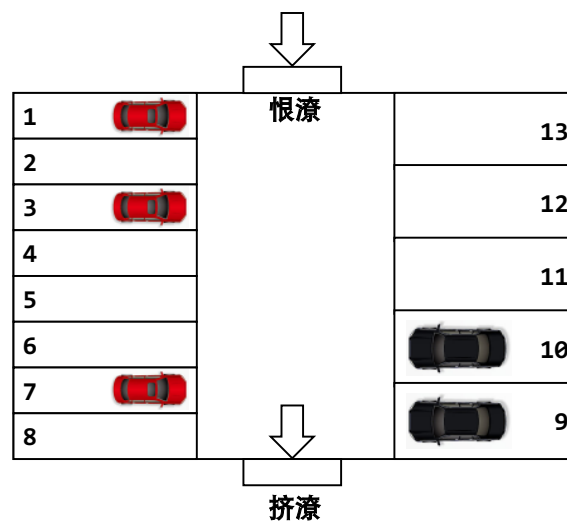


1 给出一个 Java ADT 题目

某公司拟设计和开发一个停车场管理系统，其基本需求陈述如下：

- (1) 一个停车场有 n 个车位($n \geq 5$)，不同停车场包含的车位数不同。
- (2) 一辆车进入停车场，如果该停车场有空车位且其宽度足以容纳车的宽度，则可以在此停车。
- (3) 停在停车场里的车，随时可以驶离停车场，根据时间自动计费（每半小时 10 元，不足半小时按半小时计算）。
- (4) 停车场管理员可以随时查看停车场的当前占用情况。

下图给出了一个包含 13 个停车位的小型停车场示例图，其中 1-8 号停车位较窄，9-13 号停车位较宽。在当前状态下，第 1、3、7、9、10 号车位被占用，其他车位空闲。



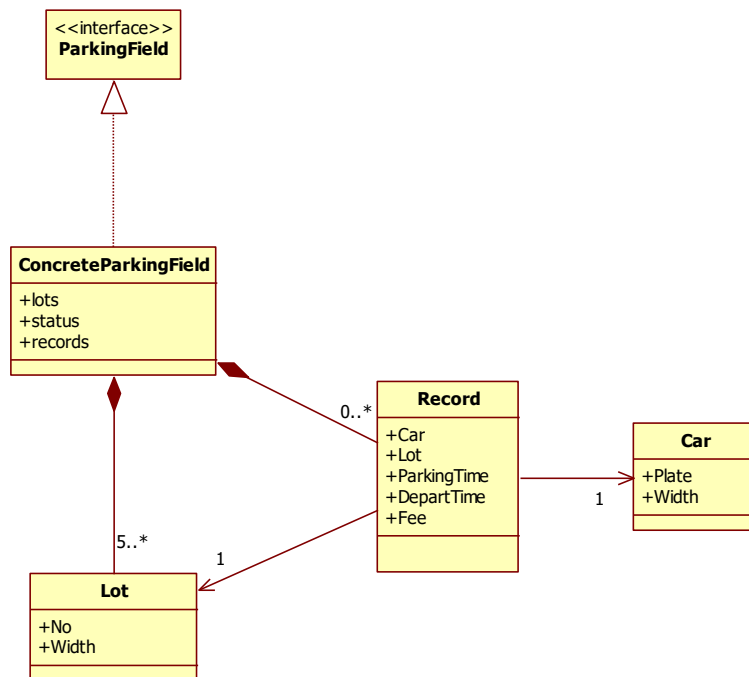
客户端程序的功能需求：

- 构造一个停车场
- 构造若干台车
- 依次将车停进停车场，可以指定车位，也可以不指定车位（随机指派）
- 随机将车驶离停车场，车辆驶离时给出入场时间、出场时间、费用金额
- 查看当前停车场的状态（目前每个车位停了什么车）

特殊情况：

- 停车进场的时候（两种情况）：该车辆已经在停车场里面了
- 停车进场的时候（不指定车位）：停车场已没有可供该车停车的位置
- 停车进场的时候（指定车位）：该车位已被占用、该车位过窄、没有该车位
- 驶离停车场的时候：该车并没有停在这里

2 回顾 3-6 ADT 设计习题课的结果



3 扩展 Car 至其他交通工具类型：可复用性

对上述设计进行扩展，考虑将来不仅可以停汽车，也可以停马车、摩托车、飞机（相当于将停车场扩展到了飞机场）等，并可在后续持续扩展其他事物。只要某个 lot 的宽度（width）大于某个对象的宽度，即可停在该位置。但是，除了牌照号和宽度之外，马车、摩托车、飞机等还具有与汽车不同的属性和方法。为此需如何修改当前设计？

思路：

- 目前 ParkingField 里只能停 Car，ConcreteParkingField 里构造 Car 对象并存储于 status 和 records 中。
- 为了扩展，这里用到的所有的 Car 都要替换为一个更抽象的类型。
- 按照 OOP 的思路，不妨用一个接口来实现它。在接口中定义共性操作，然后派生出具体的子类型。

改造过程：

- (1) 构造 Parkable 接口，提供 getWidth(), getPlate() 方法----这是原来 Car 类实现的方法。
- (2) 把 Car 改造成 implements Parkable。
- (3) 之前的方案中，在 ConcreteParkingField 和 Record 中使用过 Car，现在将其替换为 Parkable。
- (4) 在 ConcreteParkingField 的 parking() 方法里，有 new Car(..) 的调用。故在 Parkable 中使用静态工厂方法，new 操作改为：

```
Parkable parkable = Parkable.create(plate, width);
```

工厂方法：

```
public static Parkable create(String plate, int width) {  
    return new Car(plate, width);  
}
```

(5) 设计 `Motor`、`Plane` 类，实现 `Parkable` 接口。但是，`Car`、`Motor`、`Plane` 的共性 `rep` 和方法是重复的，可以进一步抽象出父类 `ConcreteParkable` 类。上述 `Parkable` 接口中的静态工厂方法也改造为创建 `ConcreteParkable` 对象。

(6) `Client` 端代码是否受到影响？不会，因为原来的 `Car` 类对 `client` 不可见。

4 使用 Factory Method 设计模式：可维护性

有一点问题：在 `Parkable` 接口的 `create` 方法中，目前是传入 `plate` 和 `width`，那么是否要区分车辆类型——`Motor`、`Car`、`Plane`？

换句话说，`ConcreteParkable` 类是不应该被实例化的，而是应实例化具体的子类型 `Motor`、`Car` 或 `Plane`。可以将 `ConcreteParkable` 设置为 `abstract` 的，避免后续对其实例化。

扩展，允许创建时传入停车物的类型，增加参数 `String type`
同时修改 `ParkingField` 接口中的 `parking` 方法参数，增加 `type`
修改客户端代码，增加参数 `type`

考虑到目前 `Parkable` 接口的 `create` 中只能创建 `ConcreteParkable` 类型的对象，增加专门的工厂方法接口与类：`ParkableFactoryInterface`、`ParkableFactory`，目前只包含一个方法：

```
public Parkable create(String type, String plate, int width) throws Exception {  
    if(type.equals("car"))  
        return new Car(plate, width);  
    else if (type.equals("motor"))  
        return new Motor(plate, width);  
    else if (type.equals("plane"))  
        return new Plane(plate, width);  
    else  
        throw new Exception("Illegal type");  
}
```

也可以不用这种 `if/else` 结构，分别为 `Car`、`Motor`、`Plane` 单独创建工厂方法类，从而将来可扩展新子类来支持更多的车辆类型，而无需修改上述代码。

当然，你也可以把这段逻辑直接放在 `Parkable` 的 `create` 静态方法中，不再设计新的工厂接口和类。这么做的缺点是：将来扩展的时候要改代码（**OCP 原则要求：扩展而不是修改**）。

为 `Parkable` 的各个子类型撰写 `equals()`，`hashCode()`，`toString()`。

5 考虑 Car、Motor、Plane 的个性化

给子类 `Car`、`Motor`、`Plane` 增加个性化特征。

目前，三个子类的 `rep` 和方法都是一样的，完全放在 `ConcreteParkable` 中实现。

考虑以下要扩展的个性化需求：

- (1) 计费规则不同，汽车半小时 10 元，飞机 1 小时 1000 元，摩托车半小时 5 元
- (2) 疫情期间，停车场要求汽车和摩托车需要登记驾驶人信息，飞机需要登记来自哪个机场及日期。

针对第(1)点，三者计费信息的 rep 一致，可以在 ConcreteParkable 中增加 field (pricingUnit、price，为 protected)，在各自子类的构造函数中增加赋值语句。

```
public class ConcreteParkable implements Parkable {
    private String plate;
    private int width;
    protected double pricingUnit;
    protected double price;

    public class Car extends ConcreteParkable {

        public Car(String plate, int width) {
            super(plate, width);
            pricingUnit = 30;
            price = 10;
        }
    }
}
```

然后，为了支持 ParkingField 中的计费，在 Parkable 接口中给这两个 fields 增加 getter 方法，以便于从各 Parkable 对象取出计费标准。修改 Record 类的 calcFee() 方法，根据车型不同，按不同标准计算费用。

```
public double calcFee() {
    long diffInMilis = this.timeOut.getTimeInMillis() - this.timeIn.getTimeInMillis();
    long diffInMinute = diffInMilis / (60 * 1000);

    //fee = ((int) (diffInMinute / 30) + 1) * 10;
    fee = ((int) (diffInMinute / car.getPricingUnit()) + 1) * car.getPrice();
    return fee;
}
```

注：本例仅为了演示子类型设计，实际上这两个 field 完全可以以 static fields 的形式加入各个子类型，除非不同车辆个体的计费标准有差异。

针对第(2)点，需要额外增加方法，且三者不一致。按课程中所介绍的多种设计策略都可以实现复用，但各自的代价不同。这里采用 delegation 的策略，将需要登记的信息和登记动作分离出来，形成单独的类(DriverRegistration、AirportRegistration)，然后在 Car、Motor 等三个子类中根据个性化分别进行 delegation。如果登记信息和功能将来变化时，只需要扩展修改这些类即可。

强调：完全共性的放到父类，完全个性的放到子类，部分共性的最考验设计。仔细斟酌课程中所学习的各种策略。

增加了这些个性化后，如何影响 PakingField 和客户端代码？

- 修改 ParkingField 接口的 parking 方法，增加登记的额外信息
- 修改 Parkable 的 static create()方法，增加额外信息
- 修改 ParkableFactory 的 create()方法，增加额外信息

- 在 client 代码中，传入额外信息

额外信息使用 `String[] extraRegistrationInfo` 的参数。

```
@Override
public Parkable create(String type, String plate, int width, String[] extraRegistrationInfo) throws Exception {
    if (type.equals("car")) {
        Car c = new Car(plate, width);
        c.registerDriver(extraRegistrationInfo[0]);
        return c;
    } else if (type.equals("motor")) {
        Motor m = new Motor(plate, width);
        m.registerDriver(extraRegistrationInfo[0]);
        return m;
    } else if (type.equals("plane")) {
        Plane p = new Plane(plate, width);
        Calendar c = Calendar.getInstance();
        //根据extraRegistrationInfo[1]字符串设置c的值
        p.registerAirport(extraRegistrationInfo[0], c);
        return p;
    }
    else
        throw new Exception("Illegal type");
}
```

但是，Car/Motor 只需要一个数据，Plane 需要两个数据。是否有更好的办法？

实际的调用次序：

- 客户端调用 `ParkingField` 的 `parking()`
- `parking()` 调用 `Parkable` 的 `create()`，创建停车物
- `Parkable` 的 `create()` 调用工厂方法，创建具体停车物类型（共性信息和个性信息分开处理）

其他方案：直接在客户端创建 Car/Motor/Plane 对象；在 `ParkingField` 中创建具体对象。都不好，将具体类暴露于 client（`ParkingField` 相对于 `Parkable` 来说也是 client），不利于隔离变化。

6 使用 State 设计模式管理停车物状态：可扩展性

需要扩展功能：每辆 `Parkable` 对象有两个状态：在停车场、在路上。客户端输入车牌号可查询状态。

可以通过在 `ParkingField` 接口中增加新方法来实现，遍历 `status` 或 `records` 即可获得（你可以试着实现）。但这种方案不容易实现灵活扩展，状态增删的时候要改逻辑，违反 OCP。

假如用 state 设计模式，`parkable` 对象自行管理自己的状态呢？

创建 `State` 接口，有两个方法：`parking`，`depart`。

创建两个实现类，`ParkingState` 和 `OnroadState`，分别在其中实现状态转换。

`Parkable` 初始化的时候，缺省状态是 `OnroadState`。

```

public class ConcreteParkable implements Parkable {
    private String plate;
    private int width;
    protected double pricingUnit;
    protected double price;

    private State state = OnroadState.getInstance();

```

Parkable 增加 getState/setState 方法，供客户端查询状态和设置状态。

在 ParkingField 的 parking 方法停车时，转为另一状态；depart 方法的时候，又变回来。

```

// 正式停车
status.put(lot, parkable);
records.add(new Record(parkable, lot));

parkable.setState(parkable.getState().parking());

```

这么做的好处：状态可以扩展、变化，只需要改 State 子类即可，不会变动其他类/client（client 只需要在特定位置调用 State 接口的状态转换方法即可，但转换的具体逻辑不会出现在 client 代码中）。

你可以试着扩展到三种状态试一试。

在 ConcreteParkable 的 toString() 中增加状态信息的输出。

```

@Override
public String toString() {
    return plate + " (" + width + ", " + state + ")";
}

```

7 使用 Decorator 设计模式：可复用性/可扩展性

有些停车场是政府设置的公共停车场（无人管理），有些停车场则是由专门的公司管理。上述 ConcreteParkingField 的 rep 不支持后者。在不改变 ConcreteParkingField 现有实现的情况下，能够做到：

(a) 在创建 ParkingField 对象时包含“公司”信息（String company 即可）；

(b) 车辆在此类停车场进行停车（调用 parking 方法）和驶离（调用 departure 方法）的时候，能够打印输出（System.Out）欢迎和告别信息（分别为：停车场 XX 欢迎车辆 YY、停车场 XX 祝车辆 YY 旅途愉快，其中 XX 表示停车场的名字，YY 表示车辆的车牌号）。

注意：不是说所有停车场都是带公司和欢迎信息的，所以之前设计的 ParkingField 还要保留，不能直接在 ConcreteParkingField 中修改 rep 和相关方法的逻辑，而是要“扩展”——牢记 OCP。

方案 1：使用继承，派生一个子类，专门负责处理这种特殊的停车场，增加 rep（公司信息）、override 方法（parking、depart）

方案 2：使用 Decorator 设计模式，客户端可以这样：

```

ParkingField pf = ParkingField.create(lots);

```

```
ParkingField pfwc = new ParkingFieldWithCompany(pf, "HIT_CS");
```

第一行是之前的停车场对象，第二行是对其进行装饰之后的、具有新能力的停车场对象。

建立一个用于装饰的基础类ComplexParkingField，实现ParkingField接口，其中的所有方法都delegate到未装饰之前的对象。

然后，建立一个具体装饰类ParkingFieldWithCompany，也实现ParkingField接口，继承自ComplexParkingField类，增加了rep（公司信息），对需要变化的方法进行扩展（使用super.xxx()调用基础功能）。修改构造函数，增加company参数。例如：

```
@Override
public double depart(String plate) throws Exception {
    double price = super.depart(plate);
    println(plate + ", " + company + " wish you a good drive");
    return price;
}
```

你可以试着扩展其他更复杂的装饰功能。

8 使用 Visitor 设计模式：可扩展性

考虑将来对 ParkingField 的功能扩展。例如要扩展的一个功能是统计停车场当前时刻占用比例(=已停车的车位数量÷总车位数)。

在客户端代码中，给定一个 ParkingField 对象，如何使用你的 visitor 来统计当前时刻占用比例。

```
ParkingField pf = ...; //此处假设 pf 已成功创建
                        //并且进行了一系列 parking 和 depart 操作
double fullRatio = ...;
```

方案 1：该功能可以直接扩展至 ParkingField 接口中，增加一个方法。但是没有解决将来扩展其他新方法的能力。

方案 2：使用 visitor 设计模式。

- 建立 visitor 接口 ParkingVisitor，只有一个方法 double visit(ParkingField pf)。
- 建立其子类 PercentageVisitor，实现该 visit 方法，调用 pf 的方法获得 pf 的内部表示，进行计算。
- 在 ParkingField 接口中增加 accept(ParkingVisitor pv)方法，其实现很简单：pv.visit(this)。

客户端只需要使用 double ratio = pf.accept(new PercentageVisitor())即可。

如果要为 ParkingField 扩展另一个操作呢？只需要构造 ParkingVisitor 的另一个子类，在其 visit()中实现新功能，客户端调用 pf.accept()的时候传入新子类的对象即可。

你可以试一下。

9 使用 Iterator 设计模式：可维护性

ParkingField 需要具备遍历其中所停的所有 Car 对象的能力。拟使用以下形式的 client 端代码，按车辆所在停车位编号由小到大的次序，逐个读取所停车辆。请扩展现有设计方案（修改/扩展哪些 ADT），在下方给出你的设计思路描述，必要时可给出关键代码示例或 UML 类图辅助说明。

```
Iterator<String> iterator = pf.iterator();
while (iterator.hasNext()) {
    String c = iterator.next();
    System.out.println("A car " + c + " is now parked in " + pf);
}
```

正常情况下 Iterator 的泛型应该是 Parkable，为了不给 ParkingField 和 Client 泄露 Parkable，改成了 String。

修改：

- 创建一个 ParkingIterator 类，继承 Iterator<String>
- 构造函数：传递进来 pf 的 status，从中获取停车信息和车位信息，并在其中对车位进行排序；
- 实现 hasNext() 和 next()；
- remove() 要屏蔽掉；
- 修改 ParkingField 接口，使之 extends Iterable<String> 接口
- 在 ConcreteParkingField 中 override iterator() 方法，逻辑是返回一个 ParkingIterator 对象，将 this.status 传递进去。
- 将上面的代码加入 client，即可执行。

```
@Override
public Iterator<String> iterator() {
    return new ParkingIterator(this.status);
}
```

该设计模式也是相当于将排序和遍历功能 delegate 给了外部的 ParkingIterator。后续如果要修改遍历的策略，例如更改排序次序、更改遍历间隔等，都可以扩展出新的 ParkingIterator 实现类即可，无需修改 ParkingField 本身。

10 使用 Strategy 设计模式：可扩展性、可维护性

ParkingField 接口中定义了一个方法 void parking(String type, String plate, int width)，与另一个带有 num 车位号的 parking 方法相比，使用该方法的 client 端无需提供“停车位号码”信息，而是在方法内部自动进行空闲停车位的选择。现实中有不同的停车位选择方法，例如：

- (1) 随机选择一个空闲的、宽度大于车辆宽度的停车位；
- (2) 根据停车位编号，优先选择编号最小的空闲停车位，且其宽度大于车辆宽度。

使用 **Strategy** 设计模式改造现有设计：

在客户端代码调用 `void parking(String type, String plate, int width)` 的时候，如何动态传入某个特定的停车位选择方法？

本质上，相当于写两个停车位选择算法，但不直接在 `ConcreteParkingField` 里写，而是 **delegate** 出去到外部的 **Strategy** 具体实现类，跟上面的几个设计模式非常像。

修改：

- 建立 `ParkingStrategy` 接口，只有一个方法 `selectLot()`，参数是 `parkingField` 的相关信息；
- 实现该接口的两个子类 `RandomParkingStrategy`、`SequentialParkingStrategy`，分别在该方法中实现不同的算法；
- 为 `ParkingField` 增加 `setParkingStrategy(ParkingStrategy ps)` 的接口方法，在实现类中增加 `rep`，传入 `delegation` 关系。`ps` 可以设定缺省值。
- 在 `ParkingField` 的 `parking()` 方法中，利用传入的 `ps` 进行车位选择、停车；
- 客户端：构造 `ParkingStrategy` 对象，然后传入已有的 `pf`，然后即可调用 `parking()` 方法。

```
pf.setParkingStrategy(new RandomParkingStrategy());
```

11 基于语法的输入：可维护性

停车场管理系统启动时，主程序读入外部文本文件，构造多个 `ParkingField` 对象。该文本文件遵循特定的语法格式，每个以 `PF` 开头的行代表一个 `ParkingField` 对象，语法说明如下所示。车位编号为从 1 开始的自然数，车位宽度=常数。

- (1) `PF ::=` 一个由 `<>` 括起来的字符串，分为三部分，分别代表停车场名字、最大车位数、公司名字，三部分之间由逗号“,”分割。
- (2) 停车场名字 `::=` 字符串，长度不限。可以由一个单词或多个单词构成，单词由字母或数字构成，单词之间只能用一个空格分开。
- (3) 停车场最大车位数 `::=` 自然数，其值最小为 5。不能为 `012`、`0012` 的形式，只能为 `12` 的形式。
- (4) 公司名字 `::=` 与停车场名字的语法规则一致，但可以为空。若该部分为空，表示该停车场没有公司管理（即公共停车场）。

以下是三个例子：

```
PF::=<92 West Dazhi St,120,HIT>
```

```
PF::=<Expo820Roadside,10,> //无公司管理的停车场，最后一个逗号之后为空
```

```
PF::=<73 Yellow River Rd,50,Harbin Institute of Technology>
```

该任务非常简单，只要能写出正则表达式，即可从文件中每一行解析出停车场的相关信息，构造 `ParkingField` 对象。

新建一个 `TextFileReader` 类，有唯一的静态方法：

```
List<ParkingField> constructParkingFields(String filename)
```

逐行读取、匹配、解析，构造对象，加入列表。

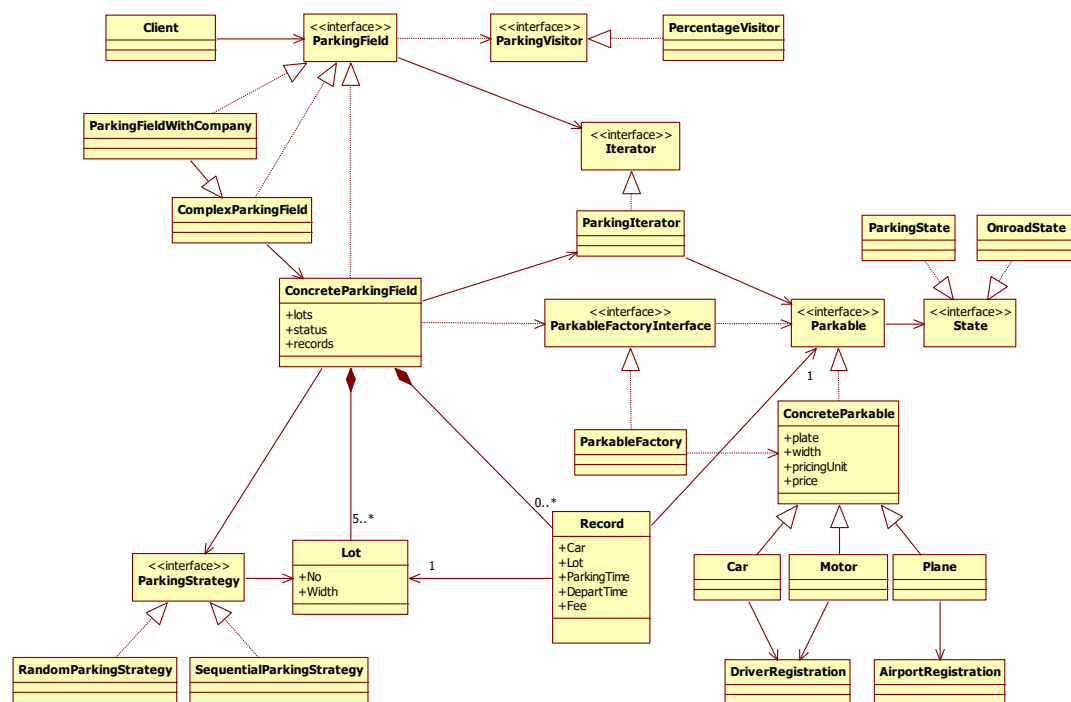
客户端代码：

```
List<ParkingField> pfs =  
    TextFileReader.constructParkingFields("d:\\example.txt");
```

目前的代码：遇到不符合语法的行在控制台提示，忽略该行，进入下一行。

你可以试着改造代码，使之能够找出其中具体不符合语法的部分。


































12 小结



只是粗略的绘制，没有细节。你可以补充完整。

本次习题课的项目结构见下方图，代码可从以下 GitHub 仓库下载：

https://github.com/iccity96/Spring2024_HITCS_SC_Exercise_3/tree/master

- ▼  client
 -  Client
 -  example.txt
 -  TextFileReader
- ▼  decorator
 -  ComplexParkingField
 -  ParkingFieldWithCompany
- ▼  factory
- ▼  field
 -  ConcreteParkingField
 -  Lot
 -  ParkingField
 -  ParkingIterator
 -  Record
- ▼  parkable
 -  AirportRegistration
 -  Car
 -  ConcreteParkable
 -  DriverRegistration
 -  Motor
 -  Parkable
 -  Plane
- ▼  state
 -  OnroadState
 -  ParkingState
 -  State
- ▼  strategy
 -  ParkingStrategy
 -  RandomParkingStrategy
 -  SequentialParkingStrategy
- ▼  visitor
 -  ParkingVisitor
 -  PercentageVisitor