类实现正确性的论证

Elevator类：

1. 抽象对象得到了有效实现论证

/\*\*

\* @Overview: Elevator class is a model of elevator, it contains two possible

\* request position and records floor now the elevator is at, time,

\* and difference between the floor before between now floor. It deals

\* with coming request and stored request. It provides methods to

\* check information.

\*/

private int floorNow; 对应overview中记录当前楼层的变量

private double clock; 对应overview中time，记录电梯时间

private Request lastRq; 储存指令的第一个空位

private Request lastRq2; 储存指令的第二个空位

private int diff; 记录上一次停留的楼层和当前楼层的差距。

根据上面的叙述，类的overview明确了抽象对象，且类的rep能够通过抽象函数映射到相应的抽象对象。

1. 对象有效性论证
2. 针对构造方法，论证对象的初始状态满足不变式，即repOK为真。

Elevator提供了一个构造方法，Elevator ()，它初始化全部的rep。

/\*\*

\* @REQUIRES: None;

\* @MODIFIES: this;

\* @EFFECTS: clock == 0.0; floorNow == 1; lastRq == null; lastRq2 == null; diff

\* == 0;

\*/

public Elevator() {

floorNow = 1;

clock = 0.0;

lastRq = null;

lastRq2 = null;

diff = 0;

}

repOK的运行结果显然返回true。

1. 逐个论证每个对象状态更改方法的执行都不会导致repOK的返回值为false。

Elevator提供了三个状态更新方法：resetClock、move、act，下面逐个进行论证。

* 假设resetClock(Request rq)方法开始执行时，repOK为true。

1. resetClock方法首先获取传入的请求的时间属性，与当前时间进行对比，如果请求的时间属性大于当前的电梯时间，判断请求是否为主请求。若果请求是主请求，就把请求的时间赋给当前时间clock。如果不是主请求，判断这个捎带请求的对应主请求的时间是否大于当前的电梯时间，如果是，就把这个捎带请求的对应主请求的时间赋给当前时间clock。不管是请求的时间，还是请求的对应主请求的时间都是大于0的。并且该函数确保了，clock只增不减，满足repOK对ckock的限制
2. 因此，该方法的执行不会导致repOK为假，不违背表示不变式。

* 假设move(Request rq)方法开始执行时，repOK为true。

1. 首先调用resetClock(rq);根据之前的论证，可以判断出来，调用完毕之后，不会使repOK为假。
2. 接下来判断传入的请求是不是主请求，如果是一条捎带请求，先判断电梯内是否已经储存了请求。如果没有，把这条请求留下，放到第一个空位上。repOK依然为true；
3. 如果已经储存了请求，判断已经储存的这一条请求与传入的请求的目标是否为同一层，如果是同一层，说明这是在一个主请求捎带的两个不同但是目标同一层的请求。此时，把传入的请求也储存起来，放到第二个位置（根据后面的论证可以知道，此时第二条的位置一定为空）。由于Queue的取请求方式可以保证，既然这两条目标相同，那么一定是先输入的请求先进入电梯，即lastRq的order小于LastRq2的order。对repOK的变量未进行修改，repOK依然为true；如果两个不相同，意味着此时，此时第一条已经可以执行。判断第二条是否已经储存，如果储存，执行储存的与第一条，把储存的第一条连同第二条输出并清空，然后把传入的指令放到第一个空位上。第二条未储存的话，执行第一条，并把第一条输出且清空，此时把传入的指令放到第一个空位上，作为新的储存。由于电梯执行命令，放生上下移动，此时会使时间变大，同时，还要加上开关门的1s但是，由于传入的请求getFloor所得楼层均为1-10层，所以floorNow满足，同时clock，只有可能变大或不变，故repOK仍然满足。
4. 对于传入的请求是主请求时，如果电梯里面没有请求，说明是这条指令之前也是一条主请求，所以直接执行并输出此请求。
5. 如果有储存的请求，根据前面的论述，电梯里面储存的要么是两条目标楼层一样的捎带请求，要么只有一条稍带请求。如果储存的请求和传入的请求目标楼层不同，那么先执行并输出储存的请求再执行并输出传入的主请求。同时增加停靠开关门的时间。如果储存的请求和传入的请求目标楼层相同，一并执行并输出传入的主请求和储存的捎带请求。最后清空第一个和第二个位置。同样由于传入的请求getFloor所得楼层均为1-10层，所以floorNow满足，同时clock，只有可能变大或不变，故repOK仍然满足。
6. 方法执行结束，因此该方法执行不会导致repOK为假。

* 假设act(Request rq)方法开始执行时，repOK为true。

1. act函数执行请求，首先将diff设置为当前楼层和目标楼层的差。把当前楼层设置为请求的目标楼层。最后把clock加上运动楼层差乘以0.5。由于传入的请求getFloor所得楼层均为1-10层，所以floorNow满足，同时clock，只有可能变大或不变，故repOK仍然满足。
2. 方法执行结束，因此该方法执行不会导致repOK为假。
3. 该类的其他几个方法的执行皆不改变对象状态，因此这些方法执行前和执行后的repOK都为true。
4. 综上，对该类任意对象的任意调用都不会改变其repOK为true的特性。因此该类任意对象始终保持对象有效性。
5. 方法实现正确性论证

目标：在方法的前置条件和后置条件没有逻辑矛盾的前提下，根据前置条件和后置条件分析整理出完整的{<effect Y> with <input partition X>}，然后对照方法代码实现来论证给定每个<partition X>，方法执行结果都满足<effect Y>。

1. predictTime(Request mainRq, Request rq):

/\*\*

\* @REQUIRES: mainRq!=null; rq!=null;

\* @MODIFIES: None;

\* @EFFECTS: \result == ((double) mainRq.getTime() > clock ? (double)

\* mainRq.getTime(): clock) + (Math.abs(floorNow - rq.getFloor())) \*

\* 0.5 + lastRq != null && rq.getFloor() != floorNow &&

\* lastRq.getFloor() != rq.getFloor()? 1: 0;

\*/

根据上述过程规格，获得如下的划分：

<\result == (double) mainRq.getTime() + (Math.abs(floorNow - rq.getFloor())) \*0.5 + lastRq != null && rq.getFloor() != floorNow &&lastRq.getFloor() != rq.getFloor()? 1: 0;> with <(double) mainRq.getTime() > clock>

<\result == clock + (Math.abs(floorNow - rq.getFloor())) \*0.5 + lastRq != null && rq.getFloor() != floorNow &&lastRq.getFloor() != rq.getFloor()? 1: 0;> with <(double) mainRq.getTime() <= clock >

* 方法首先声明一个临时变量temClock。
* 然后对于(double) mainRq.getTime()和clock两个值进行判断。
* 如果(double) mainRq.getTime()>clock，将temClock赋值为(double) mainRq.getTime()，然后加上差值(Math.abs(floorNow - rq.getFloor())) \* 0.5，如果lastRq != null && rq.getFloor() != floorNow && lastRq.getFloor() != rq.getFloor()满足，再加上代表停留的一秒种。最后输出临时变量的值。因此，满足<\result == (double) mainRq.getTime() + (Math.abs(floorNow - rq.getFloor())) \*0.5 + lastRq != null && rq.getFloor() != floorNow &&lastRq.getFloor() != rq.getFloor()? 1: 0;> with <(double) mainRq.getTime() > clock>
* 如果(double) mainRq.getTime()<=clock，将temClock赋值为clock，然后加上差值(Math.abs(floorNow - rq.getFloor())) \* 0.5，如果lastRq != null && rq.getFloor() != floorNow && lastRq.getFloor() != rq.getFloor()满足，再加上代表停留的一秒种。最后输出临时变量的值。因此，处理满足<\result == clock + (Math.abs(floorNow - rq.getFloor())) \*0.5 + lastRq != null && rq.getFloor() != floorNow &&lastRq.getFloor() != rq.getFloor()? 1: 0;> with <(double) mainRq.getTime() <= clock >

1. resetClock(Request rq)

/\*\*

\* @REQUIRES: rq!=null;

\* @MODIFIES: clock;

\* @EFFECTS: ((double) rq.getTime() > clock && rq.getMt() == 1) ==> (clock ==

\* (double) rq.getTime()); ((double) rq.getTime() > clock &&

\* rq.getMt() == 0 && (double) rq.getFt() > clock) ==> (clock ==

\* (double) rq.getFt());

\*/

根据上述过程规格，获得如下的划分：

<do nothing> with < (double) rq.getTime() <= clock || (double) rq.getTime() > clock && rq.getMt() == 0 && (double) rq.getFt() > clock >

< clock = (double) rq.getTime()> with <(double) rq.getTime() > clock && rq.getMt() == 1>

< clock = (double) rq.getFt() > with <(double) rq.getTime() > clock && rq.getMt() == 0 && (double) rq.getFt() > clock>

* 首先检查(double) rq.getTime() 与 clock大小关系，如果(double) rq.getTime() > clock，并且rq.getMt() == 1即rq为主请求，进行赋值clock = (double) rq.getTime()，因此处理满足< clock = (double) rq.getTime()> with <(double) rq.getTime() > clock && rq.getMt() == 1>
* 如果不是主请求，并且(double) rq.getFt() > clock其对应的主请求的时间>clock，进行赋值clock = (double) rq.getFt()，因此，处理满足< clock = (double) rq.getFt() > with <(double) rq.getTime() > clock && rq.getMt() == 0 && (double) rq.getFt() > clock>
* 对于其他情况，包括(double) rq.getTime() <= clock，(double) rq.getTime() > clock && rq.getMt() == 0 && (double) rq.getFt() > clock两种情况，不进行处理。所以，满足<do nothing> with < (double) rq.getTime() <= clock || (double) rq.getTime() > clock && rq.getMt() == 0 && (double) rq.getFt() > clock >

1. move(Request rq)

/\*\*

\* @REQUIRES: rq!=null;

\* @MODIFIES: this;

\* @EFFECTS: (According to whether rq is a main request(whether rq.getMt() == 1) and whether the elevator has stored one or two request, act the request or store it in lastRq or lastRq2)

\*/

根据上述过程规格，获得如下的划分：

<resetClock(rq) && (According to whether the elevator has stored one or two request and whether stored request’s floor equals rq’s floor, store it in lastRq or lastRq2 or act stored requests) > with < rq.getMt() == 0>

<resetClock(rq) && (According to the orders act stored requests and the main request) > with < rq.getMt() == 1>

* 首先resetClock(rq)设置时间，处理电梯长期停用后再次起用的情况。
* 判断是否为主请求，如果是一条捎带请求，先判断电梯内是否已经储存了请求。如果没有，把这条请求留下，放到第一个空位上。
* 如果已经储存了请求，判断已经储存的这一条请求与传入的请求的目标是否为同一层，如果是同一层，说明这是在一个主请求捎带的两个不同但是目标同一层的请求。此时，把传入的请求也储存起来，放到第二个位置（根据后面的论证可以知道，此时第二条的位置一定为空）。由于Queue的取请求方式可以保证，既然这两条目标相同，那么一定是先输入的请求先进入电梯，即lastRq的order小于LastRq2的order。如果两个不相同，意味着此时，此时第一条已经可以执行。判断第二条是否已经储存，如果储存，执行储存的与第一条，把储存的第一条连同第二条输出并清空，然后把传入的指令放到第一个空位上。第二条未储存的话，执行第一条，并把第一条输出且清空，此时把传入的指令放到第一个空位上，作为新的储存。因此满足处理<resetClock(rq) && (According to whether the elevator has stored one or two request and whether stored request’s floor equals rq’s floor, store it in lastRq or lastRq2 or act stored requests) > with < rq.getMt() == 0>
* 对于传入的请求是主请求时，如果电梯里面没有请求，说明是这条指令之前也是一条主请求，所以直接执行并输出此请求。
* 如果有储存的请求，根据前面的论述，电梯里面储存的要么是两条目标楼层一样的捎带请求，要么只有一条稍带请求。如果储存的请求和传入的请求目标楼层不同，那么先执行并输出储存的请求再执行并输出传入的主请求。同时增加停靠开关门的时间。如果储存的请求和传入的请求目标楼层相同，一并执行并输出传入的主请求和储存的捎带请求。最后清空第一个和第二个位置。<resetClock(rq) && (According to the orders act stored requests and the main request) > with < rq.getMt() == 1>

1. act(Request rq)

/\*\*

\* @REQUIRES: rq != null;

\* @MODIFIES: diff; floorNow; clock;

\* @EFFECTS: diff == \old(floorNow) - rq.getFloor(); floorNow == rq.getFloor(); clock == 0.5 \* Math.abs(\old(floorNow) - rq.getFloor()) + \old(clock);

\*/

根据上述过程规格，获得如下的划分：

< diff == \old(floorNow) - rq.getFloor(); floorNow == rq.getFloor() && clock == 0.5 \* Math.abs(\old(floorNow) - rq.getFloor()) + \old(clock) && clock == 0.5 \* Math.abs(diff) + \old(clock)> with < all conditions >

* 此方法依次计算diff，floorNow，clock三个属性，不涉及划分，只要Rq!=null，就可以进行计算，实现满足< diff == \old(floorNow) - rq.getFloor(); floorNow == rq.getFloor() && clock == 0.5 \* Math.abs(\old(floorNow) - rq.getFloor()) + \old(clock) && clock == 0.5 \* Math.abs(diff) + \old(clock)> with < all conditions >

1. toString(Request rq) :

/\*\*

\* @REQUIRES: rq != null;

\* @MODIFIES: None;

\* @EFFECTS: diff == 0 ==> \result == "[" + rq.toString() + "]/" + "(" + floorNow + String.format(",STILL,%.1f)", clock + 1); diff > 0 ==>\result == "[" + rq.toString() + "]/" + "(" + floorNow + String.format(",DOWN,%.1f)", clock); diff < 0 ==> \result == "[" + rq.toString() + "]/" + "(" + floorNow + String.format(",UP,%.1f)", clock);

\*/

根据上述过程规格，获得如下的划分：

< \result == "[" + rq.toString() + "]/" + "(" + floorNow + String.format(",STILL,%.1f)", clock + 1)> with < diff == 0 >

< \result == "[" + rq.toString() + "]/" + "(" + floorNow + String.format(",DOWN,%.1f)", clock)> with < diff > 0 >

< \result == "[" + rq.toString() + "]/" + "(" + floorNow + String.format(",UP,%.1f)", clock)> with < diff < 0 >

* 此方法查询diff的值，如果diff==0，说明是一个停留开关门的运动，返回相应的输出语句。满足< \result == "[" + rq.toString() + "]/" + "(" + floorNow + String.format(",STILL,%.1f)", clock + 1)> with < diff == 0 >
* 如果diff>0，说明是一个下降的运动，返回相应的输出语句。满足< \result == "[" + rq.toString() + "]/" + "(" + floorNow + String.format(",DOWN,%.1f)", clock)> with < diff > 0 >
* 如果diff<0，说明是一个上升的运动，返回相应的输出语句。满足< \result == "[" + rq.toString() + "]/" + "(" + floorNow + String.format(",UP,%.1f)", clock)> with < diff < 0 >

1. repOK():

/\*\*

\* @REQUIRES:None;

\* @MODIFIES:None;

\* @EFFECTS:(floorNow <= 10 && floorNow >= 0 && clock >= 0.0 && diff >= -9 && diff <= 9) ==> \result == true;

\*/

根据上述过程规格，获得如下的划分：

<\result == true> with <(floorNow <= 10 && floorNow >= 0 && clock >= 0.0 && diff >= -9 && diff <= 9) >

<\result == fasle> with <!(floorNow <= 10 && floorNow >= 0 && clock >= 0.0 && diff >= -9 && diff <= 9) >

* 返回(floorNow <= 10 && floorNow >= 0 && clock >= 0.0 && diff >= -9 && diff <= 9)，满足分支<\result == true> with <(floorNow <= 10 && floorNow >= 0 && clock >= 0.0 && diff >= -9 && diff <= 9) >与分支<\result == fasle> with <!(floorNow <= 10 && floorNow >= 0 && clock >= 0.0 && diff >= -9 && diff <= 9) >

综上所述，所有方法的实现都满足规格。从而可以推断，Elevator的实现是正确的，即满足其规格要求。

Request类：

1. 抽象对象得到了有效实现论证

/\*\*

\* @Overview: Request record all the information of a request, including its

\* guest. A FR request's information consist of its floor where it

\* comes from, the move direction, the time it merges, time time to

\* be set as a picked request, whether it is a main request and its

\* merging order. An ER request's information consist of its floor it

\* aiming at, the time it merges, time time to be set as a picked

\* request, whether it is a main request and its merging order. It

\* provides methods to check information.

\*/

private Requester guest; // 代表请求的顾客类型

private int askfloor; // 代表FR的请求来源楼层

private int target; // 代表ER的请求目的地

private Direction direction; // 代表FR的请求方向

private long time; // 代表请求的出现时间

private long ftime; // 代表如果请求是可捎带请求，记录其主请求的time即其主请求的出现时间

private int motion; // 代表其是否为一个主请求

private int order; // 代表其输入的顺序

根据上面的叙述，类的overview明确了抽象对象，且类的rep能够通过抽象函数映射到相应的抽象对象。

对象有效性论证

1. 针对构造方法，论证对象的初始状态满足不变式，即repOK为真。

Request提供了两个构造方法，Request(Requester a, int b, Direction c, long d, int e) 和Request(Requester a, int b, long d, int e)，他们分别初始化不同的的rep。

/\*\*

\* @REQUIRES: a!=null; c!=null; 0 < b < 11; d >= 0; e >= 0;

\* @MODIFIES: this;

\* @EFFECTS: guest == a; askfloor == b; direction == c; time == d; ftime == 0;

\* motion == 1; order == e;

\*/

public Request(Requester a, int b, Direction c, long d, int e) {

guest = a;

askfloor = b;

direction = c;

time = d;

ftime = 0;

motion = 1;

order = e;

}

repOK的运行结果显然返回true。

/\*\*

\* @REQUIRES: a!=null; 0 < b < 11; d >= 0; e >= 0;

\* @MODIFIES: this;

\* @EFFECTS: guest == a; target == b; time == d; ftime == 0; motion == 1; order

\* == e;

\*/

public Request(Requester a, int b, long d, int e) {

guest = a;

target = b;

time = d;

ftime = 0;

motion = 1;

order = e;

}

repOK的运行结果显然返回true。

1. 逐个论证每个对象状态更改方法的执行都不会导致repOK的返回值为false。
   1. Request提供了两个状态更新方法：setMt、setFt、，下面逐个进行论证。
   2. 由于set方法不需要写规格，并且，显然只要合理调用，都不会导致repOK为false
2. 该类的其他几个方法的执行皆不改变对象状态，因此这些方法执行前和执行后的repOK都为true。
3. 综上，对该类任意对象的任意调用都不会改变其repOK为true的特性。因此该类任意对象始终保持对象有效性。
4. 方法实现正确性论证

目标：在方法的前置条件和后置条件没有逻辑矛盾的前提下，根据前置条件和后置条件分析整理出完整的{<effect Y> with <input partition X>}，然后对照方法代码实现来论证给定每个<partition X>，方法执行结果都满足<effect Y>。

1. getFloor():

/\*\*

\* @REQUIRES: a!=null;

\* @MODIFIES: this;

\* @EFFECTS: guest == Requester.FR ==> \result == askfloor;

\* guest == Requester.ER ==> \result == target;

\*/

根据上述过程规格，获得如下的划分：

<\result == askfloor > with < guest == Requester.FR >

<\result == target > with < guest == Requester.ER >

* 方法首先判断guest类型
* 如果是FR类型的请求，其只储存请求的来源楼层，故返回askfloor作为getFloor的结果，处理满足分支<\result == askfloor > with < guest == Requester.FR >
* 如果是ER类型的请求，其只储存请求的目标楼层，故返回target作为getFloor的结果，处理满足分支<\result == target > with < guest == Requester.ER >

1. equals(Object o):

/\*\*

\* @REQUIRES: None;

\* @MODIFIES: None;

\* @EFFECTS: (o instanceof Request)&&(guest == rq.getGt() && ((guest == Requester.FR && askfloor == rq.getAf() && direction == rq.getDr() && rq.getTime() >= time) || (guest == Requester.FR && target == rq.getTg() && rq.getTime() >= time))) ==> \result == true; (other conditions) ==> \result == false;

\*/

根据上述过程规格，获得如下的划分：

<\result == true> with < o == this >

<\result == false> with <!(o instanceof Request)>

<\result == true> with <guest == ((Request) o).getGt() && guest == Requester.FR && askfloor == ((Request) o).getAf() && direction == ((Request) o).getDr() && ((Request) o).getTime() >= time >

<\result == true> with <guest == ((Request) o).getGt() && guest == Requester.ER && target == ((Request) o).getTg() && ((Request) o).getTime() >= time >

<\result == false> with <all the other conditions>

* 首先检查o是否等于该请求本身，即是否是同一块内存的不同引用。如果是，返回true。满足<\result == true> with < o == this >
* 否则的话，检查o本质上是不是一个请求类型的变量，如果不是，显然o与此请求不相等。返回false，此情况下满足分支<\result == false> with <!(o instanceof Request)>
* 此时已经可以知道o是一个请求，检查其乘客类型，如果是FR，分别对比askfloor、direction、time如果askfloor、direction相等，并且((Request) o).getTime() >= time，则返回true。此处理满足<\result == true> with <guest == ((Request) o).getGt() && guest == Requester.FR && askfloor == ((Request) o).getAf() && direction == ((Request) o).getDr() && ((Request) o).getTime() >= time >
* 如果是ER，分别对比target和time，如果target相等，并且((Request) o).getTime() >= time，则返回true。此处理满足<\result == true> with <guest == ((Request) o).getGt() && guest == Requester.ER && target == ((Request) o).getTg() && ((Request) o).getTime() >= time >
* 对于其他情况，均返回的是false。满足<\result == false> with <all the other conditions>

1. toString():

/\*\*

\* @REQUIRES: None;

\* @MODIFIES: None;

\* @EFFECTS: guest == Requester.FR ==> \result == guest + "," + askfloor + "," + direction + "," + String.format("%d", time); guest == Requester.ER ==> \result == guest + "," + target + "," + String.format("%d", time);

\*/

根据上述过程规格，获得如下的划分：

< \result == guest + "," + askfloor + "," + direction + "," + String.format("%d", time) > with < guest == Requester.FR>

< \result == guest + "," + target + "," + String.format("%d", time)> with < guest == Requester.ER >

* 首先提前准备好需要的字符串的需要用到保留功能的部分。
* 判断乘客类型，如果是FR，依次返回乘客类型，请求来源楼层，方向，请求出现的时间。此处理满足< \result == guest + "," + askfloor + "," + direction + "," + String.format("%d", time) > with < guest == Requester.FR>
* 如果是ER，依次返回乘客类型，目标楼层，请求出现的时间。此处理满足< \result == guest + "," + target + "," + String.format("%d", time)> with < guest == Requester.ER >

1. repOK():

/\*\*

\* @REQUIRES:None;

\* @MODIFIES:None;

\* @EFFECTS: guest != null && time >= 0 && ftime >= 0 && (motion == 1 || motion == 0) && order >= 0 && (guest == Requester.FR && askfloor < 11 && askfloor > 0 && direction != null || guest == Requester.ER && target < 11 && target > 0) ==> \result == true;

\*/

根据上述过程规格，获得如下的划分：

< \result == guest != null && time >= 0 && ftime >= 0 && (motion == 1 || motion == 0) && order >= 0 && (guest == Requester.FR && askfloor < 11 && askfloor > 0 && direction != null || guest == Requester.ER && target < 11 && target > 0) > with < all conditions >

* 直接返回guest != null && time >= 0 && ftime >= 0 && (motion == 1 || motion == 0) && order >= 0 && (guest == Requester.FR && askfloor < 11 && askfloor > 0 && direction != null || guest == Requester.ER && target < 11 && target > 0)，满足分支< \result == guest != null && time >= 0 && ftime >= 0 && (motion == 1 || motion == 0) && order >= 0 && (guest == Requester.FR && askfloor < 11 && askfloor > 0 && direction != null || guest == Requester.ER && target < 11 && target > 0) > with < all conditions >

综上所述，所有方法的实现都满足规格。从而可以推断，Request的实现是正确的，即满足其规格要求。

SubSchduler类：

1. 抽象对象得到了有效实现论证

/\*\*

\* @Overview: SubSchduler is a scheduler which get request and send to elevator.

\* It has a position to record last request sent to elevator.

\* And it provides methods to use.

\*/

private Request lastRq; // 储存发送给电梯的上一条请求。

根据上面的叙述，类的overview明确了抽象对象，且类的rep能够通过抽象函数映射到相应的抽象对象。

对象有效性论证

1. 针对构造方法，论证对象的初始状态满足不变式，即repOK为真。

SubScheduler提供了一个构造方法，SubScheduler()，它初始化全部的rep。

/\*\*

\* @REQUIRES: None;

\* @MODIFIES: this;

\* @EFFECTS: lastRq == null;

\*/

public SubScheduler() {

lastRq = null;

}

电梯处于初始状态时，没有上一条请求，所以实际上repOK，永远满足，repOK的运行结果显然返回true。

1. 逐个论证每个对象状态更改方法的执行都不会导致repOK的返回值为false。

SubScheduler提供了两个状态更新方法：command、schedule，下面逐个进行论证。

* 假设command(Queue queue, Elevator elevator)方法开始执行时，repOK为true。

1. 不论里面功能如何，显然不会导致return true发生改变
2. 因此，该方法的执行不会导致repOK为假，不违背表示不变式。

* 假设schedule(Queue queue, Elevator elevator)方法开始执行时，repOK为true。

1. 不论里面功能如何，显然不会导致return true发生改变
2. 因此，该方法的执行不会导致repOK为假，不违背表示不变式。
3. 该类的其他几个方法的执行皆不改变对象状态，因此这些方法执行前和执行后的repOK都为true。
4. 综上，对该类任意对象的任意调用都不会改变其repOK为true的特性。因此该类任意对象始终保持对象有效性。
5. 方法实现正确性论证

目标：在方法的前置条件和后置条件没有逻辑矛盾的前提下，根据前置条件和后置条件分析整理出完整的{<effect Y> with <input partition X>}，然后对照方法代码实现来论证给定每个<partition X>，方法执行结果都满足<effect Y>。

1. command(Queue queue, Elevator elevator):

/\*\*

\* @REQUIRES: queue != null; elevator ！= null;

\* @MODIFIES: None;

\* @EFFECTS: (drive the elevator, get a request for it and wash the queue)

\*/

根据上述过程规格，获得如下的划分：

< schedule a request for elevator, drive the elevator. Wash the queue on the basis of the request and its corresponding main request> with < all the conditions >

* 首先调用方法schedule(Queue queue, Elevator elevator)，获取到下一个应该放入电梯的请求。如果获得的是null，什么也不做。再将获取的请求放入电梯，调用elevator.move(rq)使其做出相应响应。将此请求赋值给lastRq。
* 根据lastRq是否为主请求，获取预测时间，如果是主请求。那么调用elevator.predictTime(queue.getMain(), lastRq)时，queue.getMain()与lastRq将会是同一个请求，说明电梯说明电梯在运行完lastRq后就执行完了主请求。不需要考虑楼层停留。所以进行队列清洗queue.wash(lastRq, elevator.predictTime(queue.getMain(), lastRq));
* 如果是稍带请求。那么调用elevator.predictTime(queue.getMain(), lastRq)时，queue.getMain()与lastRq将会是不一个请求，说明电梯说明电梯在运行完lastRq后会继续上升，根据对于电梯的论证，可以知道，捎带请求放入电梯后虽然有可能值请之前的储存的捎带请求，但是该请求不会立即执行，而是直接储存，所以用来预测的时间需要加上1s来模拟停留。所以进行队列清洗queue.wash(lastRq, elevator.predictTime(queue.getMain(), lastRq) + 1);+1是十分必要的
* 根据以上论证，处理符合分支< schedule a request for elevator, drive the elevator. Wash the queue on the basis of the request and its corresponding main request> with < all the conditions >

1. schedule(Queue queue, Elevator elevator):

/\*\*

\* @REQUIRES: queue != null; elevator ！= null;

\* @MODIFIES: System.out;

\* @EFFECTS: (wash the queue when queue not empty, print the information of homogeneous request and return a proper Request as rq) ==> \result == rq && lastRq == Rq;

\*/

根据上述过程规格，获得如下的划分：

<\result == null && lastRq ==null > with < queue.end()==true >

<get the validation of first element of queue, if it is a homogeneous request print its information, move the head of queue and get next one to try, until get a proper request. And assign it to lastRq and return it > with < queue.end()==false>

* 首先检查queue是否为空，如果为空，直接取出队首元素，即null，赋值给lastRq，并返回。处理满足<\result == null && lastRq ==null > with < queue.end()==true >
* 否则的话，循环遍历整个queue的内容，当队首元素对应的合法性为0的时候，说明是同质元素，输出相关信息。并且移动queue的头指针。
* 如果合法性是2，说明是已经执行过的指令，跳过并且移动queue的头指针。
* 如果合法性是1，由于调用queue.frontRq(lastRq, elevator)的时候会根据情况自动调整队列顺序，所以取出的一定是合理的，故此时跳出循环，取出队首请求元素，赋值给lastRq并且返回。综上，此处理满足分支<get the validation of first element of queue, if it is a homogeneous request print its information, move the head of queue and get next one to try, until get a proper request. And assign it to lastRq and return it > with < queue.end()==false>

1. repOK():

/\*\*

\* @REQUIRES:None;

\* @MODIFIES:None;

\* @EFFECTS:\result == true;

\*/

根据上述过程规格，获得如下的划分：

<\result == true> with < all conditions >

* 返回true，满足分支<\result == true> with < all conditions >

综上所述，所有方法的实现都满足规格。从而可以推断，SubScheduler的实现是正确的，即满足其规格要求。

Main类：

1. 抽象对象得到了有效实现论证

/\*\*

\* @Overview: The class is the start of the whole program. It provides main

\* method.

\*/

根据上面的叙述，类的overview明确了抽象对象，且类的rep能够通过抽象函数映射到相应的抽象对象。

对象有效性论证

1. 针对构造方法，论证对象的初始状态满足不变式，即repOK为真。

Main类不提供构造方法，并且repOK永远满足，repOK的运行结果显然返回true。

1. 逐个论证每个对象状态更改方法的执行都不会导致repOK的返回值为false。

Main不提供状态更新方法。

1. 综上，对该类任意对象的任意调用都不会改变其repOK为true的特性。因此该类任意对象始终保持对象有效性。
2. 方法实现正确性论证

目标：在方法的前置条件和后置条件没有逻辑矛盾的前提下，根据前置条件和后置条件分析整理出完整的{<effect Y> with <input partition X>}，然后对照方法代码实现来论证给定每个<partition X>，方法执行结果都满足<effect Y>。

1. main(String args[]):

/\*\*

\* @MODIFIES:None;

\* @EFFECTS: The start of the whole elevator system, it states a SubScheduler, a queue of request, a elevator and a Scanner, then use them to get requests and execute them;

\*/

根据上述过程规格，获得如下的划分：

< Start the whole elevator system, states a SubScheduler, a queue of request, a elevator and a Scanner, then use them to get requests and execute them, catching Throwables at the same time.> with < all the conditions >

* 首先依次进行声明所需要的SubScheduler、queue of request、elevator、Scanner变量，读取迭代器逐行读取Scanner读取System.in的输入，并记录读入的行数。并进行分析，当读到“RUN”或者去除行数超过100行的时候结束读入。并且调度请求供电梯执行，直到，queue内没有请求。显然，赐敕筛查满足分支< Start the whole elevator system, states a SubScheduler, a queue of request, a elevator and a Scanner, then use them to get requests and execute them, catching Throwables at the same time.> with < all the conditions >

1. repOK():

/\*\*

\* @REQUIRES:None;

\* @MODIFIES:None;

\* @EFFECTS:\result == true;

\*/

根据上述过程规格，获得如下的划分：

<\result == true> with < all conditions >

* 返回true，满足分支<\result == true> with < all conditions >

综上所述，所有方法的实现都满足规格。从而可以推断，Main的实现是正确的，即满足其规格要求。

Queue类：

1. 抽象对象得到了有效实现论证

/\*\*

\* @Overview: Queue is a queue of request, it provides methods to add, wash,

\* judge, justify, change information and get information. It

\* consists of a list of request with a list of corresponding

\* validity, the front and rear of the list, a request able to be

\* upgrade with its position in the list and a main request. And it

\* Defines a constant to restrict the length of the list.

\*/

private static final int MAX = 200; // 所设常数

private Request[] rqList; // 请求列表

private int[] validity; // 请求得合法性表

private int front; // 队列头

private int rear; // 队列尾

private Request unhandle; // 待升级请求

private Request mainRq; // 当前主指令

private int unhandlePosition; // 待升级请求的位置

根据上面的叙述，类的overview明确了抽象对象，且类的rep能够通过抽象函数映射到相应的抽象对象。

对象有效性论证

1. 针对构造方法，论证对象的初始状态满足不变式，即repOK为真。

Queue提供了一个构造方法，Queue()，其初始化了不同的的rep。

/\*\*

\* @REQUIRES: None;

\* @MODIFIES: this;

\* @EFFECTS: rqList!=null; validity!=null; front == 0; rear == 0; unhandle ==

\* null; mainRq == null; unhandlePosition == -1;

\*/

public Queue() {

rqList = new Request[MAX];

validity = new int[MAX];

front = 0;

rear = 0;

unhandle = null;

mainRq = null;

unhandlePosition = -1;

}

repOK的运行结果显然返回true。

1. 逐个论证每个对象状态更改方法的执行都不会导致repOK的返回值为false。

Queue提供了五个状态更新方法：frontRq、 justifyUnhandle、 moveFront、 wash、 parse、parseRq，下面逐个进行论证。

* 假设justifyUnhandle()方法开始执行时，repOK为true。

1. 首先temp = validity[unhandlePosition]记录待升级请求。不会违背表示不变式。
2. 然后从front起到unhandlePosition，把每一个请求后移一个位置。此时，front位置空出。将其赋值为Unhandle，合法性列表同样操作。只改变列表中元素的位置，显然不会违背表示不变式。
3. 因此，该方法的执行不会导致repOK为假，不违背不变式。

* 假设moveFront(int n)方法开始执行时，repOK为true。

1. front加上n，由于n>0，故front只会变大，不会影响不变式满足性。
2. 因此，该方法的执行不会导致repOK为假，不违背不变式。

* 假设wash(Request lastRq, double clock)方法开始执行时，repOK为true。

1. wash函数循环查看请求队列的请求，知道出现时间大于等于clock或者队列结束，检查每一个合法性为1得请求是否和传入的lastRq同质，其中判断同质调用equals函数。如果同质，将其对应位置的合法性置为0，代表其为一个同质请求。这个过程中，只会改变合法性队列中得数字大小，不违背不变式。
2. 因此，该方法的执行不会导致repOK为假，不违背不变式。

* 假设parse(String str)方法开始执行时，repOK为true。

1. parse执行parseRq方法，如果返回值不为0，输出非法请求。依照parseRq得正确性，不会使repOK为假，不违背不变式。
2. 因此，该方法的执行不会导致repOK为假，不违背不变式。

* 假设parseRq(String str)方法开始执行时，repOK为true。

1. 先用正则表达式判断基本格式是否满足，如果不满足直接返回非0常数，使用正则表达式进行分割，转换为相应的内容，一次判断其合法性，如果是不合法的输入，依次返回不同的非0常数。当经过所有的检测之后，使用Request的构造方法构造一个Request实例，对于第一条指令，判断是否为指定指令。至此，不曾修改Queue中的变量。因此不会导致repOK为假，不违背不变式。
2. 将指令加入rqList，并将相应的位置的合法性置为1。同时队尾增长。此时不会导致repOK为假，不违背不变式。
3. 因此，该方法的执行不会导致repOK为假，不违背不变式。

* 假设frontRq(Request lastRq, Elevator elevator)方法开始执行时，repOK为true。

1. 先判断队首元素的合法性如果为0，直接返回该元素。不会导致repOK为假，不违背不变式。
2. 循环依据条件进行遍历，先用judge(mainRq, rqList[i], elevator)判断指令的类型，依据类型进行不同的操作，找到最合适的指令，期间调用justifyUnhandle()操作，根据上面论述，不会改变repOK结果。同时修改unhandle， unhandlePosition，对Unhandle进行覆盖或清空操作。并标记其位置到unhandlePosition。由于产生的位置均>=-1，不会改变repOK。以及对于队列内的一些请求进行了主请求或者捎带请求的状态修改。但这不会导致repOK的状态。不违背不变式。
3. 因此，该方法的执行不会导致repOK为假，不违背不变式。
4. 该类的其他几个方法的执行皆不改变对象状态，因此这些方法执行前和执行后的repOK都为true。
5. 综上，对该类任意对象的任意调用都不会改变其repOK为true的特性。因此该类任意对象始终保持对象有效性。
6. 方法实现正确性论证

目标：在方法的前置条件和后置条件没有逻辑矛盾的前提下，根据前置条件和后置条件分析整理出完整的{<effect Y> with <input partition X>}，然后对照方法代码实现来论证给定每个<partition X>，方法执行结果都满足<effect Y>。

1. end():

/\*\*

/\*\*

\* @EFFECTS: (front >= rear) ==> (\result == true); (front < rear) ==> (\result == false);

\*/

根据上述过程规格，获得如下的划分：

<\result == true > with < front >= rear >

<\result == false > with < front < rear >

* 判断front和rear的关系，进行响应的返回，显然满足规格。

1. frontRq(Request lastRq, Elevator elevator):

/\*\*

\* @REQUIRES: None;

\* @MODIFIES: this;

\* @EFFECTS: (According to the information from lastRq and elevator, choose a proper request on the basic of the information of this queue. And change some fields if necessary) ==> \result == request;

\*/

根据上述过程规格，获得如下的划分：

<\result == rqList[front] > with < validity[front] == 0 >

<According to the information from lastRq and elevator, choose a proper request on the basic of the information of this queue. And change some fields if necessary > with < validity[front] != 0>

* 先判断队首元素的合法性如果为0，直接返回该元素，满足<\result == rqList[front] > with < validity[front] == 0 >
* 否则的话，循环进行遍历，如果与主请求同质或者rqList[i].getTime() <= newClock + 1，均证明是同质，给其合法性赋值为0，跳过此次循环。如果rqList[i].getTime() >= newClock。跳出循环。如果合法性为0同质或2已执行，则跳过。此时再用judge(mainRq, rqList[i], elevator)判断指令的类型，依据类型进行不同的操作，找到最合适的指令。
* 如果judge结果为1，若用来储存最有请求的临时变量nearRequest未进行赋值（nearPosition == -1），在满足稍带条件时，将此临时变量赋值到储存最有请求的临时变量，同时对其位置记录。如果nearRequest不为空，在可稍待的条件下，进行对比，如果此请求与电梯当前楼层比nearRequest更接近，赋值此临时变量赋值到储存最有请求的临时变量，同时对其位置记录。
* 如果judge结果为2，代表其为可升级指令，判断待升级位置是否为空，如果为空，将其放入，同时对其位置记录。
* 如果judge结果为0，跳过。
* 此时判断循环完毕后nearRequest是否获取到，如果没有说明队说元素已执行元素，向后取知道碰到非同质。然后调用justifyUnhandle()，将待升级升级为主请求，并调整队列的位置关系。返回此时的主请求，即刚刚升级的待升级请求。
* 否则说明取到，并且为捎带请求，置请求的motion为0，代表为捎带请求。将请求的合法性置为2，代表已执行，返回之。综上叙述，满足<According to the information from lastRq and elevator, choose a proper request on the basic of the information of this queue. And change some fields if necessary > with < validity[front] != 0>

1. getRelativeDirection(int a, int b):

/\*\*

\* @MODIFIES: None;

\* @EFFECTS: a > b ==> \result == Direction.DOWN; a < b ==> \result ==

\* Direction.UP; a == b ==> \result == null;

\*/

根据上述过程规格，获得如下的划分：

<\result == Direction.DOWN > with < a > b >

<\result == Direction.UP > with < a < b >

<\result == null > with < a == b >

* 将楼层的大小关系转化为方向。如果a > b返回Direction.DOWN。满足分支<\result == Direction.DOWN > with < a > b >
* 如果a < b返回Direction.UP。满足分支<\result == Direction.UP > with < a < b >
* 如果a == b返回null。满足分支<\result == null > with < a == b >

1. judge(Request mainRq, Request rq, Elevator elevator):

/\*\*

\* @REQUIRES: None;

\* @MODIFIES: this;

\* @EFFECTS: (rq is pickup-able for mainRq and elevator but not scalable) == \result == 1; (rq is scalable for mainRq and elevator) == \result == 2; (other conditions) ==> \result == 0;

\*/

根据上述过程规格，获得如下的划分：

<judge FR, return its type, such as 0, 1> with < rq.getGt() == Requester.FR >

< judge FR, return its type, such as 0, 1, 2> with < rq.getGt() == Requester.ER >

* 首先计算估计运行完主请求的时间newClock。然后根据楼层关系计算direction。判断请求类型。
* 如果是FR，当请求的位置位于elevator当前位置以及主请求目标楼层间，并且时间不会冲突的时候，返回1，代表可捎带。否则返回0，代表不可捎带。处理满足<judge FR, return its type, such as 0, 1> with < rq.getGt() == Requester.FR >
* 如果是ER，当请求的位置位于elevator当前位置以及主请求目标楼层间，并且时间不会冲突的时候，返回1，代表可捎带。如果时间合理请求的位置位于elevator当前位置以及主请求目标楼层延长线上，并且与主请求（此时一定FR）方向一致，返回2，代表可升级。否则返回0，代表不可捎带。满足分支<\result == null > with < a == b >

1. justifyUnhandle():

/\*\*

\* @REQUIRES: None;

\* @MODIFIES: this;

\* @EFFECTS: (\all int i; unhandlePosition >= i > front; validity[i] == old(validity[i - 1])&&rqList[i] == \old(rqList[i - 1])); validity[front] == \old(validity[unhandlePosition]); rqList[front] == unhandle;

\*/

根据上述过程规格，获得如下的划分：

< justify the position between unhandlePosition and front and move unhandled front> with < all conditions>

* 首先记录unhandlePosition位置上的请求的合法性为temp。从front到unhandlePosition将请求一次向后移动一个位置。front位置上的请求赋值为unHandle，其合法性记录为temp。综上处理满足< justify the position between unhandlePosition and front and move unhandled front> with < all conditions>

1. moveFront(int n):

/\*\*

\* @MODIFIES: front;n>0;

\* @EFFECTS: front == \old(front) + n;

\*/

根据上述过程规格，获得如下的划分：

< front == \old(front) + n > with < all conditions>

* 直接相加赋值，满足分支< front == \old(front) + n > with < all conditions>

1. wash(Request lastRq, double clock):

/\*\*

\* @REQUIRES: clock>=0;

\* @MODIFIES: validity;

\* @EFFECTS: (\all int i; rear > i >= front && rqList[i].getTime() <= clock; (validity[i] == 1 && lastRq != null && lastRq.equals(rqList[i])) ==> validity[i] == 0;);

\*/

根据上述过程规格，获得如下的划分：

< do nothing > with < lastRq == null>

< validity[i] = 0 > with < lastRq != null && exists request in rqList , (satisfied the conditions) && lastRq.equals(rqList[i])>

* 如果lastRq为空，不做处理，满足< do nothing > with < lastRq == null>
* 否则，循环访问，i < rear && rqList[i].getTime() <= clock时停止访问，如果合法性为1，赋值其合法性为0，满足分支< validity[i] = 0 > with < lastRq != null && exists request in rqList , (satisfied the conditions) && lastRq.equals(rqList[i])>

1. parse(String str):

/\*\*

\* @MODIFIES: System.out; this;

\* @EFFECTS: (parse the input str and according to the parsing result print relative text);

\*/

根据上述过程规格，获得如下的划分：

< do nothing > with < parseRq(str) == 0>

< print relative information > with < parseRq(str) != 0>

* 执行parseRq(str)，如果返回0，退出，满足< do nothing > with < parseRq(str) == 0>
* 否则，print出invalid信息。满足< print relative information > with < parseRq(str) != 0>

1. parseRq(String str):

/\*\*

\* @EFFECTS: (str is a valid request) ==> (rear == \old(rear) + 1) && validity[rear] == 1 && \result == 0; (str is not a valid request) ==> \result != 0;

\*/

根据上述过程规格，获得如下的划分：

< rear == \old(rear) + 1 && validity[rear] == 1 && \result == 0> with < str is a valid request >

< \result != 0 > with < str is not a valid request >

< \result == 1 > with < caught an exception >

* 调用try catch块，如果catch返回1，满足< \result == 1 > with < caught an exception >
* 先用正则表达式判断基本格式是否满足，如果不满足直接返回非0常数，使用正则表达式进行分割，转换为相应的内容，一次判断其合法性，如果是不合法的输入，依次返回不同的非0常数。当经过所有的检测之后，使用Request的构造方法构造一个Request实例，对于第一条指令，判断是否为指定指令。如果不是合法指令，返回零，满足分支< \result != 0 > with < str is not a valid request >
* 否则将指令加入rqList，并将相应的位置的合法性置为1。同时队尾增长。此时满足分支< rear == \old(rear) + 1 && validity[rear] == 1 && \result == 0> with < str is a valid request >

1. repOK():

/\*\*

\* @REQUIRES:None;

\* @MODIFIES:None;

\* @EFFECTS:(rqList != null && validity != null && front >= 0 && rear >= 0 && unhandlePosition >= -1) ==> \result == true;

\*/

根据上述过程规格，获得如下的划分：

< \result == (rqList != null && validity != null && front >= 0 && rear >= 0 && unhandlePosition >= -1) > with < all conditions >

* 直接返回(rqList != null && validity != null && front >= 0 && rear >= 0 && unhandlePosition >= -1)，满足分支< \result == (rqList != null && validity != null && front >= 0 && rear >= 0 f&& unhandlePosition >= -1) > with < all conditions >

综上所述，所有方法的实现都满足规格。从而可以推断，Queue的实现是正确的，即满足其规格要求。