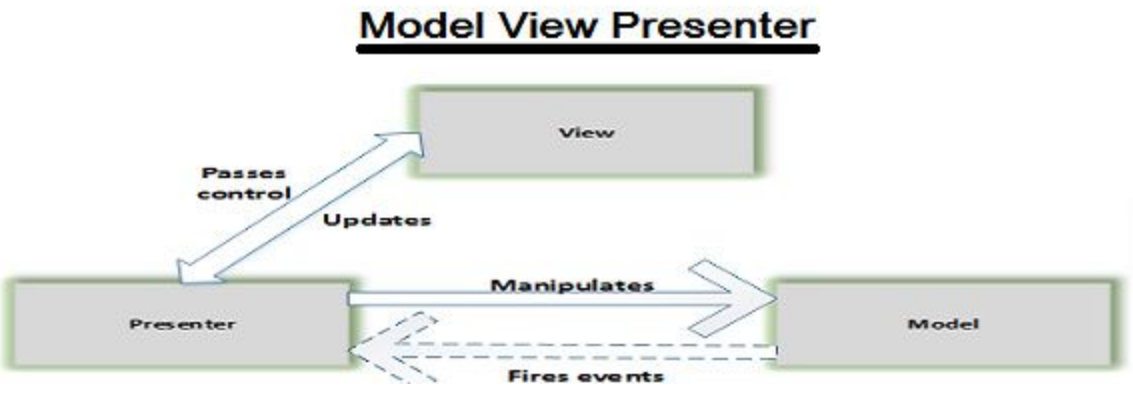


Model-View-Controller pattern - Model contains the core functionality and data, View displays the information to the user, Controller handles user input and directs the model



Many different MVC variants, disagreement over whether model and view communicate directly or all interactions go through controller. Some variant is used for nearly all GUI applications





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, often includes business logic that manipulates state and does the work - ideally a fully functional application program (no GUI)

- 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)
- Responds to instructions to do something or change state (usually from the controller)

View = GUI output: all CSS, HTML, etc. 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, again such as web vs. mobile

Reading for Thursday: design patterns cookbook

Second Iteration due tonight, 11:59pm

Second Iteration Demo due Tuesday December 4, 11:59pm

Second Individual Assessment will become available on December 4

We went over the first individual assessment on Tuesday. If you weren't here, watch the CVN video