**实验5** **Java程序规格设计(1)**

### 1.实验目的

（1）能够对方法进行抽象，进行规格撰写

（2）会按照规格说明设计测试用例

### 2. 知识要点

面向对象的最大的优点是以层次化、模块化的方式来设计功能内聚和独立完整的类和方法。随着系统规模的增加，类之间的关系数量也会增加，类、方法之间构成了复杂的协作网。在这种情况下，程序的局部错误往往会导致程序运行时其他对象或方法也出现错误。为了避免出现这种不可控的情况，需要在设计时对类和方法的行为进行严格的限制和要求，这就是课堂上所讲授的规格。通过规格来开展程序设计，可以在类实现之前就确定类及其关系，以及方法的行为要求，可以有效保证软件整体行为的逻辑清晰性和功能稳定性。

类的方法在被调用时存在参数传递，运行时传递的参数会出现多种情况，调用者和被调用者之间就需要有一种沟通机制：被调用者需要事先说明自己对输入参数的要求，也需要在运行时把输入不满足要求的情况及时告知调用者。方法规格通过前置条件（precondition）和后置条件（postcondition）来达到实现这个沟通机制。本课程介绍和训练基于JSF的规格化设计，即通过具有一定规范性的注释来描述方法的规格。其中，前置条件使用“@REQUIRES:…”来描述，后置条件使用“@EFFECTS…”来描述。更一般的，如果调用者满足了被调用者的前置条件要求，则被调用方法的执行就必须要满足其声明的后置条件，否则就是被调用方法运行出现了错误。前置条件、后置条件和方法执行时的副作用一起形成了方法规格。方法规格不仅对于程序设计和实现具有重要意义，同样也是测试活动的重要依据。可以针对规格的描述进行重点测试前置条件违反的情况，以及满足前置条件时检查后置条件是否满足等。

###### 2.2.1前置条件pre-condition

方法执行之前必须要满足的条件，规定了方法必须正确处理的有效输入范围。方法A在调用另外一个方法B之前需要满足B所要求的前置条件，否则B可以不对A输入的数据进行处理。相应的，如果满足了B所要求的前置条件，则B必须对A输入的数据进行正确处理。

###### 2.2.2后置条件post-condition

### 后置条件是对方法执行之后必须达到的效果的描述。若一个方法在被调用时其前置条件得到了满足，则执行后必须满足其后置条件。如果说前置条件是一个方法对调用者的要求，那么后置条件则是对调用者的承诺。如果调用者满足了被调用者的前置条件，被调用者的执行就必须满足所承诺的后置条件，否则视为被调用者执行有错误。

###### 2.2.3 副作用side-effects

方法在执行过程中不可避免要修改方法所在类的属性，或者传递进来的对象，甚至是全局对象。这些修改会对其他对象的行为产生影响。因此，方法规格应明确执行过程要对哪些数据进行修改。作为规格的组成部分，描述副作用的目的不是简单记录实际执行修改了哪些数据，而是在实现之前，从设计角度就规定要修改哪些数据。在实现时，必须严格按照要求来修改相关数据，多修改和少修改都将给程序其他对象带来不确定的影响。副作用使用”@MODIFIES…“注释来加以描述。

### 3. 实验作业

以下代码完成了本学期第三次作业要求的电梯ALS调度的具体要求

* 实现了类：电梯、请求队列、调度器、请求类
* 固定为10层楼，一层只有向上请求按钮，10层只有向下请求按钮，其他有两个请求按钮（向上和向下）；电梯内部有只有楼层号对应的按钮
* 电梯一开始停靠在一层
* 输入为按照请求产生时间排序的请求序列

(FR, n, UP/DOWN,t)

(ER,n,t)

t为请求产生的相对时刻，第一个请求的t为0。假设电梯运行一个楼层距离消耗时间为0.5；停靠、开关门等一系列动作消耗时间为1。

* 输出为电梯经过的楼层和停靠楼层

(n, UP/DOWN, P/S,t):n为楼层号，UP/DOWN为电梯运行方向，P为经过，S为停靠，t为观察到电梯相应状态的时刻。

* 依次列出“顺路捎带”的请求

除了“输出为按照时间排序的电梯运动停靠、运动方向及时间”外，按照电梯实际响应情况，输出主响应请求和捎带的响应请求。

* ALS\_Schedule (A Little Smart Schedule)
  + 只要队列不为空，每次都取出队列头请求来调度
  + 电梯在运动过程中不能突然改变运动方向
  + 在调度电梯完成一个请求的过程中，可以让电梯完成“顺路”的楼层请求
  + 调度算法要确保电梯在当前运动方向上完成所有能完成的电梯内请求

请理解给出的程序，并仔细阅读这五个函数的规格或代码：

（1）对于Elevator类的**moveUP( )**、**moveforQuery( )**方法、请根据它的代码，使用JSF规范补充给出它的规格说明

（2）对于QueryList类的**append( )**、Elevator的**pickupQuery( )**、**checkFinishedQuery( )**方法，请根据所给出的规格说明，补充或修改其代码

（3）以上五个方法，请对照规格说明设计几组测试用例（**测试用例仅针对具体方法，参见下面给出的例子**），使得你**修改过的代码能够通过测试**或你**撰写的规格能覆盖代码的所有执行路径**。

例：以下方法为请求队列QueueList中的一个方法，完成从队列中删除指定位置的请求。

public boolean remove(int index) throws EmptyQueueException, InvalidIndexException {

/\*@MODIFIES:this

@EFFECTS:

normal\_behavior

(\old(this).get(index) !=null) ==> (this.size == \old(this).size-1) && (this.contains(\old(this.get(index)))==false) && (\result==true) ;

(\old(this).size ==0)==>exceptional\_behavior(EmptyQueueException)

(index >=\old(this).size) ==> exceptional\_behavior (InvalidIndexException);

(index < 0) ==> exceptional\_behavior (InvalidIndexException);

\*/

try {

queue.remove(index);

} catch(Exception e) {

return false;

}

return true;

}

编写测试用例

public class TestQueueList {

public static void main(String[] args) throws UnsortedException {

QueryList ql=new QueryList(10,1);

System.out.println("remove from empty queue:"+ql.remove(3));

ql.append(new Query(10,0.0));

ql.append(new Query(8,1.0,Query.Direction.UP));

ql.append(new Query(5,1.5,Query.Direction.DOWN));

System.out.println("remove an nonexistent element from the queue:"+ql.remove(-1));

System.out.println("remove the first element from the queue:"+ql.remove(0));

System.out.println("remove an exceeded element from the queue:"+ql.remove(4));

}

}