System Design problems

**SOLID OOD Principal:**

**S**– **S**ingle Responsibility Principle | Single responsibility to each Object

**O**– **O**pen/Closed Principal  | On production level Objects are ready for extension for not for modification

**L**–**L**iskov Substitution Principal | Base Class and Derived class follow ‘IS A’ principal

**I** – **I**nterface segregation principle | If an implementation don’t require then don’t implement it.

**D-D**ependency Inversion principle | Reduce the dependency In composition of objects.

General Steps For OOD System Design:

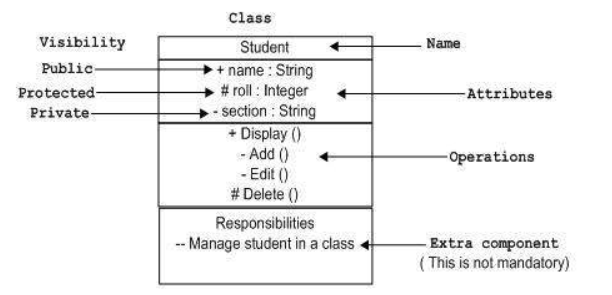
Now, here are some steps to follow to answer an object-oriented design question like design an object-oriented parking lot.

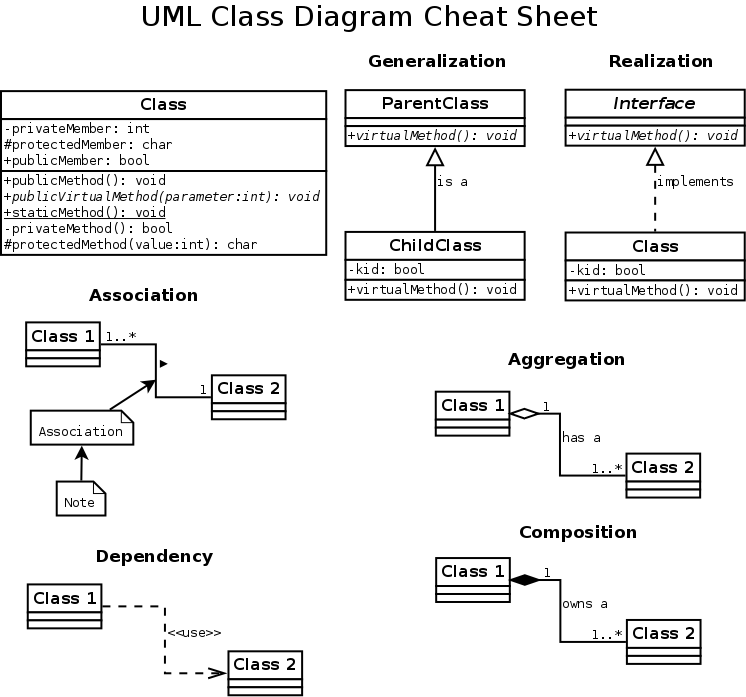
1. Identify the **major entities** like vehicles, parking spaces, parking structure and so on using interfaces.
2. Identify **specialized subtypes of these entities** using abstract classes or just regular classes. For example, your parking spaces may be of different types, two-wheeler parking spots, car parking spots, handicapped parking spots and so on.
3. Determine the **functionality(behaviour)** of each of these entities, such as a car should have a license plate, and have the ability to move; a parking structure should be able to tell you the next available spot to park; a parking space should be able to keep track of your cost of parking and so on.
4. Determine the functionality of specialized entities. All parking spaces have same properties like indicating whether they are occupied or not, and calculating the cost of parking your vehicle but two wheeler spots are cheaper than four wheeler spots.
5. **Setup relationships between entities** that communicate with each other. Parking space and parking structure can be connected association, that is a parking structure has many parking spaces, and of different types.
6. Lastly, to make yourself standout, **use design patterns where appropriate**. The most obvious scenario is, obviously not everyone should be able to create their own parking structure on the fly. Make it a singleton class so there is only parking structure, and vehicle owners will use it using your public methods.

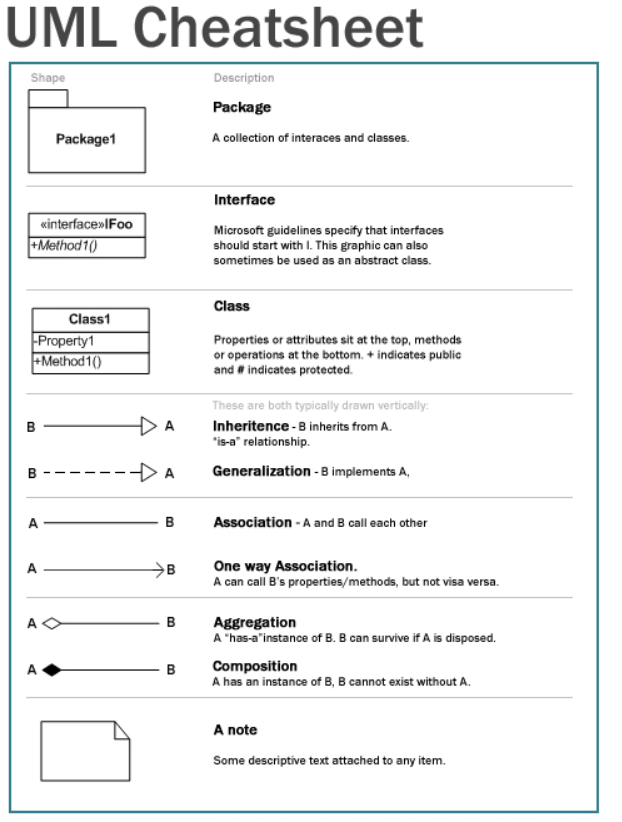
Steps for System Design in general:

1. Get the system details, requirements. Ask clarifying questions, features…
2. First write down/sketch a high-level system design - architecture, big picture. Don’t dive into technical details. Just high-level design – LB, servers, DB, caching, scalability etc
3. Low level system design: DB schema, UML diagrams, OOD class and relationships, API design, etc

UML Notations:







**1) Elevator design**

Ask possible questions about the entities – about elevator, its building, no of ppl, floors, features, buttons etc

Entities: Elevator, User, Button, ElevatorController, States, Building.

User

• Presses the floor button to call the lift

• Presses the elevator button to move to the desired floor

Floor Button & Elevator Button

• Illuminates when pressed by user

• Places an elevator request when pressed

Elevator

• Moves up/down as per instruction

• Opens/closes the door

ElevatorController

* Centralized control for elevators
* Will have a queue for requests from users
* Will dispatch the request to nearest lift using some scheduling algorithms

State

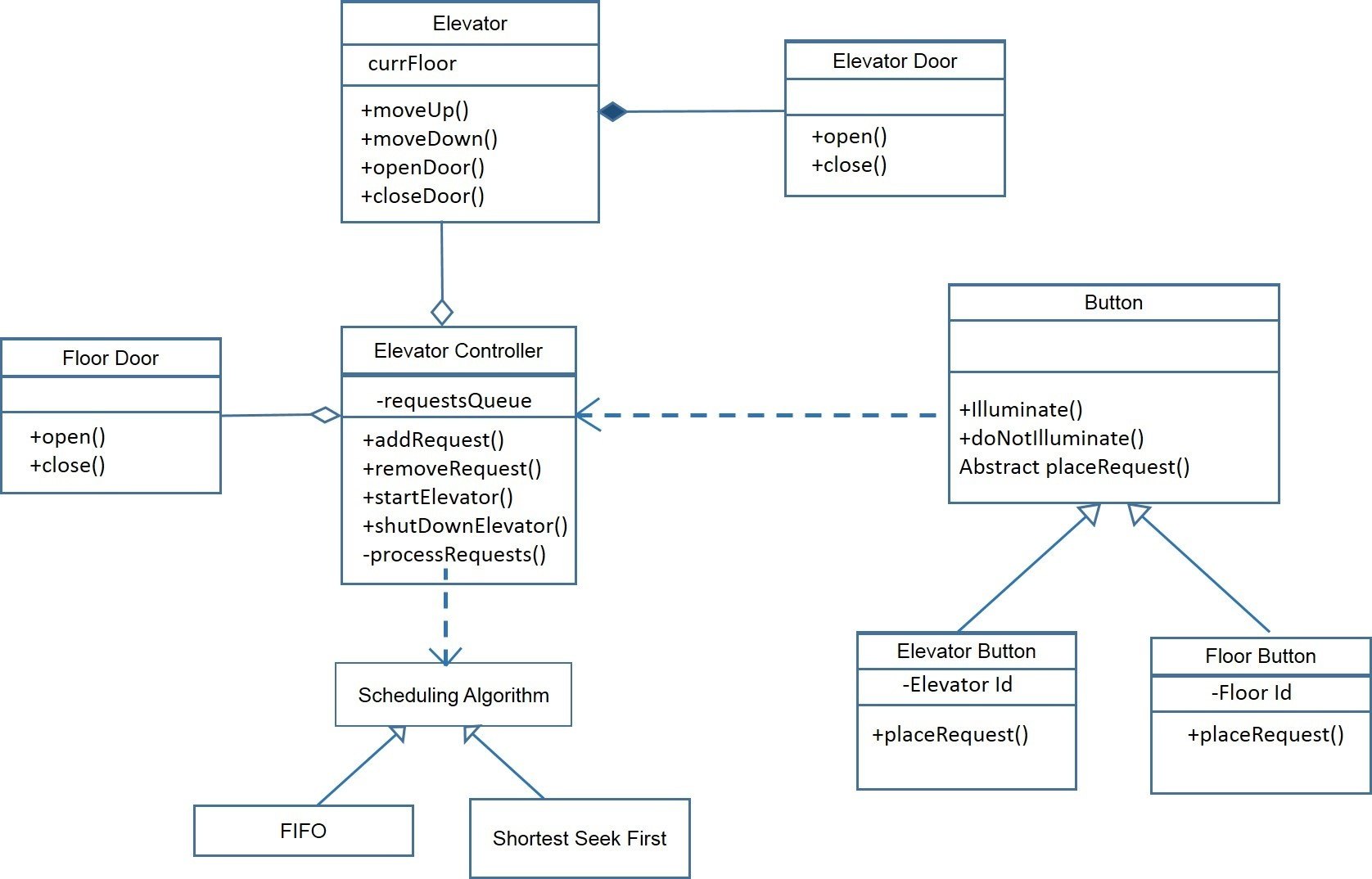
* Idle
* Going UP
* Going Down
* DoorOpening
* DoorClosing
* Beeping

Inheritance…

* Create interface IElevator – methods like goUP, goDown, Opendoors, closeDoors..
* Create an abstract class BaseElevator … all properties , general method definitions…abstract method handlerequest()..
* Create specific Elevator implementations like PassengerLift, ServiceLift… They will extend BaseElevator and implement IElevator..
* IEventController…interface

Design:

1. Each button press results in an elevator request which has to be served. Each of these requests is tracked at a centralized place. Elevator Requests, the class that stores, elevator requests can use different algo to schedule the elevator requests. The elevator is managed by a controller class, which we call Elevator Controller. Elevator controller class provide instructions to the elevator. Elevator controller reads the next elevator request to be processed and served.
2. The button is an abstract class defining common behavior like illuminate, doNotIlluminate. FloorButton, Elevator Button extends Button type and define placeRequest() method which is invoked when a button is pressed.
3. When a floor button or elevator button is presses a requests is added to a common queue.ElevatorController reads the next request and instruct next action to the elevator.
4. FloorButton's placeRequest() adds a request to the common queue, which is accessed by the elevator controller thereby assigning the request to one of the elevators.
5. ElevatorButton's placeRequest adds a request to the elevator directly as it is supposed to serve it.
6. Elevator controller will be running various algorithms like shortest seek etc. to decide which lift is supposed to handle which request.



Algorithm:

But in the simple steps, the algorithm can be followed as:

1. Proceed in the same direction until the last request in that direction.
2. If there is no request, stop and proceed towards other direction, if there is any request from other direction.

Links: <https://www.quora.com/What-is-the-best-answer-to-Design-an-elevator-system>

<https://www.quora.com/How-would-you-design-an-elevator>

Twitter Design:

*“Design a simplified version of Twitter where people can post tweets, follow other people and favorite\* tweets.”*

1) Ask Clarifying questions:

**How many users** do we expect this system to handle? Our interviewer says:

*“well… to make things interesting, let’s aim for 10 million users generating around 100 million requests per day”*

**Connections?** Since we have the notion of following someone, how connected will these users be? After all, they will form a graph with the users being the nodes and the edges will represent who follows whom.

*“we expect that each user will be following 200 other users on average, but expect some extraordinary users with tens of thousands of followers”*

**Tweets**: “Hm, that’s hard to say… we expect that there will be a maximum of 10 million tweets per day and each tweet will probably be favorited twice on average but again, expect some big outliers.”

Let’s make a few very simple calculations with the information we just received. We will have around 10 million users. Average number of followed other users is 200. This means that the network of users will have about 200 \* 10 million edges.

**Ask about availability and response time:** In addition to the expected size of different things, for a system like that it will be important to know what availability is expected and what response times are tolerable. Naturally, our interviewer wants to have a system, which loads pretty quickly. None of the operations described above should take more than a few hundred milliseconds. The system should be online all of the time

#### Requirements gathering: To summarize, here are some more things we now know:

* 10 million users
* 10 million tweets per day
* 20 million tweet favorites per day
* 100 million HTTP requests to the site
* 2 billion “follow” relations
* Some users and tweets could generate an extraordinary amount of traffic

**2) High level Design:**

we can divide our architecture in two logical parts: 1) the logic, which will handle all incoming requests to the application and 2) the data storage that we will use to store all the data that needs to be persisted.

a) logic: our application will need to handle requests for:

* posting new tweets
* following a user
* favoriting a tweet
* displaying data about users and tweets

The first three operations require things to be written somewhere in our database, while the last is more about reading data and returning it back to the user.

Therefore our architecture should allow the app to handle at least a few thousand requests per second at times.

In our particular problem we would definitely suggest using a load balancer, which handles initial traffic and sends requests to a set of servers running one or more instances of the application.

Behind the load balancer we will be running a set of servers that are running our application and are capable of handling the different requests that arrive.

b) Storing the data:

. First of all, users have profiles with some data fields attached to them. We’ll need to store that. Each user has a set of tweets that they have produced over time.

*“Sounds good, but could you tell me more about why reading from the cache would be better than just reading from the database?”*

 we could say that a database stores data on disk and it is much slower to read from disk than from memory. A solution like memcached stores data in memory, which provides way faster access.

Low level System design:

a) Database Schema

*“If you’re going to use a relational database for storing all the data could you draft the tables and the relations between them?”*

We have two main entities: users and tweets. There could be two tables for them.

#### Table users

* ID (id)
* username (username)
* full name (first\_name & last\_name)
* password related fields like hash and salt (password\_hash & password\_salt)
* date of creation and last update (created\_at & updated\_at)
* description (description)
* and maybe some other fields...

#### Table tweets

* ID (id)
* content (content)
* date of creation (created\_at)
* user ID of author (user\_id)

These two entities have several types of relations between them:

1. users create tweets
2. users can follow users
3. users favorite tweets

The first relation is addressed by sticking the user ID to each tweet. This is possible because each tweet is created by exactly one user. It’s a bit more complicated when it comes to following users and favoriting tweets. The relationship there is many-to-many.

For following user we can have a table like that:

#### Table connections

* ID of user that follows (follower\_id)
* ID of user that is followed (followee\_id)
* date of creation (created\_at)

Let’s also add a table, which represents favorites. It could have the following fields:

#### Table favorites

* ID of user that favorited (user\_id)
* ID of favorited tweet (tweet\_id)
* date of creation (created\_at)

### **Building a RESTful API**

The API endpoints will likely be built around the data entities that we have and the needs of the user-facing part of the application.

**GET /api/users/<username>**

**GET /api/users/<username>/tweets**

**GET /api/users/<username>/tweets?page=4**

**GET /api/users/<username>/followers**

**GET /api/users/<username>/followees**

**POST /api/users/<username>/tweets**

**GET /api/users/<username>/tweets/<tweet\_id>/favorites**

<https://www.hiredintech.com/classrooms/system-design/lesson/67>

<https://www.quora.com/What-are-some-of-the-best-answers-to-the-question-How-would-you-design-Twitter-in-a-system-design-interview>