OOD chapter 1

* 评判OOD面试
  + S – Single responsibility principle

Text

Description automatically generated

* + O – Open close principle

Graphical user interface, text, application

Description automatically generated

* + L – Liskov substitution principle

Text

Description automatically generated

* + I – Interface segregation principle

Text

Description automatically generated

* + D – Dependency inversion principle

Graphical user interface, text

Description automatically generated

**5C解题法**

* Clarify: 通过和面试官交流，去除题目中的歧义，确定答题范围
* Core objects: 确定题目所涉及的类，以及类之间的映射关系
* Cases: 确定题目中所需要实现的场景和功能
* Classes: 通过类图的方式，具体填充题目中涉及的类
* Correctness: 检查自己的设计，是否满足关键点

**Design Elevator System**

**Clarify**

* Overview
  + What: 针对题目中的**关键字**来提问, 通过名词的属性来考虑
  + How: 针对问题主题的**规则**来提问
  + Who: 以**系统**为主导
* What - 关键词
  + Elevator:
    - 电梯能够获取当前重量?
    - 客梯和货梯?
  + Building
    - 是否有多处能搭乘的电梯口?
* How - 规则
  + 当按下按钮时，哪一台电梯会相应
  + 当电梯在运行时，哪些按键可以响应
* Who
  + 电梯系统如何获取每位乘客的重量

**Core Object**

* 为什么要定义Core Object ?
  + 这是和面试官初步的纸面contract
  + 承上启下，来自于Clarify的结果，成为Use case的依据
  + 为画类图打下基础
* 如何定义Core Object ?
  + 以一个Object作为基础，**线性**思考
    - 只考虑input/output
  + 确定Objects之间的映射关系

Diagram

Description automatically generated

* Good Practice
  + Use Access modifier
    - package - 尽量避免
    - public
    - private - 封装

Graphical user interface, text, application, email

Description automatically generated

* + - protected - 继承
  + package
    - 如果什么都不声明，变量和函数都是package level visible
    - 在同一个package内的其他类都可以访问
    - 在类图中，避免使用default的package level access
    - 常用于写Unit Test

**Cases**

* 为什么要写Use cases
  + 这是你和面试官白纸黑字达成的第二份共识，把你将要实现的功能列在白板上
  + 帮助你在解题过程中，理清条例，一个一个Case实现
  + 作为检查的标准
* 怎么写Use cases
  + 利用定义的Core Object, 列举出每个Object对应产生的use case.
  + 每个use case只需要先用一句简单的话来􏲼述即可
* ElevatorSystem
  + Handle request
* Request
  + N/A
* Elevator
  + Take external request
  + Take internal request
  + Open gate
  + Close gate
  + Check weight
* ElevatorButton
  + Press button

**Class**

* 为什么要画类图?
  + 可交付，Minimal Viable Product
  + 节省时间，不容易在Coding上挣扎
  + 建立在Use case上，和之前的步骤层层递进，条例清晰，便于交流和修改
  + 如果时间允许/面试官要求，便于转化成Code
* 怎么画类图?
  + 遍历你所列出的use cases
  + 对于每一个use case，更加详细的描述这个use case在做什么事情
    - 例如:take external request
      * ElevatorSystem takes an external request, and decide to push this request to an appropriate elevator
  + 针对这个描述，在已有的Core objects里填充进所需要的信息
* Use case: Handle request
  + ElevatorSystem takes an external request, and decide to push this request to an appropriate elevator

**Correctness**

* 从以下几方面检查:
  + Validate use cases (检查是否支持所有的use case)
  + Follow good practice (面试当中的加分项，展现一个程序员的经验)
  + S.O.L.I.D
  + Design pattern

**Challenge**

* Q: What if I want to apply different ways to handle external requests during different time of a day?
* A: Use Strategy design pattern

**Design Pattern: Strategy Pattern**

* 封装了多种 算法/策略
* 使得算法/策略之间能够互相替换