管理类面向对象设计

文泰来 老师

扫描二维码关注微信/微博
获取最新面试题及权威解答

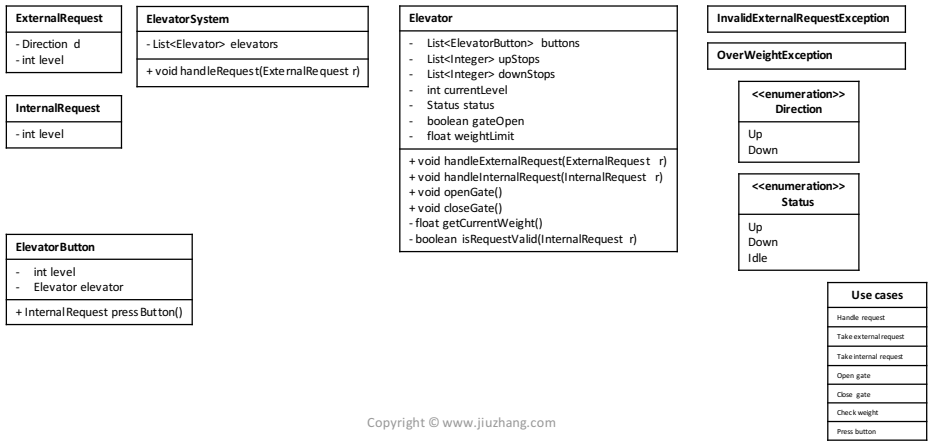
微信: ninechapter
知乎专栏：http://zhuanlan.zhihu.com/jiuzhang
微博: http://www.weibo.com/ninechapter
官网: www.jiuzhang.com

课程大纲

- Elevator System Follow-up
- 管理类OOD题型
- 管理类OOD解题思路
- Parking Lot

1. Clarification: 问 what how when
2. 列出core project, 有哪些 use case
3. 扩展 use case, 画类图
4. 检查 use case 能不能通过

Class – Final view



Challenge

- How do you handle an external request?

如我们最早和面试官讨论的结果：

同方向 > 静止 > 反向

Challenge

九章算法

- What if I want to apply different ways to handle external requests during different time of a day?

Challenge

九章算法

- What if I want to apply different ways to handle external requests during different time of a day?
- Solution 1: if - else

```
public void handleRequest(ExternalRequest r)
{
    if(time == TIME.PEAK)
    {
        // use peak hour handler
    }

    else if(time == TIME.NORMAL)
    {
        // use normal hour handler
    }
}
```

if else 的方法可扩展性不好

Challenge

九章算法

- What if I want to apply different ways to handle external requests during different time of a day?

Solution 2: Strategy design pattern

Challenge

九章算法

- Strategy Pattern

主题

接口

不同的 strategy 来 handle request

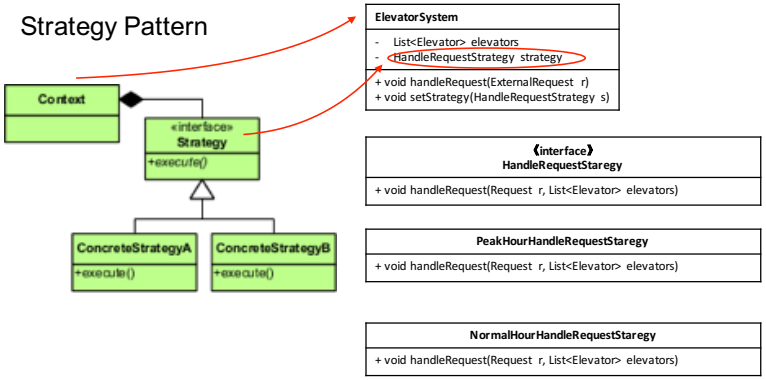
具体的不同的类来 handle strategy

※ 封装了多种算法/策略
※ 使得算法/策略之间可以相互替换

Challenge

九章算法

- Strategy Pattern



Challenge

九章算法

- Strategy design pattern

```
interface HandleRequestStrategy
{
    public void handleRequest(ExternalRequest request, List<Elevator> elevators);
}

class RandomHandleRequestStrategy implements HandleRequestStrategy
{
    public void handleRequest(ExternalRequest request, List<Elevator> elevators)
    {
        Random rand = new Random();

        int  n = rand.nextInt(elevators.size());

        elevators.get(n).handleExternalRequest(request);
    }
}

class AlwaysOneElevatorHandleRequestStrategy implements HandleRequestStrategy
{
    public void handleRequest(ExternalRequest request, List<Elevator> elevators)
    {

        elevators.get(0).handleExternalRequest(request);
    }
}
```


Challenge

九章算法

- Strategy design pattern

和 if else 的区别是什么?
1. 单独封装策略，可随意修改，不影响外部实现
2. 容易替换

```
class MyJavaApplication
{
    ElevatorSystem system = new ElevatorSystem();

    system.setStrategy(new RandomHandleRequestStrategy());

    ExternalRequest request = new ExternalRequest(Direction.UP, 3);

    system.handleRequest(request);
}

class ElevatorSystem
{
    private HandleRequestStrategy strategy = new HandleRequestStrategy();
    private List<Elevator> elevators = new ArrayList<>();

    public void setStrategy(HandleRequestStrategy strategy)
    {
        this.strategy = strategy; 可以轻易的改变 strategy
    }

    public void handleRequest(ExternalRequest request)
    {
        strategy.handleRequest(request, elevators);
    }
}
```

```
interface HandleRequestStrategy
{
    public void handleRequest(ExternalRequest request, List<Elevator> elevators);
}

class RandomHandleRequestStrategy implements HandleRequestStrategy
{
    public void handleRequest(ExternalRequest request, List<Elevator> elevators)
    {
        Random rand = new Random();

        int  n = rand.nextInt(elevators.size());

        elevators.get(n).handleExternalRequest(request);
    }
}

class AlwaysOneElevatorHandleRequestStrategy implements HandleRequestStrategy
{
    public void handleRequest(ExternalRequest request, List<Elevator> elevators)
    {

        elevators.get(0).handleExternalRequest(request);
    }
}
```

其他例子use case：使用哪种 payment，可以用 strategy design pattern

面向对象设计

九章算法

管理类 – Management

管理类 —— 什么是管理类面向对象设计?

- Gym
- Parking lot
- Restaurant
- Library
- Super market
- Hotel
- …

题目后面都可以接上三个字：“管理员”

管理类

体育馆 管理员

停车场 管理员

餐厅 管理员

图书馆 管理员

超市 管理员

宾馆 管理员

…

管理类

设计一个模拟/代替管理员日常工作的系统

管理类题目特点

- 频率：高
- 难度：中
- 题目：日常生活中常见的场景

管理类解题方法

- Clarify – What　确定是管理类问题后，如何写 core object?
  遵循以下三点：

除了题目中问的名词外，还需要从管理的名词来考虑
①　②

例子：Design parking lot.

关键字1： Parking lot
关键字2： Vehicle

管理类解题方法

- Core object -> 有进有出

③
考虑这个管理系统中，Input和Output是什么

例子：Elevator System

Input: Request
Output: Elevator

管理类解题方法

- Use case -> 从管理员角度考虑　所有的管理服务的共性

- Reserve　1. 提供预订
- Serve　2. 如何服务
- Checkout　3. 结账

本节课主要讲后两点，预定系统是一个单独的部分

管理类解题方法

- Class

在设计类图的时候，经常可以使用收据的形式，来保管信息

例子：图书馆　很重要的小技巧：

User　管理类的问题，用收据的形式来保管信息
如：user 和 book 的联系

Receipt　创建一个 recepit 的 class
不需要建立额外的 dependent

Book

Parking lot

- Can you design a parking lot ?

Clarify

- What
- How
- Who

Clarify

- What

关键字：Parking lot

Parking lot 管理什么？

Clarify

- What

关键字：Parking lot

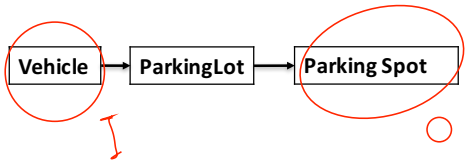
Parking lot 管理什么？

Clarify

- What

关键字：Parking lot

Parking lot 管理什么？
Vehicle/ Parking spots



Clarify

- What

关键字：Parking lot, Vehicle, Parking Spot

即便遇到自己熟悉的题，
也要跟面试官交流交流，了解面试官的需求

What

关键字：Parking lot

属性：？

What

关键字：Parking lot

Challenge



- What are the differences between these parking lots?

Challenge


Parking lot -> Parking spaces

Parking lot -> Parking level -> Parking spaces

Parking lot -> Parking level(optional) -> Parking space -> Upper/Lower space

level — 1
— 1+
space — upper
— lower

What

关键字： Vehicle

属性：？

What

关键字：Vehicle



- 设计的停车场是否需要纳不同大小的车辆？

What

关键字：Parking Spot

属性？

What

关键字：Parking Spot

- 设计的停车场是否要考虑残疾人停车位？

What

关键字：Parking Spot

- 设计的停车场是否要考虑充电车位？

What

针对本题：

- Parking lot: 考虑多层的Parking lot, 没有错层
- Vehicle: 考虑三种大小的车
- 不考虑残疾人停车位/充电车位

Challenge

如何设计停车场
来支持停不同大小的车?

- 当寻找合适的车位的时候，需要看边上的位置是否是空位

  一个车占多个停车位，
  要考虑旁边是否有空位

- 当有新的车形需要支持的时候，需要大量修改
- 利用率更低

  专门 size 专门用
  1. 可以用继承，
  2. 有新车型，需要大量修改

Clarify

- How

停车场有哪些规则？

要以停车场为主题，不要以开车的人为主题来思考

How

- 规则1：如何停车?

已知信息：所有车位都是相同规格的

How

- 规则1：如何停车?

从车的角度出发：开进停车场 -> 经过每一个位置看看能不能停 -> 停进一个位置

Input Output

从停车场的角度出发：开进停车场 -> 返回一个能停的地方 ->停进一个位置

How

- 规则1：如何停车?

VS.

How

- 针对本题：根据车的大小，横向停车

是否可以横向停车?

How

- 规则1：如何停车?

是否需要显示每层的空位?

How

- 规则2：收费

免费还是付费？

按时付费还是按天付费?
对于大规模的,
对于小规模的,
用不同的 strategy design pattern

Clarify

- Who

Optional, 通过思考题目当中是否有人的出现，来帮助确定解题范围

*一般可以考虑人的角色以及人的属性，看是否题目需要

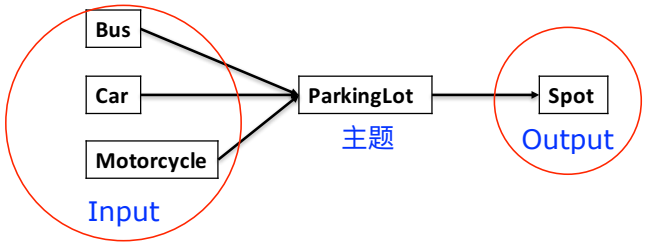
Who

- 针对本题：N/A

Core Object

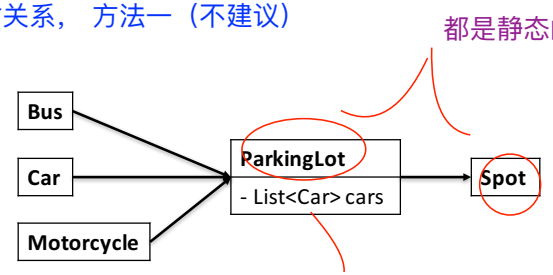
- 什么是Core object ?
- 为什么要定义Core Object ?
- 如何定义Core Object ?

Core Object



Core Object

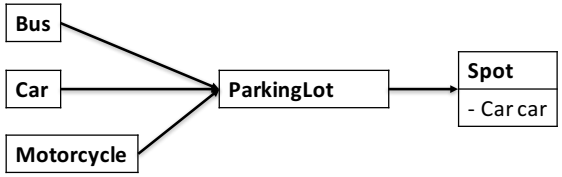
建立映射关系， 方法一（不建议）



动态与静态之间互相存放，不太好
相当于增加了一个外界的 dependency
如果以后改变了车的车牌号，spot 也需要改，但这是 unnecessary 的

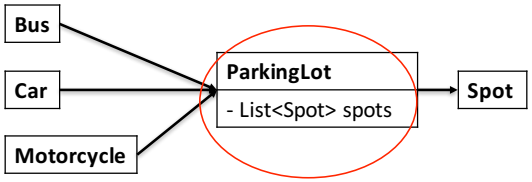
Core Object

建立映射关系， 方法二（不建议）



Core Object

parking 和 spot 都是静态的，适合相互建立联系
车和 spot 之间，可以用 receipt 进行连接



Cases

- 什么是Use case？ use case —— parking lot 需要支持的 feature
- 为什么要写Use cases？
- 如何写Use cases？

Cases

- Bus / Car / Motorcycle

站在管理员的角度想

Cases

- Bus / Car / Motorcycle

N/A

Cases

- ParkingLot

站在管理员角度考虑

Cases

- ParkingLot

- Get available count
- Park vehicle
- Clear spot
- Calculate price

Management类常见Use case:
- Reservation : X
- Serve: Park vehicle
- Check out: Clear spot + Calculate price

Cases

- Spot

Cases

- Spot

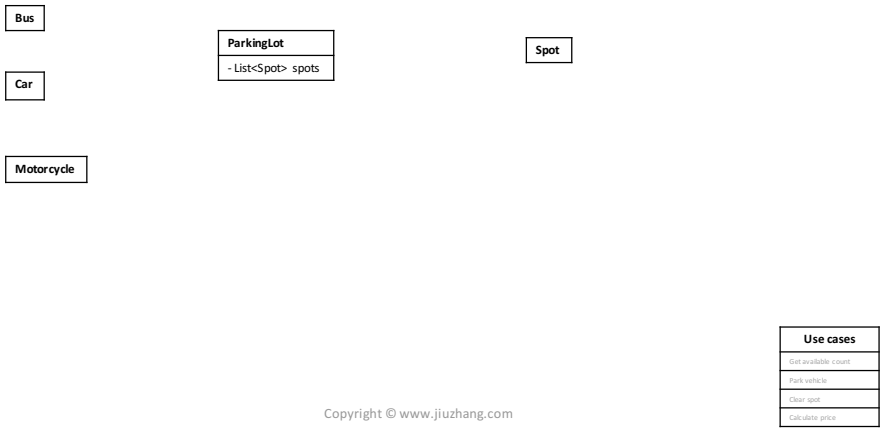
- N/A

Class

- 什么是类图？
- 为什么要画类图？
- 怎么画类图？

Class

Class

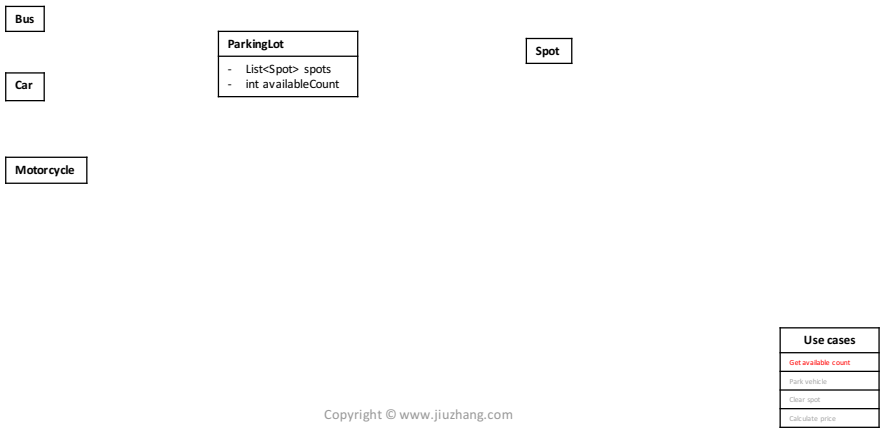


Class

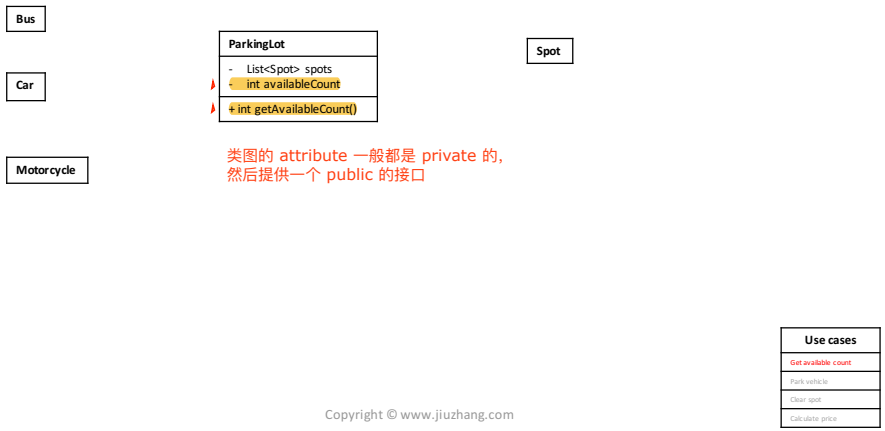
- Use case: Get available counts

Parking lot shows how many **available spots** in total

Class



Class



类图的 attribute 一般都是 private 的，
然后提供一个 public 的接口

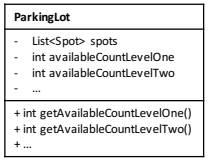
Challenge

- 如何分别显示出每一层的空位个数？

Challenge

- 如何分别显示出每一层的空位个数？

Solution 1: 有几层就保存几个变量　不够 extenedable 有增删时不方便



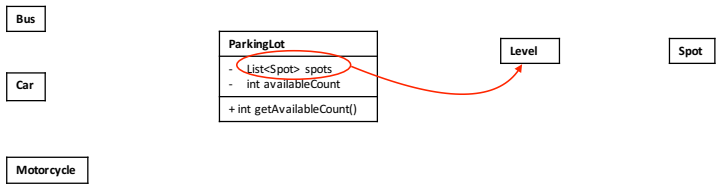
Challenge

- 如何分别显示出每一层的空位个数？

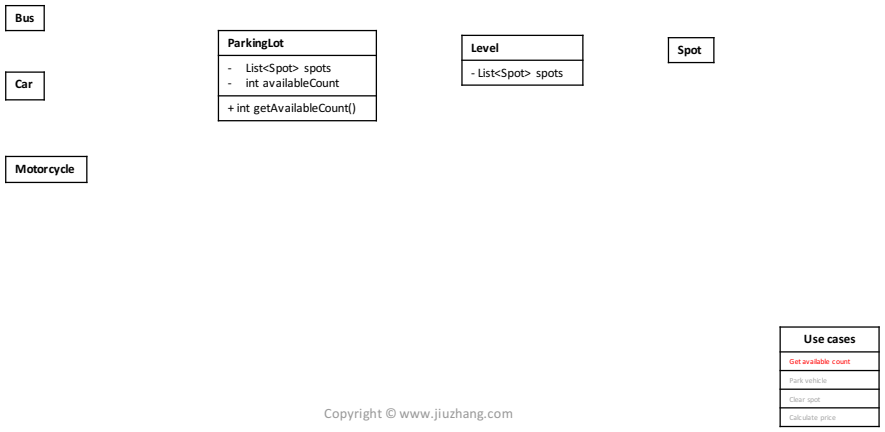
Solution 2: 新建一个Level类

以下讲解如何扩展

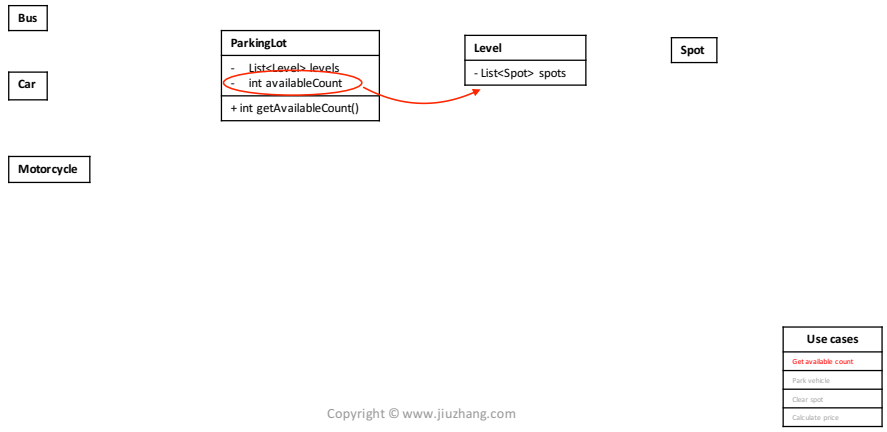
Class



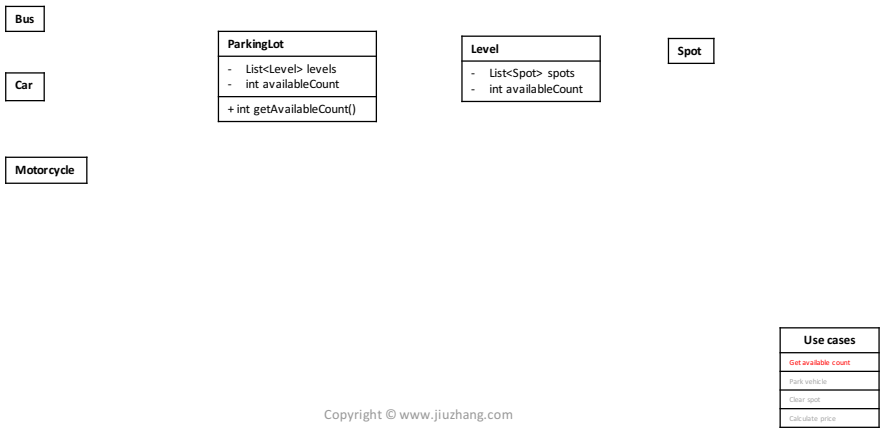
Class



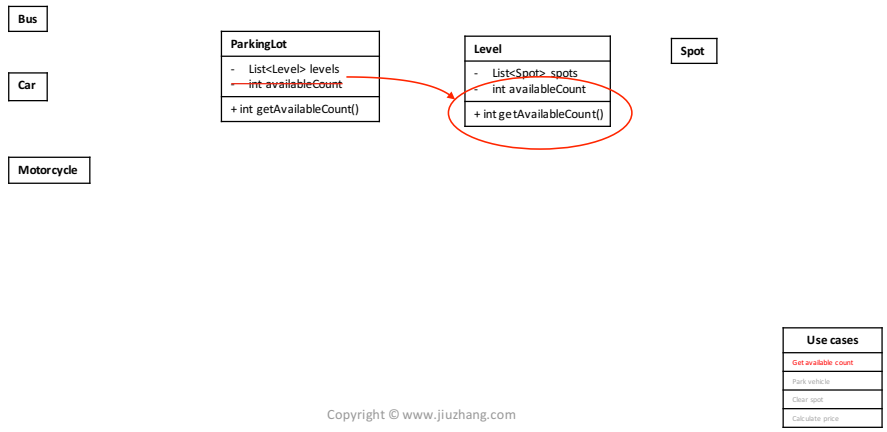
Class



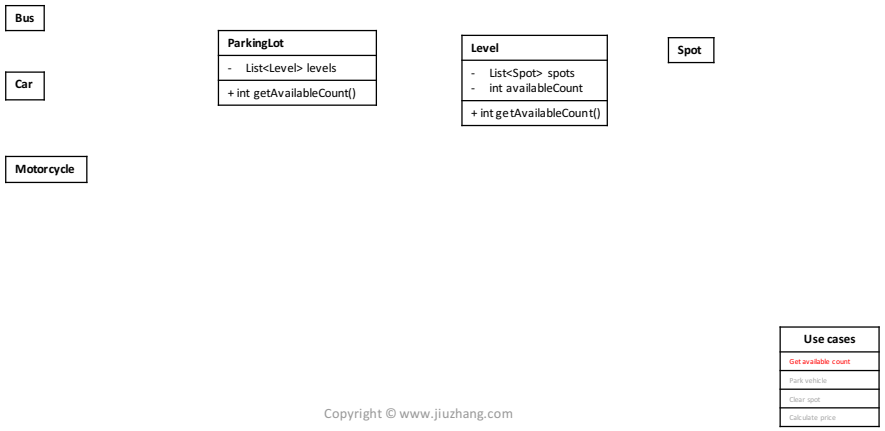
Class



Class



Class



Class

- Use case: Park vehicle

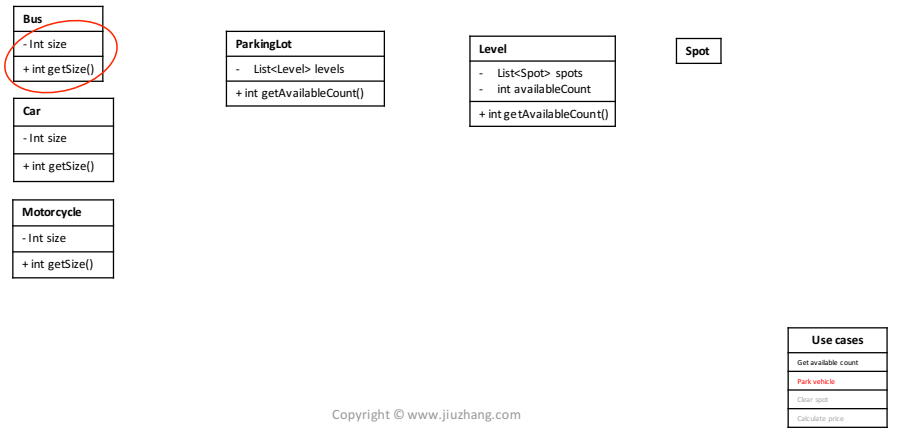
- Parking lot checks the size of vehicle
- Parking lot find an available spot for this vehicle
- Vehicle takes the spot(s)

Class

- Use case: Park vehicle

- Parking lot checks the size of vehicle

Class

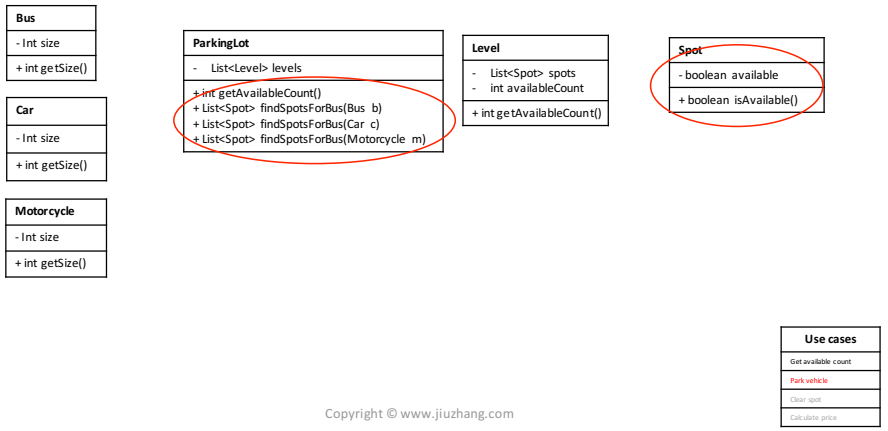


Class

- Use case: Park vehicle

- Parking lot checks the size of vehicle
- Parking lot find an available spot for this vehicle

Class

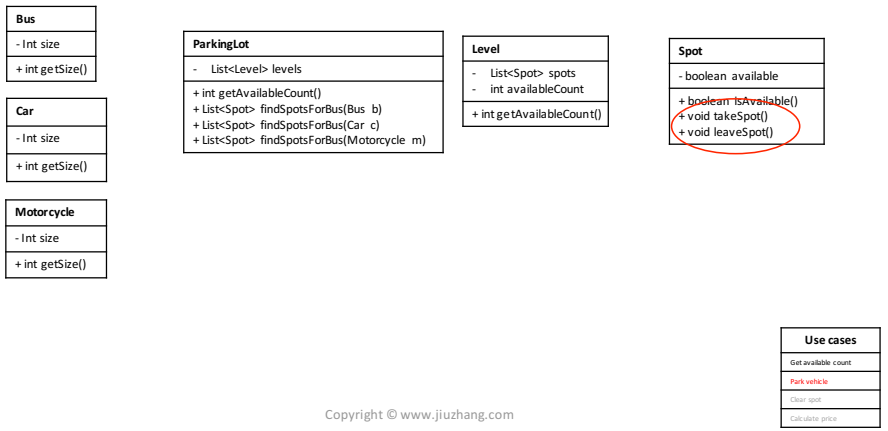


Class

- Use case: Park vehicle

- Parking lot checks the size of vehicle
- Parking lot find an available spot for this vehicle
- Vehicle takes the spot(s)

Class



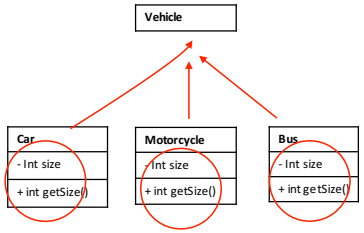
Best practice

- Open for extension, close for modification

  刚刚的方法扩展性不太好，

  现在我们要用继承使他更 extendable

Best practice 先 identify 有哪些可以继承



ParkingLot
- List<Level> levels

+ int getAvailableCount()
+ List<Spot> findSpotsForBus(Bus b)
+ List<Spot> findSpotsForBus(Car c)
+ List<Spot> findSpotsForBus(Motorcycle m)

Spot
- boolean available

+ boolean isAvailable()
+ void takeSpot()
+ void leaveSpot()

Level
- List<Spot> spots
- int availableCount

+ int getAvailableCount()

由于 private 只能被本身用，
子类也没有 access，所以这里要用 protected

Use cases
Get available count
Park vehicle
Clear spot
Calculate price

Best practice

Vehicle
Int size
+ int getSize()

Car | **Motorcycle** | **Bus**

ParkingLot
- List<Level> levels

+ int getAvailableCount()
+ List<Spot> findSpotsForBus(Bus b)
+ List<Spot> findSpotsForBus(Car c)
+ List<Spot> findSpotsForBus(Motorcycle m)

用一个就可以了

Spot
- boolean available

+ boolean isAvailable()
+ void takeSpot()
+ void leaveSpot()

Level
- List<Spot> spots
- int availableCount

+ int getAvailableCount()

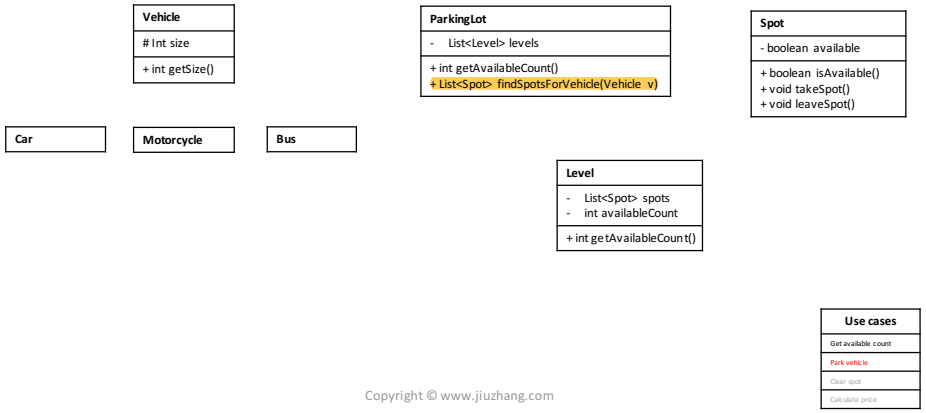
Use cases
Get available count
Park vehicle
Clear spot
Calculate price

Best practice

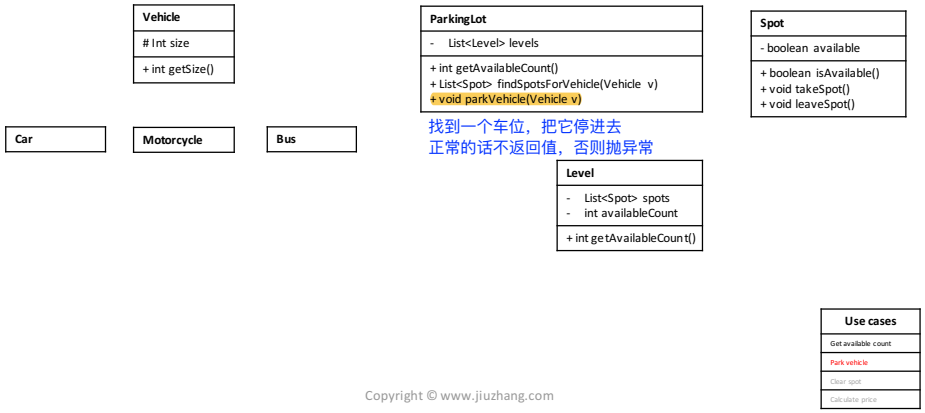
```java
public class Vehicle
{
    private int size;

    public int getSize()
    {
        return size;
    }
}

public class Bus extends Vehicle
{
    private int size;

    public int getSize()
    {
        return size;
    }
}
```

```java
public class Vehicle
{
    protected int size;

    public int getSize()
    {
        return size;
    }
}

public class Bus extends Vehicle
{
    public Bus()
    {
        size = 3;
    }
}
```

Best practice



Class



Class

Class

- Use case: Clear spot

- Parking lot find the spot to clear
- Update spot to be available
- Update available count for each level

Class

- Use case: Clear spot

- Parking lot find the spot to clear

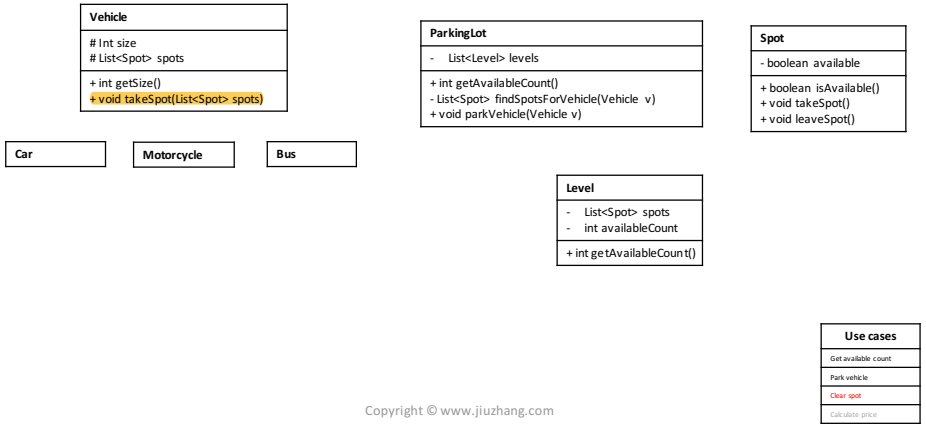
Challenge

- 如何找到需要被free的spot？

Solution 1: Vehicle保存停的车位

把 spot 的信息放到车里?

Class

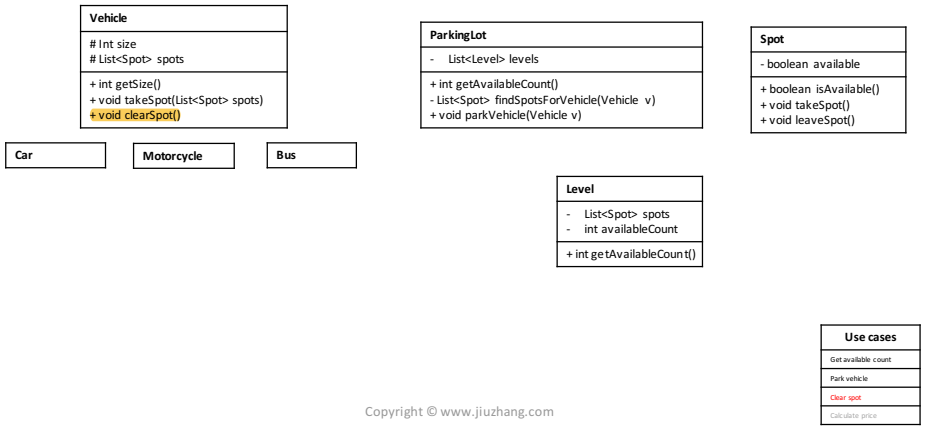


Class

- Use case: Clear spot

- Parking lot find the spot to clear
- Update spot to be available

Class

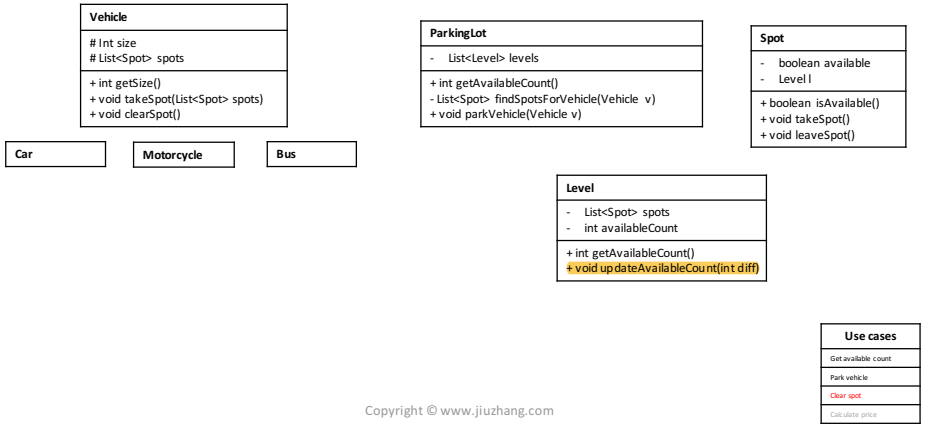


Class

- Use case: Clear spot

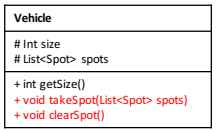
- Parking lot find the spot to clear
- Update spot to be available
- Update available count for each level

Class



Best practice

- 不是说好从Parking lot出发吗？

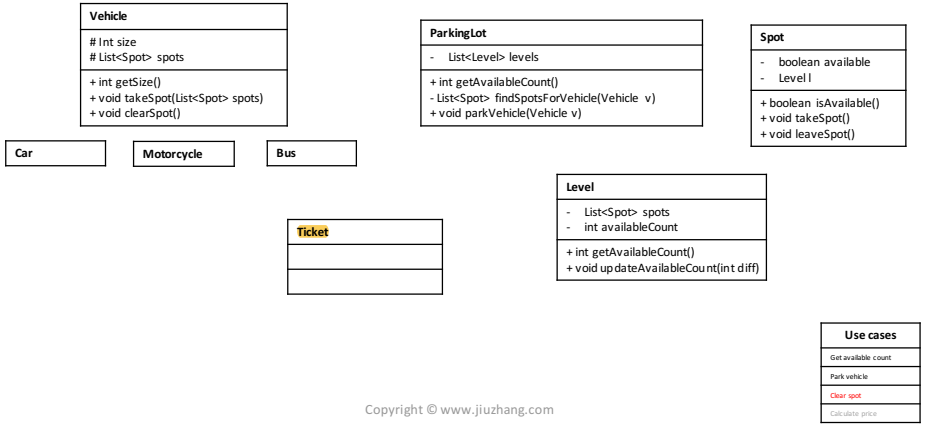


这个方法不好，把车和 spot 建立了联系 :(

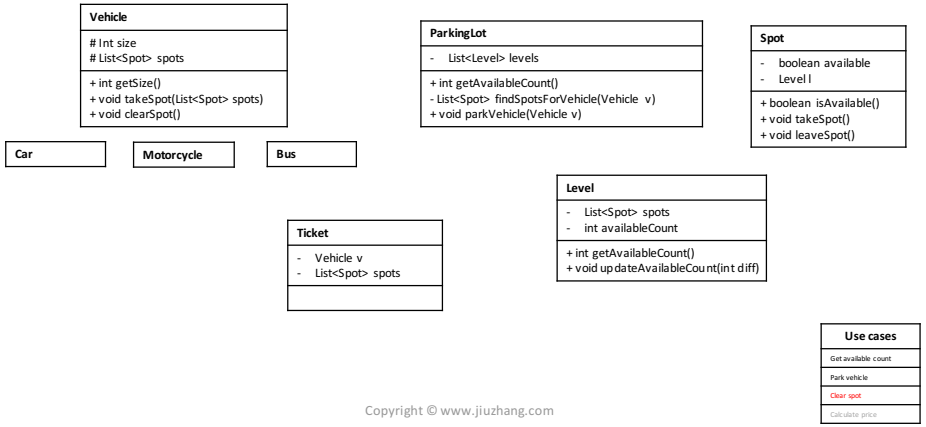
Best practice

- Solution: **Receipt**

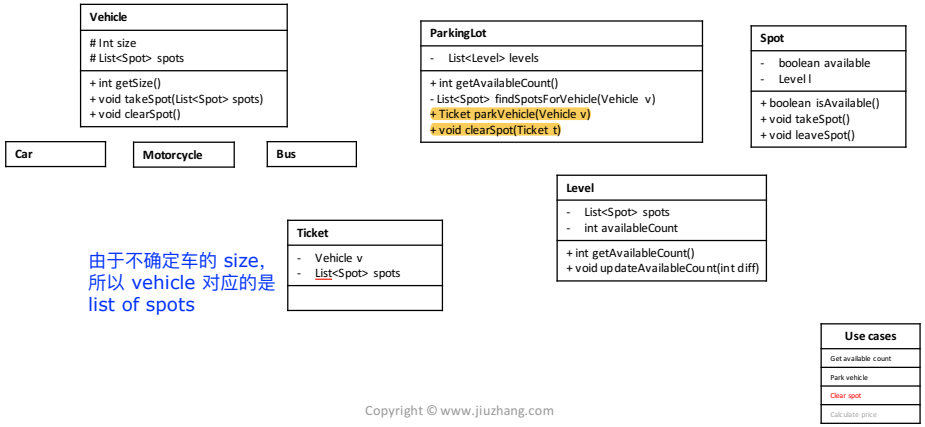
Class



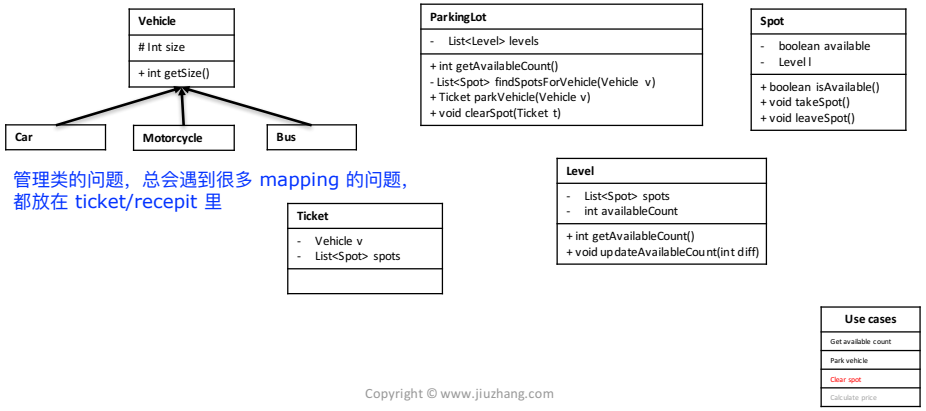
Class



Class



Class



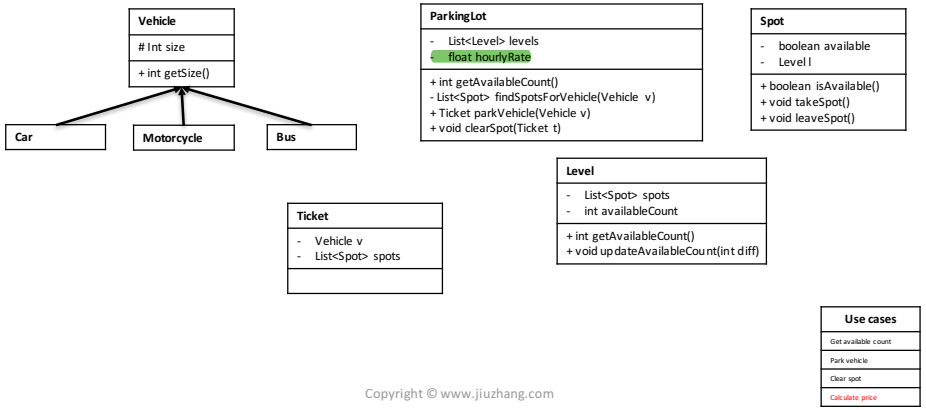
管理类的问题，总会遇到很多 mapping 的问题，都放在 ticket/recepit 里

Class

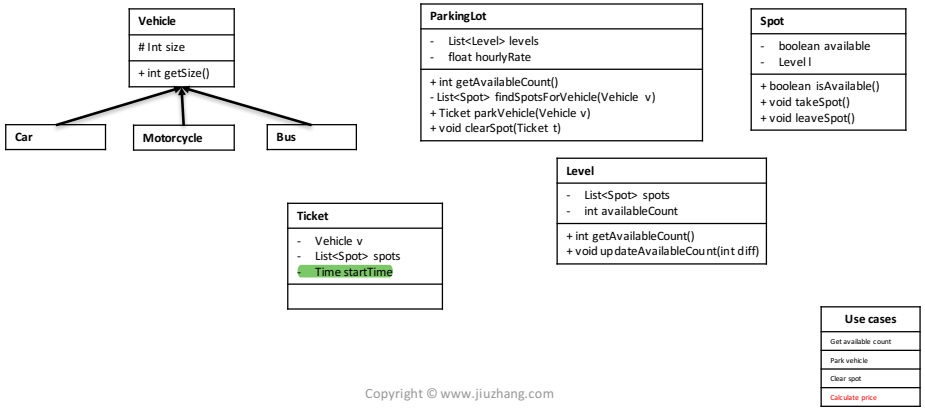
- Use case: Calculate price

- When clear spot, parking lot calculates the expected price to pay

Class

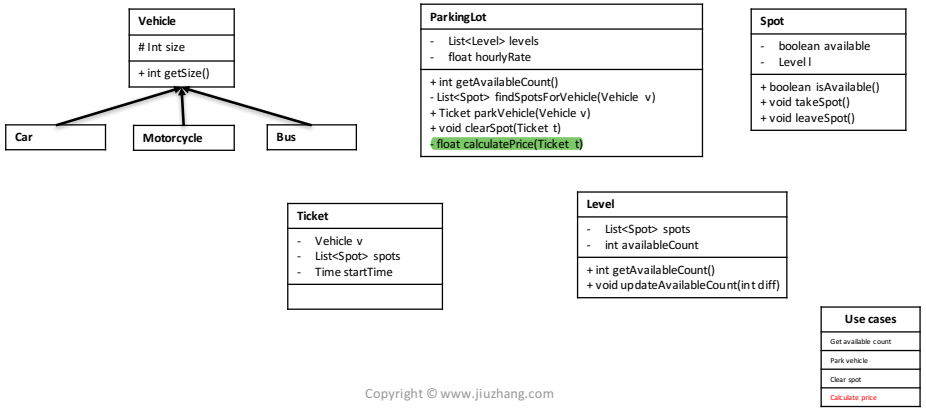


Class



Class

从 parking lot 的视角，有什么尽量往这里加



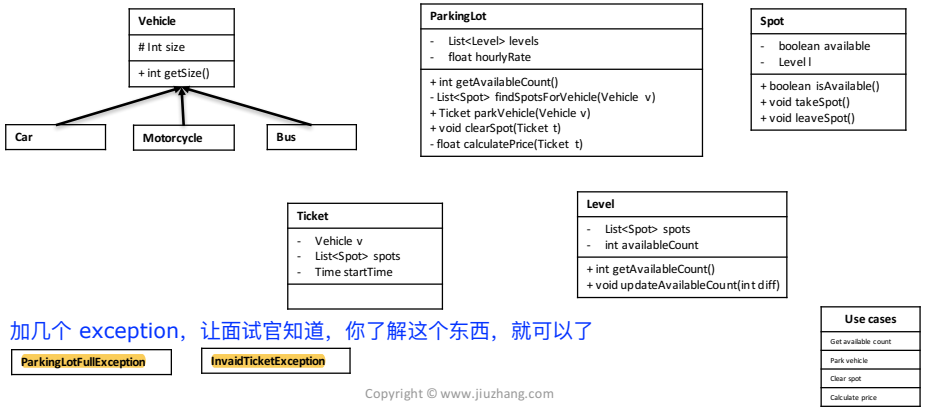
Correctness

- 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

※ 继承
※ expection

Exceptions



加几个 exception，让面试官知道，你了解这个东西，就可以了

ParkingLotFullException　InvaidTicketException

Challenge

- Parking lot里每层的spots，是怎么排列的？当停Bus时，是否有问题？

仅用一个 list of spot 来表示 level
则停车场就是一长条，
这不是一个合适的 design，
因为有时候需要知道连续的位置

Challenge

VS.

Challenge

- Solution 1:

- 在Level里加一个变量，作为每行固定的停车位个数
- 在Spot里加一个变量，作为Spot Id
- 这样能够知道哪些Spot在一行 / 一行有没有足够的Spots

Challenge

- Solution 2:

- 像添加Level一样，添加一个Row作为新的Class

  相当于二维化了

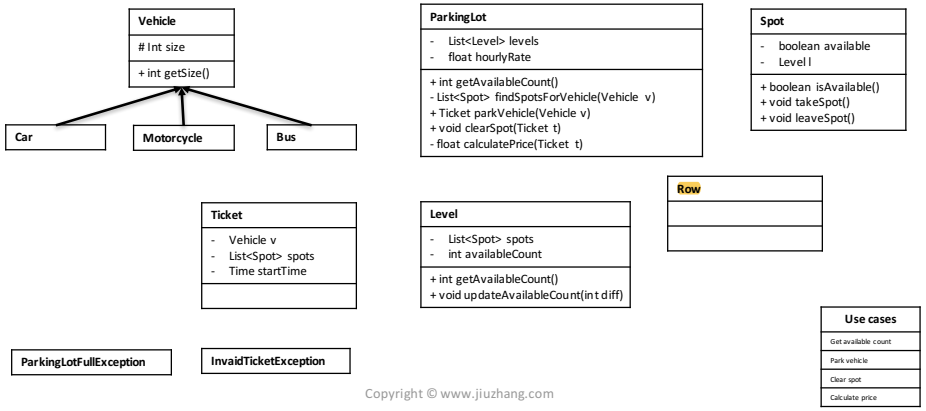
Challenge

- Solution 1 VS. Solution 2 ?

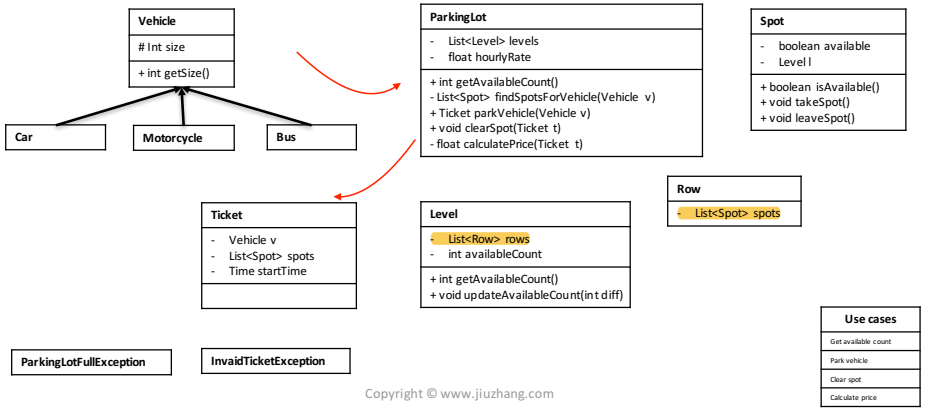
如果用Solution 1, 每行的个数必须要一样

实际情况，有时候停车场有一些柱子，
每行个数有可能不一样

Class

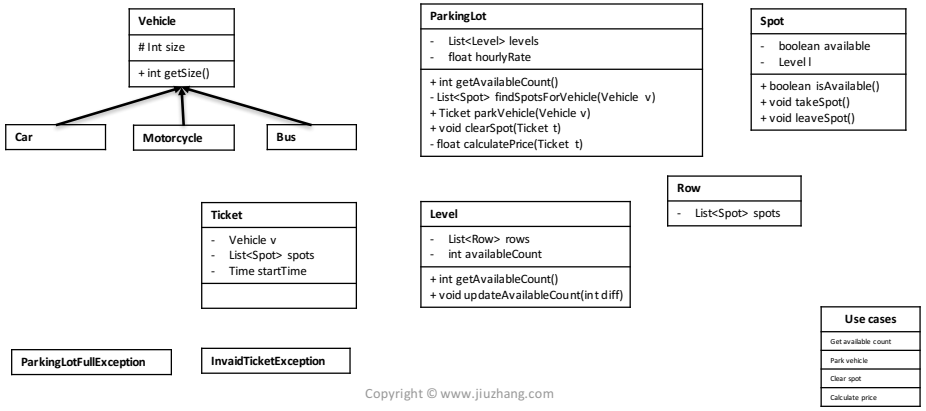


Class

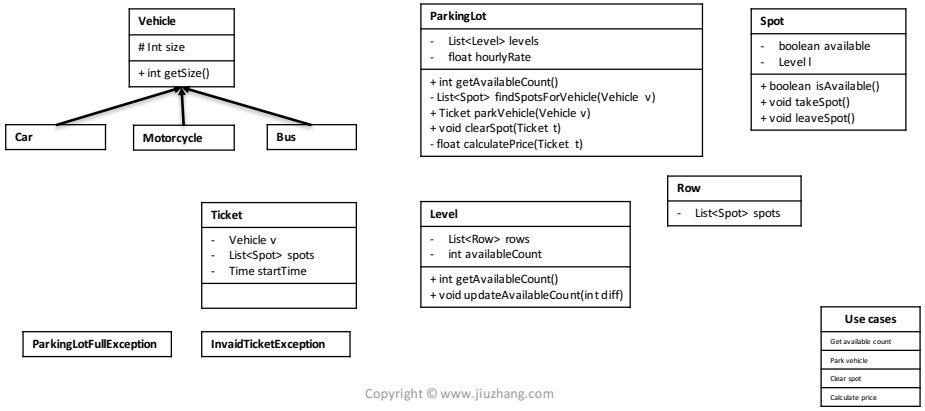


Class – Final view

多线程：寻找 critical data，
同一个时间，只能被一个主题访问，
给它加一个锁



Exception



Design pattern

- Clean and elegant
- Keep code extendable
- Safe
- Show off your skills ！

Design pattern

- Singleton 保证一个类只有一个 instance

ensure a class has only one instance, and provide a global point of access to it

不要盲目的运用singleton，
要先跟面试官讨论，
把其他的设计做完，
再问需不需要保证 single instance
如果需要的话，再做

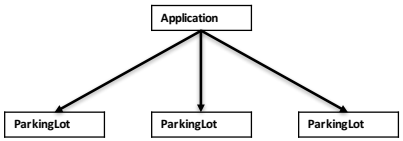
Design pattern

- Singleton

```
public class ParkingLot
{
    private List<Level> levels;

    public ParkingLot()
    {
        levels = new ArrayList<Level>();
    }
}
```


Design pattern

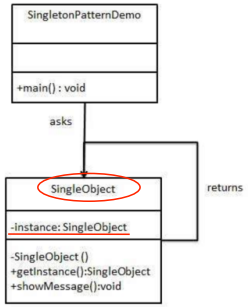
- Singleton



Design pattern

九章算法

- Singleton



Design pattern

九章算法

- Singleton – 基本式

```
public class ParkingLot
{
    private static ParkingLot _instance = null;

    private List<Level> levels;

    private ParkingLot()
    {
        levels = new ArrayList<Level>();
    }

    public static ParkingLot getInstance()
    {
        if(_instance == null)
        {
            _instance = new ParkingLot();
        }
        return _instance;
    }
}
```

class 外部不能 new class 里private 的东西

下面定义一个 public 的用于外面调用

static 意味着和 class 本身有关系，
而不是和具体的 instance/object有关系
static 调用的东西也不能是具体的 instance

保证了外部都有一个 **global entry to the parking lot**

Design pattern

九章算法

- Singleton – 线程安全式

```
public class ParkingLot
{
    private static ParkingLot _instance = null;

    private List<Level> levels;

    private ParkingLot()
    {
        levels = new ArrayList<Level>();
    }

    public static synchronized ParkingLot getInstance()
    {
        if(_instance == null)
        {
            _instance = new ParkingLot();
        }
        return _instance;
    }
}
```

代表进来就是上锁的

很安全，但不够 ***efficient***

Design pattern

九章算法

- Singleton – 静态内部类式

```
public class ParkingLot
{
    private ParkingLot(){}

    private static class LazyParkingLot
    {
        static final ParkingLot _instance = new ParkingLot();
    }

    public static ParkingLot getInstance()
    {
        return LazyParkingLot._instance;
    }
}
```

build time 就已经生成了
一个无法改变的 instance,

如果 run time 别人要调用,
只需要调用
和这个类 associate 在一起的
这个 instance,
因为是 final,
所以永远都会返回这个东西