Lecture Notes
October 12, 2017

Requirements and Wireframes team assignment due tonight

Design team assignment due next Thursday October 19

Full set of first iteration team assignments posted

Next week review on Tuesday and first exam on Thursday

Exam will cover all required readings and all lectures through today

Read the readings!

Software Architecture

Requirements often change during development

This may lead to changes in the design during development, (hopefully) abiding by design principles - micro level

Architecture is *fundamental* system structure that does not (or should not) change - macro level

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 ourselves, someone else already judged whether it is good, but we still need to judge whether it is the 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 towns, in cities

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

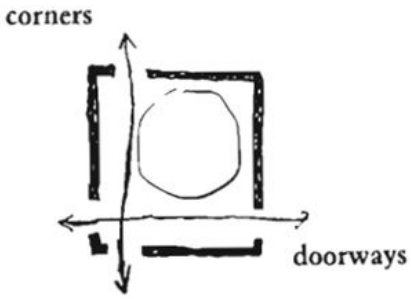
On door placement:

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 the doors?

Alexander identified four elements to describe a pattern:

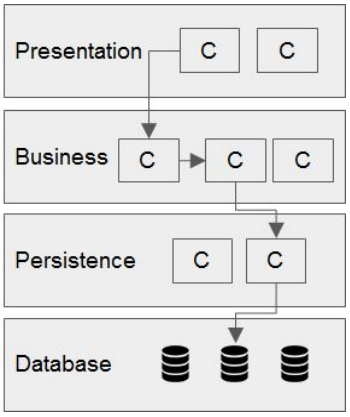
1. Name of the pattern
2. Purpose of the pattern, what problem it solves
3. How to solve the problem
4. Constraints to consider in our solution

Benefits of patterns in software engineering

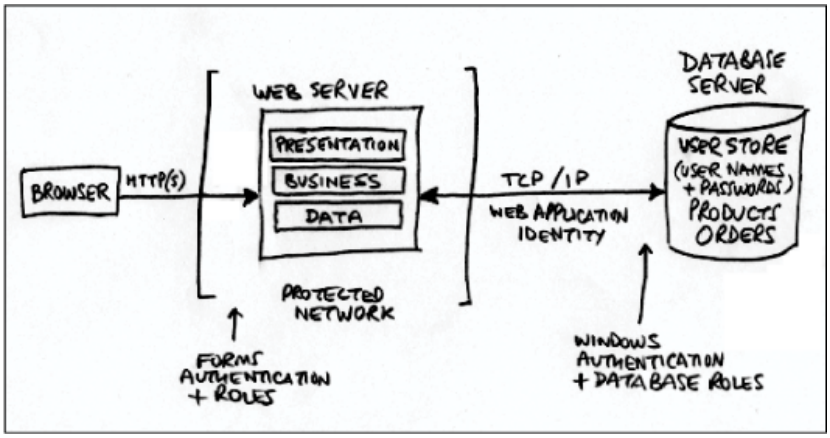
- Reuse well understood and tested architectures and designs
- Save time and don't reinvent the wheel
- Communication language among software engineers

Classic software architectural styles discussed at
https://msdn.microsoft.com/en-us/library/ee658117.aspx

Example classic architecture pattern: Layered (3-tier, N-tier)

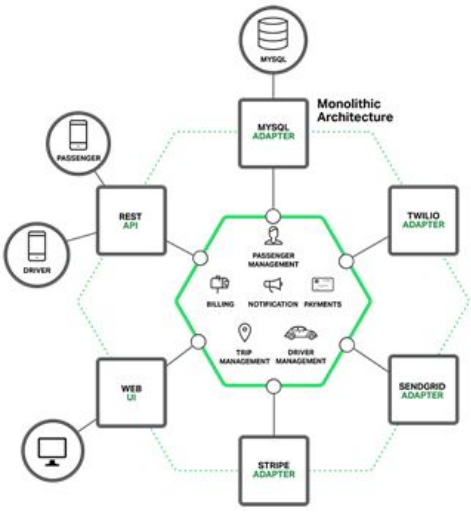


- Each 'tier' has a defined responsibility
- Requests can not skip layers, communicate only with immediately lower layer (and respond only to immediately upper layer)
- Higher layer treats lower layer as set of services
- Each layer "replaceable"
- No statement on physical deployment



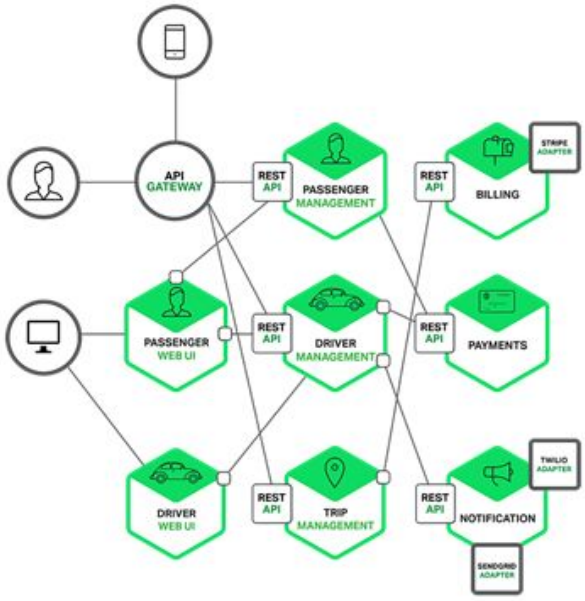
“Modern” architecture styles

Monolithic



- APIs from application to independent external services rather than bundled layers
- Simple to begin with
- Hard to scale
- Hard to change over time

Microservices



- Application itself componentized into internal services
- Services represent business functions
- Business organization alignment to architecture, separation of (human) management
- Associated with high availability and scalability architectures

There are patterns at both the macro architectural level (often called styles instead of patterns) and at the micro design level - the “design patterns” in the Gang of Four book are micro level

> There are books and websites the provide examples of the GoF (or similar) design patterns for nearly every language, e.g., python-patterns

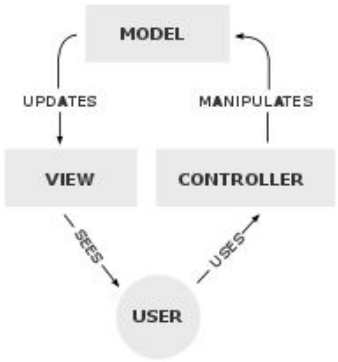
Besides many developers discovering the same good solution = “pattern”, many developers may also discover the same really bad non-solution = “anti-pattern”

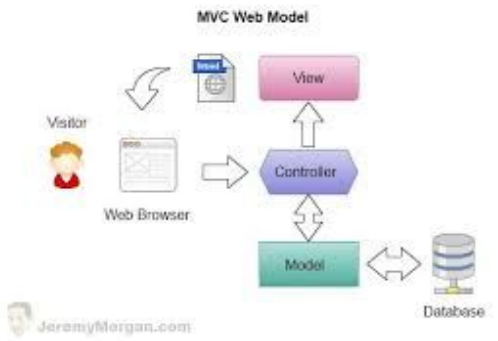
> See **The Little Book of Python Anti-Patterns**

How to structure a GUI application is a very common problem

So of course there is a standard architecture pattern for GUI applications: Model-View-Controller (MVC), built into most web and mobile application frameworks

Model View Controller





Model = application data, not possible to alter data except through the model, often includes business logic that manipulates application state and does the work - ideally the model is a fully functional application program (with no GUI)

> 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)
>
> 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, passive observer does not affect the model

> May be multiple views of same data such as web and mobile versions of a website
>
> Push - view registers callback 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

Separation of concerns

> Model is loosely coupled with the view and controller, knows nothing about either
>
> View and controller may be tightly coupled
>
> In OO languages usually implemented as three different classes, although sometimes view and controller are combined or view receives user input and passes to controller

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

Modularity - changes to one don't affect the others, can develop in parallel

Example: iTunes or similar music player

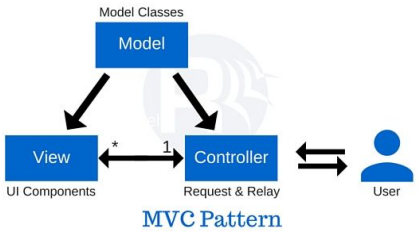
> View - playlists, current song, graphics, sound, video, etc., with some elements that accept user selection, action, data entry
>
> Controller - interprets the user actions applied to view and tells model what to do
>
> Model - stores content, knows how to play songs, shuffle, rip, and so on

Controversy as to what belongs in the model vs. in the controller, e.g., in some variants, the model is *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

Different skills sets for GUI (view) vs. business logic, independent development - this is an argument for including only data in the model, since data analysis, data schema design, SQL queries, etc. another skill set distinct from business logic

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

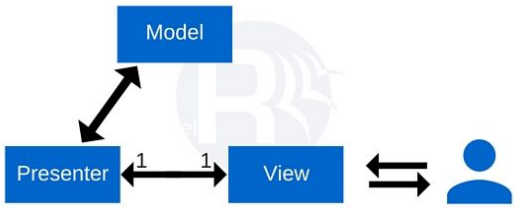
Contrast to Cohesion = degree to which program unit exhibits a single purpose (SRP)



MVC Pattern

Some alternatives to MVC (primarily server-side):

MVP = Model View Presenter (client-side)

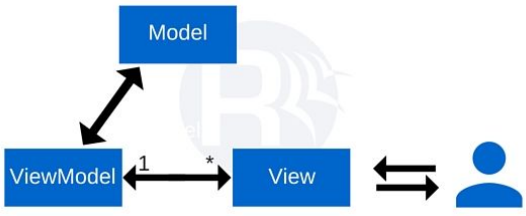


MVP Pattern

Presenter replaces controller, all GUI I/O through view with presenter intermediating with model

> Enables complete separation between presenter and view, through interfaces, whereas controller needs to handle user requests and know exactly what view displays

MVVM = Model View ViewModel (client-side)



MVVM Pattern

Similarly, ViewModel replaces controller, all GUI I/O through view with ViewModel intermediating with model

Different from Presenter because uses two-way data binding wrt views rather than API

Distinctions for Android apps discussed at
https://academy.realm.io/posts/eric-maxwell-mvc-mvp-and-mvvm-on-android/