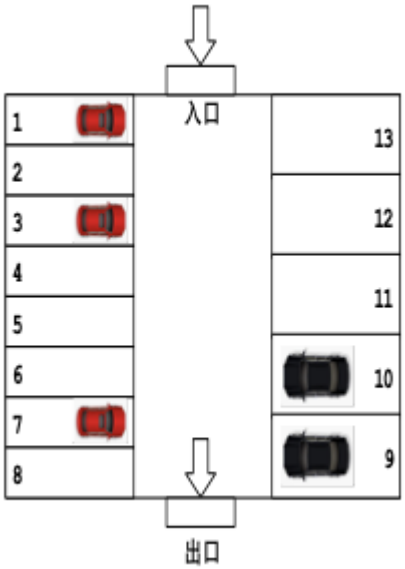
1. Problem

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>=5)，不同停车场包含的车位数目不同。 (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。



客户端程序功能要求:

- 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. 设计思路

- 使用OOP的思路, 寻找这里的“名词”:
    - 停车场: 车位数目
    - 停车位: 编号、宽度

    - 车: 车牌号、宽度
    - 一次停车（从入场到出场）：车、入场时间、出场时间、所在停车位、费用

- “动词”:

    - 构造“停车场”
    - 构造“车”
    - 停车
    - 驶离
    - 计费
    - 查看状态

- 哪些名词可以作为Object？哪些名词作为其他Object的属性？

    - 停车场: 车位数目、具体哪些车位
    - 停车位: 编号、宽度
    - 车: 车牌号、宽度
    - 一次停车（从入场到出场）：车、入场时间、出场时间、所在停车位、费用——==可变的，不是一下子能构造出来==（进入的时候要记录时间/车位、出去的时候要记录时间，目的是为了计算费用）

- 设计原则: 尽可能缩小\mutable的范围

    - 一次停车（从入场到出场）：车、入场时间、出场时间、所在停车位、费用 这些可变信息在哪里管理?

    > 单独造一个 mutable的 class `Record`，单独管理它。

- 谁来管理所有停车记录？客户端程序？no，客户端只面向具体功能，不能让其管理这些。

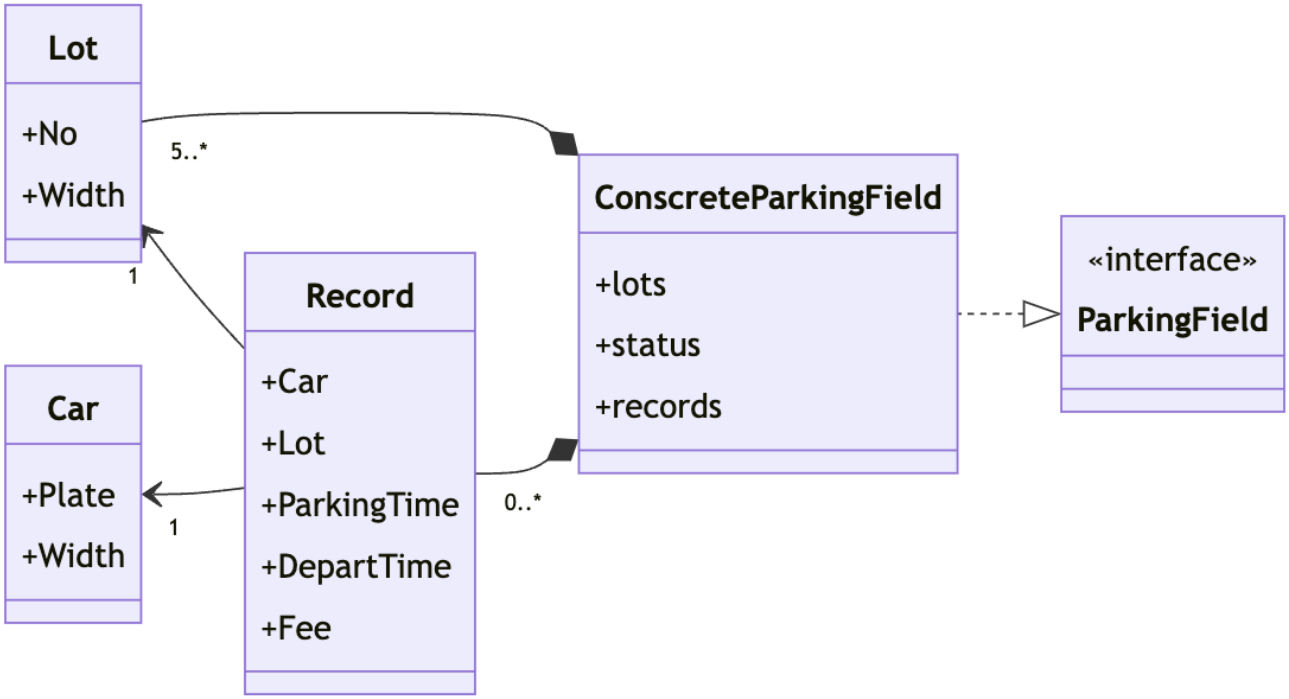
    - 方案1: 造一个类，专门负责管理停车记录
        - 优点:
            - 集中管理：集中管理停车记录，使得对停车过程的查询、修改和统计变得更加集中和方便。
            - 解耦：将停车记录的管理从车辆、停车位和停车场的实现中解耦，有助于减少类之间的依赖，简化系统设计。
            - 易于扩展：当需要添加新的功能（如不同类型的计费方式）时，只需修改这个类，而不会影响到其他类。
        - 缺点:
            - 数据一致性维护：需要确保停车记录的准确性和及时更新，否则可能会导致数据不一致的问题。
            - 复杂性: 多了一个类，稍微复杂
    - 方案2: 让“车”管理自己的状态：停在哪个车场的车位、何时进、何时出。
        - 优点:
            - 自管理：车辆自己管理自己的状态，可以直接从车辆对象获取其停车信息，简化了信息的获取流程。
        - 缺点:
            - 过重的职责：车辆对象除了维护自己的基本信息外（不可变），还要管理停车状态（可变），这可能会使车辆类过于复杂。可变和不可变混在一起

        - 数据访问限制：如果想要从停车场或停车位的角度获取信息，可能会需要额外的步骤或数据结构。
    - 方案3: 让“车位”管理状态：谁目前停在“我”的位置上?
        - 优点:
            - 直观的状态管理：车位管理自己的状态（是否被占用，以及被哪辆车占用）可以直接反映停车场的当前状况。
            - 简化查询：对于查询某个车位是否空闲或是被哪辆车占用，这种方法可以直接提供答案。
        - 缺点:
            - 有限的信息：车位只能提供关于当前停车状态的信息，对于历史记录或者车辆的其他信息（比如停车时间），这种方法可能就不够用了。
            - 可变与不可变混在一起
    - 方案4: 让“”停车场“管理状态：谁目前正停在我这里。
        - 优点:
            - 全局视角：停车场管理所有车位和车辆的状态，可以提供一个全局的视角，便于实现对停车场整体状态的监控和管理。
            - 灵活的管理：可以更灵活地实现各种管理策略，如优先填充空闲车位、特定车位预留等。
        - 缺点:
            - 复杂性增加：停车场需要处理更多的逻辑，如车位的分配、车辆的进出记录等，这可能会使停车场类变得相对复杂。
            - 可变与不可变混在一起

> 每种方法都有其适用场景和局限性。在实际开发中，可能需要根据具体需求和预期的系统规模、性能要求等因素，综合考虑使用哪一种或哪几种方法来设计系统。 ==!!Trade-off!!==

习题课上: 按第4种方式实现!!!!

- ADT设计: 接口还是抽象类、具体类?
    - 本例中，只有一个应用，不存在复用，可以直接写具体类
    - 但是，考虑到将来可能面临的变化，以及不同的实现方式，最好按接口方式设计ADT
    - 接口: ParkingField
    - 类：ConcreteParkingField、Lot、Car、Record（后三者比较简单，Lot和Car都是immutable的，不用接口）



3. 设计ParkingField接口, 撰写Spec

功能需求:

- 构造一个停车场 -- 构造函数constructor, 见下一节
- 构造若干台车 -- 和 ParkingField 无关
- 依次将车停进停车场，==可以指定车位，也可以不指定车位==——==mutator==，两个方法，`parking(Car c, Lot lot)`、`parking(Car c)`，增加停车记录。第一个方法的`Lot`参数，是否有必要暴露给客户端使用？建议改为`int num`，用停车位编号。同样的道理，如果不希望把`Car`暴露出去，可改成`parking(String plate, int width, int num)`,`parking(String plate, int width)`
- 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——mutator， `double depart(String plate)`，增加离场记录，并计算费用。如果觉得两项功能耦合在一起，可以拆开：`void depart(String plate)`、`double calcFee()`。但计算费用这个方法无需对外暴露，所以不能作为接口的公共方法。
- 查看当前停车场的状态——observer，`xxx status()`。希望返回什么数据结构？最适合的应该是`Map`结构，Key是车位，Value是停的车，Value可以为空，所以`Map<Integer, String> status()`，用车位编号作为key，用车牌号作为value。

汇总所有接口方法:

```
create() // 静态工厂方法, construction
void parking(String plate, int width, int lot) // mutator, 按车位停车
void parking(String plate, int width) // mutator, 随机分配车位停车
double depart(String plate) // mutator, 离场计费
Map<Integer, String> status() // observer
```

分别设计它们的Spec

需求中的特殊情况怎么处理?

- 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

建议处理方式:

- 更强的规约：更弱的前置前置、更强的后置条件，让client使用更容易，把责任放在实现者身上。
- 太弱的spec，client不放心、不敢用 (因为没有给出足够的承诺)。开发者应尽可能考虑各种特殊情况，在post-condition给出处理措施。
- 惯用做法是：不限定太强的precondition，而是在postcondition中抛出异常：输入不合法。
- 例如：针对"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"这种情况，无法限定client已知停车场是否已满，所以不能在pre里写，而是交给post来处理。
- 所以，可以针对上述特殊情况，在post里使用异常，告知client。

4. 为ParkingField接口增加静态工厂方法

增加静态工厂方法

```
/**
     * 创建一个新的停车场实例。
     *
     * 本方法根据提供的停车位编号和相应的宽度信息，初始化一个停车场对象。停车场中的每个
停车位将根据提供的
     * 编号和宽度进行设置，且初始时，所有车位均为空（即没有车辆停放）。
     *
     * @param nos 一个整型数组，表示停车位的唯一编号。这些编号必须是自然数（正整数），
且在数组中不能重复。
     * @param widths 一个整型数组，表示对应编号停车位的宽度（以某个单位表示，例如
米）。
     *               该数组中的元素数量必须与`nos`数组的元素数量完全相同。每个宽度值
必须大于或等于5（包括5）。
     *
     * @return ParkingField 返回一个新初始化的ParkingField对象，该对象包含与
`widths`数组长度相同的停车位数量。
     *                      每个停车位的宽度与`nos`数组中相对应的编号的宽度一致。
     *
     * @throws IllegalArgumentException 如果输入参数`nos`和`widths`的长度不相
等，或者如果`nos`数组中包含重复的编号，
     *                                  或者`widths`数组中的任何宽度小于5，则抛
出此异常。
     * @throws NullPointerException 如果`nos`或`widths`为null，则抛出此异常。
     */
    public static ParkingField create(int[] nos, int[] widths) throws
Exception {
        return new ConcreteParkingField(nos, widths);
    }
```

参数为何这么实现?

- 车场需要包含一组车位，最开始没有停车，所以构造时需要知道多少个车位、每个车位宽度如何。目前用了两个数组，规定了它们的pre-condition。
- 也可以用其他参数策略，例如两个List、一个Map。Map中的key是车位号，value是车位宽度，Map可天然保证key不重复
- 在ParkingField中增加了一个新的create()方法，参数不同，是overload

```
/**
     * 创建一个新的停车场对象。
     *
     * 本方法接受一个Map作为输入，该Map的Key为车位的唯一编号（自然数），值为对应车位的宽度（自然数）。
     * 使用这些信息，方法初始化一个停车场对象，其中包含的车位数量与输入映射的大小相同。
     * 每个车位的编号和宽度将根据映射中的键值对设置，并保证在初始化时所有车位都是空的（即没有车辆停放）。
     *
     * @param lots 一个Map，其中的键（Integer类型）代表车位的编号，值（Integer类型）代表相应车位的宽度。
     *             所有的编号都应为正整数，所有的宽度也都应为正整数。`lots`的大小必须大于或等于5。
     *
     * @return ParkingField 返回一个新初始化的ParkingField对象。该对象根据输入的映射信息包含相应数量`lots.size()`的车位，
     *                      每个车位的编号和宽度与输入映射中的对应键值对一致。
     *
     * @throws IllegalArgumentException 如果输入的映射`lots`的大小小于等于5，或者映射中的任何键（车位编号）或值（车位宽度）
     *                                  不是正整数，则抛出此异常。
     * @throws NullPointerException 如果`lots`为null，则抛出此异常。
     */
    public static ParkingField create(Map<Integer, Integer> lots) throws Exception {
        return new ConcreteParkingField(lots);
    }
```

5. 针对ParkingField接口设计测试用例

根据各个方法的spec设计测试用例，撰写Testing Strategy，写出测试用例代码

静态工厂方法和实例方法的测试，分开。让测试类的职责更清晰。

针对静态工厂方法：`ParkingFieldStaticTest.java`

调用`create(int[] nos, int[] widths)`或`create(Map<Integer, Integer> lots)`创建一个新停车场对象 如何观察它？目前的接口方法里没有足够的观察者。 要能够检查所有post-condition是否满足。 要观察什么：

- 有几个车位；

- 参数里的每个“车位编号+车位宽度“是否包含；
- 每个车位上是否没有停车

已有的一个observer方法`public Map<Integer, String> status()`，可以用来间接实现(1)(3)，但无法实现(2)。 所以给ParkingField增加三个observer方法：

```
public int getNumberOfLots();   //非必须，可以用status().size()
public boolean isLotInParkingField(int num, int width);
public boolean isEmpty(); //非必须，可以检查status()返回值的每个value是否为“”。
```

将它们补充到接口里，写出spec。

如何为create设计测试用例？——以下都用`create(Map ..)`为例

```
/**
 * ParkingField.create(Map<Integer, Integer> lots) 方法的测试策略。
 * <p>
 * 测试策略基于以下组合情况覆盖各种场景：
 * 1. 车位数量：
 *    - 使用0个车位测试，确保对空映射输入的异常处理。
 *    - 使用少于5个车位测试，确保对车位不足的异常处理。
 *    - 使用正好5个车位测试，验证方法的功能性。
 *    - 使用超过5个车位测试，验证方法的功能性。
 * <p>
 * 2. 车位编号（映射中的键）：
 *    - 使用全部自然数测试，确保正常执行。
 *    - 包含非自然数（零、负数）测试，确保异常处理。
 * <p>
 * 3. 车位宽度（映射中的值）：
 *    - 使用全部自然数测试，确保正常执行。
 *    - 包含零或负数测试，确保异常处理。
 * <p>
 * 4. 特殊情况：
 *    - 测试所有车位宽度相同的情况。
 *    - 测试车位宽度不同的情况。
 * <p>
 * 5. 特殊输入：
 *    - 测试输入映射为null的情况，确保抛出NullPointerException。
 * <p>
 * 测试设计至少覆盖每个值场景一次，对车位编号和宽度的维度考虑了笛卡尔积全覆盖，特别是对有效和无效输入组合的情况。
 */
```

设计测试用例:

- 空
- <1,10><2,12><3,15><4,20>
- <1,11><2,12><3,13><4,14><5,15>
- <1,11><2,12><3,13><4,14><5,15><6,16>

- <-1,10><2,12>
- <1,-10><2,12>

为每个测试用例写测试函数，函数前写覆盖了哪个分类，例如

```
// 测试用例1：lots为空，期望抛出IllegalArgumentException
@Test
void testCreateWithEmptyLots() {
    Map<Integer, Integer> lots = new HashMap<>();
    assertThrows(IllegalArgumentException.class, () -> {
        ParkingField.create(lots);
    });
}
```

针对其他方法：`ParkingFieldInstanceTest.java`

针对parking函数，输入车牌号、车宽、车位号。按上述过程设计测试用例 ==前提：用create()静态方法创建一个车场==

观察其结果是否正确：

- 车牌号为plate的车辆，之前没停在车场，执行后停在了车位号为num的车位上
- 该车位宽度大于等于车宽度
- 其他车位的状态不变

增加观察者方法?

- `boolean isParkingOn(String plate, int width, int num)` 车辆`plate`是否停在车位`num`上。该函数非必须，可以对`status()`返回的`KV`进行查询得到结果。
- `int getLotWidth(int num)` 返回某车位宽度，判断宽度是否足够该车。
- 针对“其他车位状态不变”，可以在`parking()`调用前后分别调用`status()`返回的KV对，逐个作比较。也可以先取出每个车位上的停车情况，执行`parking`后比较结果是否变化，为此需要增加观察者方法 `String getCarOnLot(int num)`，返回停在num车位上的车牌号，没有车则返回“”。

在类的开始位置写testing strategy:

```
/**
 * 测试策略
 * <p>
 * 按照`plate`（车牌号）划分：
 * 1. 该车已经停在该停车场：尝试将一辆已经在停车场中的车辆再次停入，预期抛出
IllegalStateException。
 * 2. 该车未在停车场：尝试将一辆不在停车场中的车辆停入，进一步按照下列条件测试。
 * <p>
 * 按照`num`（车位编号）划分：
 * 1. 该车位是车场的合法车位：车位编号在停车场的范围内，进一步按照下列条件测试。
 * 2. 不是合法车位：车位编号超出停车场范围，预期抛出IllegalArgumentException。
 * <p>
 * 按照车位占用情况划分：
 * 1. 该车位合法，已被其他车占用：尝试将车辆停到已经有车的车位上，预期抛出
```

```
IllegalStateException。
 * 2. 该车位合法，未被占用：进一步按照车位宽度条件测试。
 * <p>
 * 按照`num`和`width`（车辆宽度）划分：
 * 1. 车位宽度不超过车辆宽度：尝试将车辆停入宽度不足以容纳该车辆的车位上，预期抛出
IllegalStateException。
 * 2. 车位宽度等于车辆宽度：尝试将车辆停入宽度刚好与车辆宽度相等的车位上，预期成功停车。
 * 3. 车位宽度大于车辆宽度：尝试将车辆停入宽度超过车辆宽度的车位上，预期成功停车。
 * <p>
 * 特殊情况测试：
 * 1. `plate`为空：预期抛出IllegalArgumentException。
 * 2. `width`不是正整数：预期抛出IllegalArgumentException。
 * 3. `num`不是正整数：预期抛出IllegalArgumentException。
 *
 */
```

设计测试用例：

- 车场：<1,10><2,15><3,20><4,20><5,20> 利用`setUp()`, 每个测试用例之前自动构建停车场

```
@BeforeEach
void setUp() {
    // 假设停车场有5个车位，编号1到5，宽度依次为10, 15, 20, 20, 20。
    Map<Integer, Integer> lots = new HashMap<>();
    lots.put(1, 10);
    lots.put(2, 15);
    lots.put(3, 20);
    lots.put(4, 20);
    lots.put(5, 20);
    try {
        ParkingField parkingField = ParkingField.create(lots);
    } catch (Exception e) {
        throw new RuntimeException(e);
    }
}
```

为每个测试用例写测试函数，函数前写覆盖了哪个分类。

```
// 初始化车位和宽度的假设设置，假设车位编号从1到5，宽度逐渐增大
// 车位宽度假设为：1 -> 2, 2 -> 2, 3 -> 3, 4 -> 3, 5 -> 4

- parking("PERFECT1", 2, 2)  // 覆盖车未在停车场、车位合法且未被占用，车宽=位宽
    /* 测试预期：成功停车，没有异常抛出。 */

- parking("ABC123", 2, 1) followed by parking("ABC123", 2, 2)  // 覆盖车已在
停车场、车位合法且未被占用
    /* 测试预期：第二次调用时抛出IllegalStateException。 */

- parking("XYZ789", 2, 99)  // 覆盖车未在停车场、车位不合法
```

```
    /* 测试预期: 抛出IllegalArgumentException。 */

- parking("SMALL100", 2, 5)  // 覆盖车未在停车场、车位合法且未被占用，车宽<位宽
    /* 测试预期: 成功停车，没有异常抛出。 */

- parking("CAR456", 2, 3) followed by parking("NEW789", 2, 3)  // 覆盖车未在
停车场、车位合法且已被占用
    /* 测试预期: 第二次调用时抛出IllegalStateException。 */

- parking("BIG999", 4, 1)  // 覆盖车未在停车场、车位合法且未被占用，车宽>位宽
    /* 测试预期: 抛出IllegalStateException，因为车位宽度不足以容纳车辆。 */
```

6. 编写具体实现类 ConcreteParkingField

Rep里要表达什么？内部存储数据结构 要存储什么？——

- 车位数：int即可
- 哪些车位：集合类List, Set等
- 当前车位占用情况（哪个车位停了哪个车、或者为空）——Map<>，key为车位，value为车
- 停进来的车何时进入的、何时离场的、在哪个车位停（要计费）——List<Record>

说明：车位数可以不用单独保存，可以从其他属性中间接得到 所以，一种可行的rep：

```
List<Lot> lots;  //一组车位
//占用情况。如果某lot上没有车，那么存什么?
// null不好，因为当status.get(lot)返回null的时候，无法判断是因为没停车还是因为没有这
个lot。
//因此没有车的时候不要放在status里。
Map<Lot, Car> status;
List<Record> records; //停车记录
```

你是否还有其他方案？可以写出不同的实现类

设计AF、RI

```
  /**
     * Abstraction Function (AF):
     * AF(c) = 一个停车场，如果c.lots为空，则代表一个没有车位的停车场；否则，对于每个
非空的Lot l ∈ c.lots,
     * 表示一个具有编号l.getNumber()和宽度l.getWidth()的车位。如果c.status包含l作
为键,
     * 则该车位被c.status.get(l)表示的Car占用。c.records表示该停车场的所有停车记
录，其中每个Record r
     * 描述了一次停车行为，包括车辆r.getCar()在时间r.getTimeIn()时停入车位
r.getLot()，并在r.getTimeOut()时离开,
     * 该次停车花费了r.getFee()元。
     * <p>
     * Representation Invariant (RI):
```

```
     * - c.lots.size() >= 5 表示停车场至少有5个车位。
     * - c.lots.size() >= c.status.size() 确保车位数不少于已停车辆数。
     * - c.status中的每个键均为c.lots中的元素，保证每个已占用的车位都是有效车位。
     * - c.status中的值（Car对象）之间不重复，确保每辆车只占用一个车位。
     * - 对于c.status中的每个条目<key, value>，value（Car对象）的宽度不大于
key（Lot对象）的宽度，
     * 保证车辆可以适合其车位。
     * - 对于c.records中的每个Record对象r，如果r.getTimeOut()为空，则必须有一个与
之对应的条目<key, value>在c.status中，
     *   其中key为r.getLot()且value为r.getCar()，表示正在停车中的记录必须与当前占
用状态一致。
     */
```

实现checkRep()

按上面RI，逐条翻译过来即可 将`checkRep()`加到每个方法`return`之前。

```
private void checkRep() {
  assert lots.size() >= 5 : "停车场至少应有5个车位。";
  assert lots.size() >= status.size() : "车位数应不少于已停车辆数。";

  // 检查status中的每个key是否在lots中
  for (Lot lot : status.keySet()) {
      assert lots.contains(lot) : "每个已占用的车位都应为有效车位。";
  }

  // 检查status中的车辆是否不重复
  Set<Car> cars = new HashSet<>(status.values());
  assert cars.size() == status.values().size() : "每辆车只能占用一个车位。";

  // 检查车辆宽度是否适合其车位
  for (Map.Entry<Lot, Car> entry : status.entrySet()) {
      assert entry.getKey().getWidth() >= entry.getValue().getWidth() : "车
辆宽度应小于等于车位宽度。";
  }

  // 检查正在停车中的记录是否与当前占用状态一致
  for (Record record : records) {
      if (record.getTimeOut() == null) {
        // 表示车辆尚未离开
          assert status.containsKey(record.getLot()) &&
          status.get(record.getLot()).equals(
            record.getCar()) :
                "正在停车中的记录应与当前占用状态一致。";
      }
  }
}
```

实现构造函数

目前有两个构造函数，以`ConcreteParkingField(Map<Integer, Integer> lots)`为例

- 在构造函数中，要造出车位，填充到Rep中
- `Status`和`record`在刚启动时为空

实现接口方法

==根据spec和rep逐个实现即可，都不困难==

override `equals()`、`hashCode()` ParkingField是mutable的，所以先不需要写override这两个

toString()

例子：某ParkingField对象的有5个停车位，状态如下表所示。

停车位编号	停车位宽度	当前所停车辆
1	200	车牌号：AB001，宽度180
2	180	空闲
3	200	车牌号：CD002，宽度180
4	170	车牌号：EF003，宽度160
5	190	空闲

从上表看出，该停车场目前有60%的停车位已停车 (=3/5)。针对该例子，你所写的`toString()`的输出结果应为：

```
The parking field has total number of lots: 5
Now 60% lots are occupied
Lot 1 (200):    Car AB001
Lot 2 (180):    Free
Lot 3 (200):    Car CD002
Lot 4 (170):    Car EF003
Lot 5 (190):    Free
```

这是个基本的observer。

7 编写辅助类Car、Lot

Immutable类

设计rep，是否有多种表示？ 设计RI，与Rep密切相关 实现checkRep() 设计方法 override equals()、hashCode()、toString()

==学生自行完成==

8. 编写客户端程序

某个停车场管理系统，实现一系列功能

```
Map<Integer, Integer> lots = new HashMap<>();
lots.put(1, 10);
lots.put(2, 15);
ParkingField pf = ParkingField.create(lots);

pf.parking("HA001", 10, 1);
pf.parking("HA002", 10, 2);
System.out.println(pf);
```

9. 绘制Snapshot Diagram

针对客户端程序运行到某个时刻，想象内存状态，绘制snapshot

在以上主程序执行完之后：

- 首先有一个lots，是个Map，先画出来，分别指向各个Integer；
- 然后创建了一个pf，看它的构造函数，包含三个部分：lots, status, records，都是final的，双线。这三个对象是mutable的，所以单线圈，pf也是单线圈。
- lots有两个对象，指向Lot，Lot内部分别指向number和width，都是直接的值（int）
- status目前为空
- records目前也为空
- 然后执行parking，看它的代码，首先创建了一个Car对象，指向了一个String双线圈和一个width值（无圈）
- 然后在status里加入了一个KV元素, <Lot, Car>，分别指向lot对象和car对象
- 在records里加了一个元素，该元素也指向lot、car对象，并增加了一个Calendar对象（停车时间，单线圈，mutable），离场时间目前指向null，fee指向0。

10. 表示泄漏与安全性

考察`ParkingField`、`Car`、`Lot`、`Record`这些ADT，考虑客户端代码，判断这些ADT是否存在表示泄露？列出可能存在表示泄露的地方，分析其潜在风险，并给出其修改策略。

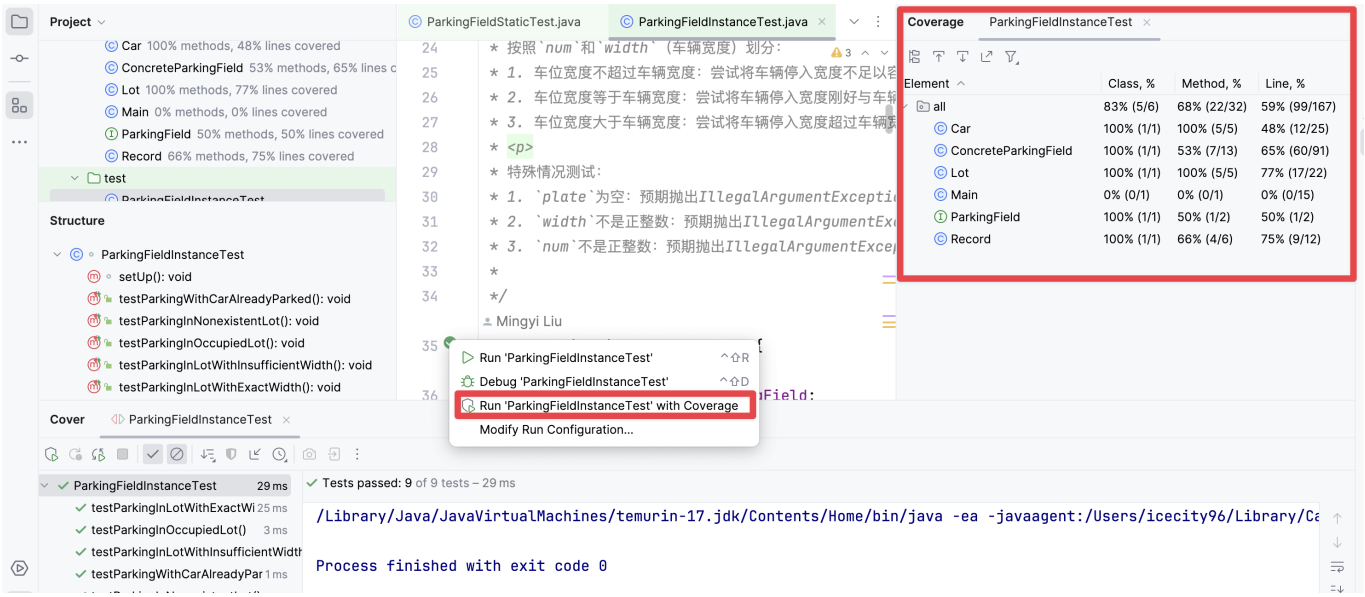
给出几个表示泄露的客户端例子，以及如何通过防御式拷贝策略修改ADT设计。

```
private Calendar timeOut = null;
public Calendar getTimeOut() {
  return timeOut;
}
```

修改后

```
public Calendar getTimeOut() {
    return (timeOut == null) ?
      null : (Calendar) timeOut.clone();
}
```

11. JUnit测试 & 测试覆盖度查看



12. 使用SpotBugs查看代码潜在风险

例如

```
Car doesn't define a hashCode() method but is used in a hashed data
structure [Scariest(1), High confidence]
Comparison of String objects using == or !=  [Troubling(11), Normal
confidence]
```

代码可以从以下GitHub仓库下载：

https://github.com/icecity96/HIT_SC_Exec_2