类正确性论证报告

1、Main 类

1、抽象对象有效性论证

根据 Overview，Main 类的表示对象为 RQ，input，通过抽象函数映射为能处理输入 input，能加入请求队列 RQ 的主类。

2、对象有效性论证

（a）对象的初始状态满足不变式，即 repOK 为 true。

（b）逐个论证对每个对象状态更改方法的执行都不会导致 repOK 返回值为 false。

Main 类提供了两个状态更新的方法:InputSovling、main,下面逐个进行论证。

⇨ 假设 InputSolving()方法开始执行时，repOK 为 true。

1）InputSolving 方法用 i 进行计数，每输入一行并进行处理后，i 数目加 1，newTimeMax 改为当前请求发出时间，i 达到 100 后或者一行的输入为“RUN”时跳出输入处理循环。输入的过程是 input 改变的过程，但是改变的 过程不会导致 input==null，所以 repOK 始终为 true。至于对于输入的处理，下面将进行论证。

2）对于每一行输入，都存入 getsIni 中并对其进行处理。将 input.nextLine() 存入 getsIni 中并消除空格存入 gets 中，显然不改变 repOK 的取值。

3）当 gets 匹配满足 floorUP 时，当 gets 的请求发出时间 T 大于最大边界情况时，报错并结束单次循环，不会导致 repOK 为 false。当 T<newTimeMax 或者第一个请求不是（FR,1,UP,0）时，报错并结束单次循环，不会导致 repOK 为 false。将 Requ（m,1,T）加入到 RQ，RQ 增加了请求，不会导致 RQ==null，不会导致 repOK 为 false。

4）当 gets 匹配 floorDOWN 和 elevReq 时同理不会导致 repOK 为 false。

5）当 gets 匹配 RUN 时，循环体结束，不会更新对象状态，不会导致 repOK 为 false。

6）当 gets 匹配其他无效输入时，报错并结束单次循环，不会导致 repOK 为

false。

7）因此，该方法的执行不会导致 repOK 为 false，不违背表示不变式。

⇨ 假设 main 方法开始执行时，repOK 为 true。

1）由于 InputSolving 方法不会导致 repOK 为 false，main 的 InputSolving 方法执行后不会导致 InputSolving 为 false。

2）由于 New_sch 类实现正确，new_sch 方法执行不会导致 repOK 为 false，因此 sch.Schedule()方法的执行不会导致 repOK 为 false。

3）因此，该方法的执行不会导致 repOK 为 false，不违背表示不变式。

（c）综上，对该类的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。

3、方法实现正确性论证

（a）
```
public void InputSolving()
        /**@REQUIRES: None;
        @MODIFIES: input,RQ,System.out;
        @EFFECTS:
        *       (\all int i;i<100) ==>
(gets==floorUP)&&((gets.T<=border&&gets.T>=newTimeMax&&((first_one==gets&&i
==0)||i!=0))&&(i==(\old)i+1&&RQ.size==(\old)RQ.size+1&&RQ.contains(Requ(m,1
,T)))||(gets.T>border||gets.T<newTimeMax||(first_one!=gets&&i==0))&&(System
.out.println("INVALID["+gets+"]")))||
        *
(gets==floorDOWN)&&((gets.T<=border&&gets.T>=newTimeMax&&i!=0)&&(i==(\old)i
+1&&RQ.size==(\old)RQ.size+1&&RQ.contains(Requ(m,-
1,T)))||(gets.T>border||gets.T<newTimeMax||i==0)&&(System.out.println("INVA
LID["+gets+"]")))||
        *
(gets==elevReq)&&((gets.T<=border&&gets.T>=newTimeMax&&i!=0)&&(i==(\old)i+1
&&RQ.size==(\old)RQ.size+1&&RQ.contains(Requ(m,0,T)))||(gets.T>border||gets
.T<newTimeMax||i==0)&&(System.out.println("INVALID["+gets+"]")))||
        *                              (gets==END)&&(\result)||
        *
System.out.println("INVALID["+gets+"]");
        */
```

根据上述过程规格，获得如下划分：

<RQ will add Requ(m,1,T), i will add 1 to the original> with <i<100 and gets is

matched with floorUP and the request gets is not out of range>

<print “INVALID["+gets+"]"” and continue> with <i<100 and gets is matched with floorUP and the request gets is out of range>

<RQ will add Requ(m,-1,T), i will add 1 to the original> with <i<100 and gets is matched with floorDOWN and the request gets is not out of range>

<print “INVALID["+gets+"]"” and continue> with <i<100 and gets is matched with floorDOWN and the request gets is out of range>

<RQ will add Requ(m,0,T), i will add 1 to the original> with <i<100 and gets is matched with elevReq and the request gets is not out of range>

<print “INVALID["+gets+"]"” and continue> with <i<100 and gets is matched with elevReq and the request gets is out of range>

<get out of the loop and the method ends> with <i<100 and gets is matched with END>

√ 在循环体中，当 i<100 且输入的请求 gets 匹配满足 floorUP，且请求发出时间等满足要求时，将该请求 Requ（m,1,T）加入请求队列 RQ 中，且 i++。在此过程中只修改了 i 的值和 RQ，未对其他对象进行修改。因此满足<RQ will add Requ(m,1,T), i will add 1 to the original> with <i<100 and gets is matched with floorUP and the request gets is not out of range>

√ 在循环体中，当 i<100 且输入的请求 gets 匹配满足 floorUP，但请求发出时间等不满足要求时，输出报错信息且结束单次循环（即 continue）。在此过程中未对其他对象进行修改。因此满足<print “INVALID["+gets+"]"” and continue> with <i<100 and gets is matched with floorUP and the request gets is out of range>

√ 在循环体中，当 i<100 且输入的请求 gets 匹配满足 floorDOWN，且请求发出时间等满足要求时，将该请求 Requ（m,-1,T）加入请求队列 RQ 中，且 i++。在此过程中只修改了 i 的值和 RQ，未对其他对象进行修改。因此满足<RQ will add Requ(m,-1,T), i will add 1 to the original> with <i<100 and gets is matched with floorDOWN and the request gets is not out of range>

√ 在循环体中，当 i<100 且输入的请求 gets 匹配满足 floorDOWN，但请求发出时间等不满足要求时，输出报错信息且结束单次循环（即 continue）。在此过程中

未对其他对象进行修改。因此满足<print “INVALID["+gets+"]”” and continue> with <i<100 and gets is matched with floorDOWN and the request gets is out of range>

√ 在循环体中，当 i<100 且输入的请求 gets 匹配满足 elevReq，且请求发出时间等满足要求时，将该请求 Requ（m,0,T）加入请求队列 RQ 中，且 i++。在此过程中只修改了 i 的值和 RQ，未对其他对象进行修改。因此满足<RQ will add Requ(m,0,T), i will add 1 to the original> with <i<100 and gets is matched with elevReq and the request gets is not out of range>

√ 在循环体中，当 i<100 且输入的请求 gets 匹配满足 elevReq，但请求发出时间等不满足要求时，输出报错信息且结束单次循环（即 continue）。在此过程中未对其他对象进行修改。因此满足<print “INVALID["+gets+"]”” and continue> with <i<100 and gets is matched with elevReq and the request gets is out of range>

√ 在循环体中，当 i<100 且输入请求匹配满足 END 时，跳出循环体且方法结束。在此过程中未对其他对象进行修改。因此满足<get out of the loop and the method ends> with <i<100 and gets is matched with END>

（b）
```
public static void main(String[] args) {
        /**@REQUIRES: None;
        @MODIFIES: input,RQ,System.out;
        @EFFECTS: 处理输入和调度器调度;
        */
        Main main = new Main();
        main.InputSolving();
        New_sch sch = new New_sch();
        sch.Schedule();
    }
```
易知，main的正确性直接与Main的InputSolving()方法和New_sch的Schdule()方法，由于可论证Main的InputSolving()方法和New_sch的Schdule()方法的实现是正确的，因此main方法的实现是正确的。

（c）
```
public boolean repOK()
            /** @REQUIRES: None;
            @MODIFIES: None;
            @EFFECTS: \result == invariant(this);
            @ */
```

根据上述规格，获得如下划分：

<return false> with <RQ==null or input ==null>

<return true> with <RQ!=null && RQ!=null>

√若 RQ==null 或者 input==null，返回 false，满足<return false> with <RQ==null or input ==null>

√若 RQ!=null 且 input!=null，返回 true，满足<return true> with <RQ!=null && RQ !=null>

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

2、Ele 类

1、抽象对象有效性论证

根据 Overview，Ele 类通过抽象函数映射为一个可以计算运行时间，开关门时间的电梯。

2、对象有效性论证

该类没有属性，repOK 恒为 true，因此该类任意对象始终保持对象有效性。

3、方法实现正确性论证

（a）
```
public double run(int nowfl,int aimfl,double nowT)
    /** @REQUIRES: (nowfl>=1&&nowfl<=10);
     * (aimfl>=1&&aimfl<=10);
     * (nowT>=0);
    @MODIFIES: None;
    @EFFECTS: \result == abs(nowfl-aimfl)*0.5 + nowT;
    */
```

根据上述规格，获得如下划分：

<throw new Exception> with <nowfl<1 or nowfl>10 or aimfl <1 or aimfl>10 or nowT<0>

<result is abs(nowfl-aimfl)*0.5 + nowT and return the result> with < 10>=nowfl>=1, 10>=aimfl>=1 and nowT>=0 >

√ 方法首先检查输入 nowfl,aimfl,nowT 的状态，如果不满足要求(即 nowfl<1 or nowfl>10 or aimfl <1 or aimfl>10 or nowT<0)，则 throw new Excption，满足<throw new Exception> with <nowfl<1 or nowfl>10 or aimfl <1 or aimfl>10 or nowT<0>

√满足要求 10>=nowfl>=1, 10>=aimfl>=1 and nowT>=0，返回值为 abs(nowfl-aimfl)*0.5 + nowT，除了 finT 未对其他对象进行修改，满足<result is abs(nowfl-aimfl)*0.5 + nowT and return the result > with < 10>=nowfl>=1, 10>=aimfl>=1 and nowT>=0 >

（b）
```
public double OpenDo(double nowT)
    /** @REQUIRES: (nowT>=0);
    @MODIFIES: None;
    @EFFECTS: \result == nowT + 1;
    */
```

根据上述规格，获得如下划分：

<throw new Exception> with <nowT < 0 >

<result is nowT + 1 and return the result> with <nowT >= 0>

√ 方法首先检查输入 nowT 的状态，如果不满足要求(即 nowT < 0)，则 throw new Excption，满足<throw new Exception> with <nowT < 0 >

√满足要求 nowT >= 0，返回值为 nowT + 1，未对任何对象进行修改，满足<result is nowT + 1 and return the result> with <nowT >= 0>

（c）
```
public boolean repOK()
    /** @REQUIRES: None;
    @MODIFIES: None;
    @EFFECTS: \result == true;
    */
```

Ele 类没有属性， repOK 恒为 true，方法实现正确。

综上所述，所有方法都满足规格。从而可以判断 Ele 的实现是正确的，即满足其规格要求。

3、New_sch 类

1、抽象对象有效性论证

根据 Overview，New_sch 类的表示对象为 RQ，通过抽象函数映射为对请求队列 RQ 进行调度的调度器。

2、对象有效性论证

（a）对象初始状态满足不变式，即 repOK 为真。

（b）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。New_sch 提供了三个状态更新方法：Schedule，dele_same，setRQ 下面逐个进行论证。

⇨ 假设 dele_same 方法开始执行时，repOK 为 true。

1）该方法从 j=i+1 开始循环加 1 计数知道 j<RQ.size，判断请求队列 RQ 中第 j 个请求是否满足与第 i 个请求同质的条件，若满足，则将 RQ 的第 j 个请求记为无效请求，输出同质请求提示信息，从 RQ 中删除该请求，且 j--。循环体的执行只会在遇到与第 i 个 同质的请求时从 RQ 中删除该请求，并不会导致 RQ==null，因此不会导致 repOK 为 false。

2）因此，该方法的执行不会导致 repOK 为 false，不违背表示不变式。

⇨ 假设 Schedule 方法开始执行时，repOK 为 true。

1）初始化当前楼层和目标楼层都为 1，显然不改变 repOK 的值。当 RQ.size>0 时，进入循环体，直到 RQ.size==0 循环体结束。循环体结束后，方法结束，repOK 依然为 true。

2）下面论证循环体内部的对象有效性。首先在 RQ 队列里寻找是否有 nowvalid 为 2 的请求，若有，将其设为主请求，若无，将 RQ 队列的第一个请求设为主请求。这一过程可能会更改 RQ，但是并不会造成 RQ==null，repOK 依然为 true。更新 T_now，pre_stage,aim_stage 的值，判断电梯主请求方向，若 aim_stage>pre_stage，则主请求方向为 UP，在 i=pre_stage+1 到 aim_stage 之间，每到一层后，在 RQ 里从 j=0 到 j<RQ.size 循环判断有没有可以捎带的请求，若有，则输出该请求执行信息，删除该请求的同质请求，最后从请求队列里删除该请求，j--，若有可捎带的请求，执行

ev.OpenDo 方法完成开关门时间计算，该方法的有效性可论证，不会导致 repOK 为 false，在执行完包括主请求的可捎带请求后，从 i=0 到 i<RQ.size 循环判断是否有在原主请求方向上的还未执行的可捎带请求，将第一个出现的此类请求的 nowvalid 设为 2。在这个过程中，之前已论证删除同质请求（即执行 dele_same 方法）不会导致 repOK 为 false，删除被捎带的请求只是在 RQ 里删除该请求，不会导致 RQ==null，不会导致 repOK 为 false，设置 RQ 队列里的请求的有效性 nowvalid 不会导致 RQ==null，不会导致 repOK 为 false。若 aim_stage<pre_stage，则主请求方向为 DOWN，有效性论证与 UP 类似，此处不再赘述。若 aim_stage==pre_stage，则主请求为 STILL，主请求执行完开关门动作，输出执行信息，删除同质请求，从 RQ 队列中删除该请求，此过程中 dele_same 方法的执行不会导致 repOK 为 false（已论证），从 RQ 队列中删除主请求也不会导致 RQ==null，因此不会导致 repOK 为 false。

3）因此，该方法的执行不会导致 repOK 为 false，不违背表示不变式。

⇨ 假设 setRQ 方法开始执行时，repOK 为 true。

1）rQ！null，则 RQ=rQ！=null，因此不会导致 reqOK 为 false。

2）因此，方法的执行不会导致 repOK 为 false，不违背表示不变式。

（c）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。

（d）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。

3、方法实现正确性论证

（a）
```
public void dele_same(int i,double T_now,int aim_stage)
    /** @REQUIRES: (i>=0&&i<RQ.queue.length);
     *            (T_now >= 0);
     *            (1<= aim_stage <= 10);
    @MODIFIES: RQ,System.out;
    @EFFECTS:
    *      (\all int j=i+1;j<RQ.size) ==> ((RQ(j).aimstage==aim_stage
&& RQ(j).T<=T_now && RQ(j).Reqnum==RQ(i).Reqnum) && (RQ(j).valid==0) &&
(RQ(j).Reqnum==0 && print ""#SAME[ER,%d,%.0f]%n",RQ(j).aimstage ,RQ(j).T")
&& (RQ.size == (\old)RQ.size - 1 && j == (\old)j - 1) ||
    *                                  (RQ(j).aimstage==aim_stage &&
```

```
RQ(j).T<=T_now && RQ(j).Reqnum==RQ(i).Reqnum) && (RQ(j).valid==0) &&
(RQ(j).Reqnum==0 && print
""#SAME[FR,%d,UP,%.0f]%n",RQ(j).aimstage,RQ(j).T") && (RQ.size ==
(\old)RQ.size - 1 && j == (\old)j - 1) ||
       *                                    (RQ(j).aimstage==aim_stage &&
RQ(j).T<=T_now && RQ(j).Reqnum==RQ(i).Reqnum) && (RQ(j).valid==0) &&
(RQ(j).Reqnum==0 && print
""#SAME[FR,%d,DOWN,%.0f]%n",RQ(j).aimstage,RQ(j).T") && (RQ.size ==
(\old)RQ.size - 1 && j == (\old)j - 1)) &&
       *                                   j == (\old)j + 1;
       */
```

根据上述规格，获得如下划分：

<do nothing> with <i<0 or i>=RQ.size or T_now<0 or aim_stage<1 or aim_stage >10>

<RQ.valid(j) be set to 0,print“"#SAME[ER,%d,%.0f]%n",RQ(j).aimstage,RQ(j).T” , delete RQ(j) from RQ and j is changed to j -1> with <j<RQ.size, RQ(j).aimstage ==aim_stage , RQ(j).T<=T_now , RQ(j).Reqnum==RQ(i).Reqnum and RQ(j).Reqnum == 0>

<RQ.valid(j) be set to 0,print“"#SAME[FR,%d,UP,%.0f]%n",RQ(j).aimstage, RQ(j).T” , delete RQ(j) from RQ and j is changed to j -1> with <j<RQ.size, RQ(j).aimstage ==aim_stage , RQ(j).T<=T_now , RQ(j).Reqnum==RQ(i).Reqnum and RQ(j).Reqnum == 1>

<RQ.valid(j) be set to 0,print“"#SAME[FR,%d,DOWN,%.0f]%n", RQ(j).aimstage,RQ(j).T” , delete RQ(j) from RQ and j is changed to j -1> with <j<RQ.size, RQ(j).aimstage ==aim_stage , RQ(j).T<=T_now , RQ(j).Reqnum==RQ(i).Reqnum and RQ(j).Reqnum == -1>

√ 方法首先检查确认输入 i, T_now, aim_stage 的状态，如果不满足 REQUIRES 里的条件则直接返回结束，满足<do nothing> with <i<0 or i>=RQ.size or T_now<0 or aim_stage<1 or aim_stage >10>

√ 在循环体中，当 j<RQ.size 时，若 RQ(j).aimstage = aim_stage 且 RQ(j).T<=nowT 且 RQ(j).Reqnum = RQ(i).Reqnum =0 时，将 RQ(j)的 valid 设为 0，输出同质信息，从 RQ 队列中删除该请求，j = j -1 。在此过程中只修改了 j 和 this 所管理的 RQ，未对 i，T_now，aim_stage 和其他对象进行修改。因此，满足<RQ.valid(j) be set

to 0,print“"#SAME[ER,%d,%.0f]%n",RQ(j).aimstage,RQ(j).T” , delete RQ(j) from RQ and j is changed to j -1> with <j<RQ.size, RQ(j).aimstage ==aim_stage , RQ(j).T<=T_now , RQ(j).Reqnum==RQ(i).Reqnum and RQ(j).Reqnum == 0>

√ 在循环体中，当 j<RQ.size 时，若 RQ(j).aimstage = aim_stage 且 RQ(j).T<=nowT 且 RQ(j).Reqnum = RQ(i).Reqnum =1 时，将 RQ(j)的 valid 设为 0，输出同质信息，从 RQ 队列中删除该请求，j = j -1 。在此过程中只修改了 j 和 this 所管理的 RQ，未对 i，T_now，aim_stage 和其他对象进行修改。因此，满足<RQ.valid(j) be set to 0,print“"#SAME[FR,%d,UP,%.0f]%n",RQ(j).aimstage, RQ(j).T” , delete RQ(j) from RQ and j is changed to j -1> with <j<RQ.size, RQ(j).aimstage ==aim_stage , RQ(j).T<=T_now , RQ(j).Reqnum==RQ(i).Reqnum and RQ(j).Reqnum == 1>

√ 在循环体中，当 j<RQ.size 时，若 RQ(j).aimstage = aim_stage 且 RQ(j).T<=nowT 且 RQ(j).Reqnum = RQ(i).Reqnum =-1 时，将 RQ(j)的 valid 设为 0，输出同质信息，从 RQ 队列中删除该请求，j = j -1 。在此过程中只修改了 j 和 this 所管理的 RQ，未对 i，T_now，aim_stage 和其他对象进行修改。因此，满足<RQ.valid(j) be set to 0,print“"#SAME[FR,%d,DOWN,%.0f]%n", RQ(j).aimstage,RQ(j).T” , delete RQ(j) from RQ and j is changed to j -1> with <j<RQ.size, RQ(j).aimstage ==aim_stage , RQ(j).T<=T_now , RQ(j).Reqnum==RQ(i).Reqnum and RQ(j).Reqnum == -1>

（b）
```
public void Schedule()
       /** @REQUIRES: None;
       @MODIFIES: RQ,System.out;
       @EFFECTS:
       *      (RQ.size != 0) ==> (main_req.nowvalid == 2 && pre_stage ==
(\old)aim_stage && aim_stage == main_req.aimstage) &&
       *                        ((aim_stage > pre_stage && flag == 0 &&
((\all int i = pre_stage,i<=aim_stage)
       *                                                    ==> (T_now ==
(\old)T_now + 0.5 && flag == 0) && ((\all int j = 0,j<RQ.size)
       *
==>(T == RQ(j).T && T<T_now && (RQ(j).Reqnum != -1 || RQ(j).valid == 2) &&
RQ(j).aimstage == i)
       *
&&(print(RQ(j)/(i,T_now)) && flag ==1 && RQ.size == (\old)RQ.size -
(\old)RQ.same_req.number - 1 && RQ.contains(RQ(j).same_req) == false &&
RQ.contains((\old)RQ(j)) == false && j == (\old)j - 1) &&
       *
```

```
(j == (\old)j + 1)

    *
                ) &&
        *
(flag == 1 && T_now == (\old)T_now + 1) &&
        *                                                                   (i
== (\old)i + 1)
        *                                                                ) &&
        *                                                               ( (\all
int i=0,i<RQ.size)
        *                                                                ==>
(RQ(i).T<T_now-1 && RQ(i).aimstage>aim_stage && RQ(i).Reqnum ==0)
&&(RQ(i).valid == 2) &&(out of loop) ||
        *                                                                    i
== (\old)i + 1
        *                                                                )
        *                        ) ||
        *                        ((aim_stage<pre_stage && flag == 0 &&
((\all int i = pre_stage && i>= aimstage)
        *                                                    ==> (T_now ==
(\old)T_now + 0.5 && flag == 0) && ((\all int j  =0,j<RQ.size)
        *
==>(T == RQ(j).T && T<T_now && (RQ(j).Reqnum != 1 || RQ(j).valid == 2) &&
RQ(j).aimstage == i)
        *
&&(print(RQ(j)/(i,T_now)) && flag ==1 && RQ.size == (\old)RQ.size -
(\old)RQ.same_req.number - 1 && RQ.contains(RQ.(j).same_req) == false &&
RQ.contains((\old)RQ(j)) == false && j == (\old)j - 1)&&
        *
(j == (\old)j + 1)

    *
                ) &&
        *
(flag == 1 && T_now == (\old)T_now + 1)
        *                                                                   (i
== (\old)i - 1)
        *                                                                ) &&
        *                                                                ( (\all
int i=0,i<RQ.size)
        *                                                                 ==>
(RQ(i)).T<T_now-1 && RQ(i).aimstage>aim_stage && RQ(i).Reqnum ==0)
&&(RQ(i).valid == 2) &&(out of loop) ||
```

```
        *                                                                   i ==
(\old)i + 1
        *                                                                )
        *
        *                          ) ||
        *                          (
        *                             (aim_stage==pre_stage&& T_now ==
(\old)T_now + 1 && print(RQ(m)/(aim_stage,Tnow)) && RQ.size ==
(\old)RQ.size - (\old)RQ.same_req.number - 1&& RQ.contains(RQ.(j).same_req)
== false && RQ.contains((\old)RQ(j)) == false && j == (\old)j - 1)
        *                          )
        *                         );
        */根据上述规格，获得如下划分：
```

< for each floor from i(which is equal to pre_stage + 1) to aim_stage ,T_now is plus 0.5 ,flag is set to 0 , this means we get to the floor I, then find each request from RQ to see if it can be carried, if it can be carried ,carry it and print some message, delete its same_req and delete itself , j is minus 1 and T_now is plus 1. After executing main_req and its portable requests, examine requests in RQ, if there is a request which direction is same with main_req and can be carried, set its valid to 2 and get out of the loop > with <RQ is not empty and main_req is the request whose valid is 2 in RQ ,T_now is the maximum of T_now and main_req.T , pre_stage is the previous aim_stage , aim_stage is main_req.aimstage and aim_stage is bigger than pre_stage >

< for each floor from i(which is equal to pre_stage - 1) to aim_stage ,T_now is plus 0.5 ,flag is set to 0 , this means we get to the floor I, then find each request from RQ to see if it can be carried, if it can be carried ,carry it and print some message, delete its same_req and delete itself , j is minus 1 and T_now is plus 1. After executing main_req and its portable requests, examine requests in RQ, if there is a request which direction is same with main_req and can be carried, set its valid to 2 and get out of the loop > with <RQ is not empty and main_req is the request whose valid is 2 in RQ ,T_now is the maximum of T_now and main_req.T , pre_stage is the previous aim_stage , aim_stage is main_req.aimstage and aim_stage is smaller than pre_stage >

<The elevator is still, T_now is plus 1and print some messages , delete main_req's same_reqs and delete itself > with <RQ is not empty and main_req is the request whose valid is 2 in RQ ,T_now is the maximum of T_now and main_req.T , pre_stage is the

previous aim_stage , aim_stage is main_req.aimstage and aim_stage is equal to pre_stage >

√ 在循环体中，当 RQ 队列不为空时，主请求为 RQ 队列中有效性为 2 的请求，T_now 是之前的 T_now 和 main_req.T 的最大值，pre_stage 是之前的 aim_stage,aim_stage 是 main_req.aimstage。当 aim_stage > pre_stage 时，主请求的方向是向上，电梯从第 i＝pre_stage+1 层到第 aim_stage 层，每到一层检查 RQ 中是否有可捎带的请求，若有可捎带请求，输出可捎带请求的执行信息，从 RQ 队列中删除可捎带请求的同质请求，删除该可捎带请求，并且执行开关门动作。执行完包括主请求的可捎带请求后，若 RQ 队列里有在主请求方向上的未执行的可捎带请求，将其 valid 设为 2 并且跳出循环。在此过程中只修改了 this 所管理的 RQ，并未对其他变量进行修改。因此，满足< for each floor from i(which is equal to pre_stage + 1) to aim_stage ,T_now is plus 0.5 ,flag is set to 0 , this means we get to the floor I, then find each request from RQ to see if it can be carried, if it can be carried ,carry it and print some message, delete its same_req and delete itself , j is minus 1 and T_now is plus 1. After executing main_req and its portable requests, examine requests in RQ, if there is a request which direction is same with main_req and can be carried, set its valid to 2 and get out of the loop > with <RQ is not empty and main_req is the request whose valid is 2 in RQ ,T_now is the maximum of T_now and main_req.T , pre_stage is the previous aim_stage , aim_stage is main_req.aimstage and aim_stage is bigger than pre_stage >

√ 在循环体中，当 RQ 队列不为空时，主请求为 RQ 队列中有效性为 2 的请求，T_now 是之前的 T_now 和 main_req.T 的最大值，pre_stage 是之前的 aim_stage,aim_stage 是 main_req.aimstage。当 aim_stage < pre_stage 时，主请求的方向是向下，电梯从第 i＝pre_stage-1 层到第 aim_stage 层，每到一层检查 RQ 中是否有可捎带的请求，若有可捎带请求，输出可捎带请求的执行信息，从 RQ 队列中删除可捎带请求的同质请求，删除该可捎带请求，并且执行开关门动作。执行完包括主请求的可捎带请求后，若 RQ 队列里有在主请求方向上的未执行的可捎带请求，将其 valid 设为 2 并且跳出循环。在此过程中只修改了 this 所管理的 RQ，并未对其他变量进行修改。因此，满足< for each floor from i(which is equal

to pre_stage - 1) to aim_stage ,T_now is plus 0.5 ,flag is set to 0 , this means we get to the floor I, then find each request from RQ to see if it can be carried, if it can be carried ,carry it and print some message, delete its same_req and delete itself , j is minus 1 and T_now is plus 1. After executing main_req and its portable requests, examine requests in RQ, if there is a request which direction is same with main_req and can be carried, set its valid to 2 and get out of the loop > with <RQ is not empty and main_req is the request whose valid is 2 in RQ ,T_now is the maximum of T_now and main_req.T , pre_stage is the previous aim_stage , aim_stage is main_req.aimstage and aim_stage is smaller than pre_stage >

√ 在循环体中，当 RQ 队列不为空时，主请求为 RQ 队列中有效性为 2 的请求，T_now 是之前的 T_now 和 main_req.T 的最大值，pre_stage 是之前的 aim_stage,aim_stage 是 main_req.aimstage。当 aim_stage = pre_stage 时，主请求的方向是保持 still，电梯执行开关门动作，输出执行信息，删除主请求的同质请求和主请求。在此过程中只修改了 this 所管理的 RQ，并未对其他变量进行修改。因此，满足<The elevator is still, T_now is plus 1and print some messages , delete main_req's same_reqs and delete itself > with <RQ is not empty and main_req is the request whose valid is 2 in RQ ,T_now is the maximum of T_now and main_req.T , pre_stage is the previous aim_stage , aim_stage is main_req.aimstage and aim_stage is equal to pre_stage >

（c）
```
public static void setRQ(RequQue rQ)
    /** @REQUIRES: rQ != null;
    @MODIFIES: \this.RQ;
    @EFFECTS: \this.RQ == rQ;
    */
```

<do nothing> with <rQ is null>

<RQ is set to rQ> with <rQ is not null>

√ 方法首先检查确认输入的 rQ 是否为 null，如果，是，执行结束，this 状态没有改变。因此满足<do nothing> with <rQ is null>

√ 如果 rQ 不是 null,则将 RQ 设为 rQ，方法修改了 this 中 RQ 的值。因此满足<RQ is set to rQ> with <rQ is not null>

（d）
```
public boolean repOK()
        /** @REQUIRES: None;
        @MODIFIES: None;
        @EFFECTS: \result == invariant(this);
        @ */
```

根据上述规格，获得如下划分：

<return false> with <RQ==null >

<return true> with <RQ!=null >

√若 RQ==null，返回 false，满足<return false> with <RQ==null >

√若 RQ!=null，返回 true，满足<return true> with <RQ!=null >

综上所述，所有的方法都满足规格。从而可以判断，New_sch 的实现是正确的，即满足规格要求。

4、Requ 类

1、抽象对象有效性论证

根据 Overview，Requ 类的表示对象为 aimstage, reqnum,T,通过抽象函数映射为能够返回和输出相应信息的请求类。

2、对象有效性论证

（a） 针对构造方法，论证对象的初始状态满足不变式，即 repOK 为真。

Requ 类提供了一个构造方法，Requ(),它初始化全部的 rep，repOK 的运行结果显然返回 true。

（b）Requ 类没有对象状态更改方法，因此任意方法的执行都不会改变 repOK 的值，repOK 一直为 true，不违背表示不变式。

（c）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。

3、方法实现正确性论证

（a）
```
public Requ(int a,int b,double c)
        /** @REQUIRES: a>0&&a<11;
         *           b==-1||b==0||b==1;
         *           c>=0;
```

```
@MODIFIES: \this;
@EFFECTS: \this.aimstage == a;
 *        \this.reqnum == b;
 *        \this.T == c;
@ */
```

根据上述过程规格，获得如下的划分：

<do nothing> with <a<1 or a>10 or b<-1 or b> 1 or c <0>

<aimstage is set to a , reqnum is set to b, T is set to c> with <a>=1&&a<-10 && (b==-1 || b == 0 || b==1) && c>=0>

√ 方法首先检查确认输入的 a, b, c 是否满足 a<1 or a>10 or b<-1 or b> 1 or c <0，如果是，执行结束，this 状态没有改变。因此满足<do nothing> with <rQ is null>

√ 如果输入的 a, b, c 满足 a>=1&&a<-10 && (b==-1 || b == 0 || b==1) && c>=0，aimstage 被设为 a, reqnum 被设为 b, T 被设为 c。在此过程中修改了 this 所管理的 aimstage,reqnum,T。因此满足<aimstage is set to a , reqnum is set to b, T is set to c> with <a>=1&&a<-10 && (b==-1 || b == 0 || b==1) && c>=0>

（b）
```
public String toString()
        /** @REQUIRES: reqnum == -1||reqnum == 0||reqnum == 1;
        @MODIFIES: None;
        @EFFECTS: reqnum == 0 ==> \result == "[ER," + aimstage + ","  +
(long)T + "]";
         *       reqnum == -1 ==> \result == "[FR," + aimstage +
",DOWN,"  + (long)T + "]";
         *       reqnum ==1 ==> \result == "[FR," + aimstage + ",UP,"  +
(long)T + "]";
         */
```

根据上述规格，获得如下划分：

<the return value is “error”> with <reqnum<-1 or reqnum>1>

<the returned value is “[ER,” + aimstage + “,” + (long)T + “]”> with <reqnum==0>

<the returned value is “[FR,” + aimstage + “,DOWN,” + (long)T + “]”> with <reqnum==-1>

<the returned value is “[FR,” + aimstage + “,UP,” + (long)T + “]”> with <reqnum==1>

√ 方法首先检查确认输入的 reqnum 是否满足 reqnum==-1||reqnum==0||reqnum==1，如果不是，返回值为“error”，this 状态没有改变。因此满足<the return value is “error”> with <reqnum<-1 or reqnum>1>

√ 如果输入的 reqnum 满足 reqnum==0，则返回值为“[ER,” + aimstage + “,” + (long)T + “]”，this 状态没有改变。因此满足<the returned value is “[ER,” + aimstage + “,” + (long)T + “]”> with <reqnum==0>

√ 如果输入的 reqnum 满足 reqnum==-1，则返回值为“[FR,” + aimstage + “,DOWN,” + (long)T + “]”，this 状态没有改变。因此满足<the returned value is “[FR,” + aimstage + “,DOWN,” + (long)T + “]”> with <reqnum==-1>

√ 如果输入的 reqnum 满足 reqnum==1，则返回值为“[FR,” + aimstage + “,UP,” + (long)T + “]”，this 状态没有改变。因此满足<the returned value is “[FR,” + aimstage + “,UP,” + (long)T + “]”> with <reqnum==1>

（c）
```
public boolean repOK()
        /** @REQUIRES: None;
        @MODIFIES: None;
        @EFFECTS: \result == invariant(this);
        @ */
```

根据上述规格，获得如下划分：

<return false> with <aimstage<1 or aimstage >10 or reqnum<-1 or reqnum >1 or T<0 >

<return true> with <aimstage>=1&&aimstage<=10&&reqnum>=-1&&reqnum <=1 &&T>=0 >

√ 若 aimstage<1 or aimstage >10 or reqnum<-1 or reqnum >1 or T<0，返回 false，满足<return false> with <aimstage<1 or aimstage >10 or reqnum<-1 or reqnum >1 or T<0 >

√ 若 aimstage>=1&&aimstage<=10&&reqnum>=-1&&reqnum <=1 &&T>=0，返回 true，满足<return true> with <aimstage>=1&&aimstage<=10&&reqnum>=-1&&reqnum <=1 &&T>=0 >

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

5、RequQue 类

1、抽象对象有效性论证

根据 Overview，RequQue 的表示对象为 queue，valid，通过抽象函数映射为能够加入和删除请求的请求队列。

2、对象有效性论证

（a）RequQue 初始状态满足不变式，即 repOK 为 true。

（b）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。

RequQue 类提供了七个状态更新方法：addReq, subReq, setvalid_none, setvalid_two, setquenone,setQueue, setValid，下面逐个进行论证。

⇨ 假设 addReq(Requ Rq)方法开始执行时，repOK 为 true。

1）该方法将 Rq 加入到 queue，同时将 1 加入到 valid，不会导致 queue==null，也不会导致 valid==null，因此不会导致 repOK 为 false。

2）因此，该方法的执行不会导致 repOK 为 false，不违背表示不变式。

⇨ 假设 subReq(int i)方法开始执行时，repOK 为 true。

1）该方法将 queue 队列中第 i 个 Requ 从队列里删除，将 valid 队列中第 i 个 Integer 从队列里删除，不会导致 queue==null，也不会导致 valid==null，因此不会导致 repOK 为 false。

2）因此，该方法的执行不会导致 repOK 为 false，不违背表示不变式。

⇨ 假设 setvalid_none()方法开始执行时，repOK 为 true。

1）该方法将 valid 队列中第 i 个 Integer 元素设为 0，不会导致 valid 为 null，不会改变 queue，因此不会导致 repOK 为 false。

2）因此，该方法的执行不会导致 repOK 为 false，不违背表示不变式。

⇨ 假设 setvalid_two()方法开始执行时，repOK 为 true。

1）该方法将 valid 队列中第 i 个 Integer 元素设为 2，不会导致 valid 为 null，

不会改变 queue，因此不会导致 repOK 为 false。

2）因此，该方法的执行不会导致 repOK 为 false，不违背表示不变式。

⇨ 假设 setquenone()方法开始执行时，repOK 为 true。

1）该方法将 queue 队列和 valid 队列中的元素清除，不会导致 queue==null,也不会导致 valid==null, 因此不会导致 repOK 为 false。

2）因此，该方法的执行不会导致 repOK 为 false，不违背表示不变式。

⇨ 假设 setQueue(ArrayList<Requ> 方法开始执行时，repOK 为 true。

1）该方法设置 queue，是用于测试的。正常程序执行不会用到该方法，因此不会导致 repOK 为 false。

2）因此，该方法的执行不会导致 repOK 为 false，不违背表示不变式。

⇨ 假设 setValid(ArrayList<Integer> valid1)方法开始执行时，repOK 为 true。

1）该方法设置 queue，是用于测试的。正常程序执行不会用到该方法，因此不会导致 repOK 为 false。

2）因此，该方法的执行不会导致 repOK 为 false，不违背表示不变式。

（c）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。

（d）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。

3、方法实现正确性论证

（a）
```
public void addReq(Requ Rq)
        /** @REQUIRES: (Rq != null);
        @MODIFIES: \this.queue,\this.valid;
        @EFFECTS: \this.queue.size == \old(\this.queue.size) + 1 &&
\this.queue.contains(Rq) == true;
        *        \this.valid.size == \old(\this.valid.size) + 1 &&
\this.queue.contains(1) == true;
        @ */
```

根据上述规格，获得如下划分：

<do nothing> with <Rq == null>

<Rq is add to queue and 1 is add to valid> with <Rq !=null>

√方法首先检查确认输入的 Rq 是否满足 REQUIRES 要求，若 Rq==null，则 直接返回结束，满足<do nothing> with <Rq == null>

√ 如果输入的 Rq 满足 Rq！=null，则将 Rq 加入 queue 队列，将 1 加入 valid 队列，在此过程中修改了 this 管理的 queue 和 valid，满足<Rq is add to queue and 1 is add to valid> with <Rq !=null>

（b）
```
public void subReq(int i){
        /** @REQUIRES: (i>=0 && i<queue.size && i<valid.size);
        @MODIFIES: queue,valid;
        @EFFECTS: \this.queue.size == \old(\this.queue.size) - 1 &&
\this.queue.contains(\old(\this.queue).get(i)) == false;
        *        \this.valid.size == \old(\this.valid.size) - 1 ;
        @ */
```

根据上述规格，获得如下划分：

<do nothing> with <i < 0 or i >= queue.size>

<queue.remove(i) and valid.remove(i)> with <i>=0&&i<queue.size>

√方法首先检查确认输入的 i 是否满足 REQUIRES 要求，若 i 不满足 REQUIRES 要求（即 i<0||i>=queue.size），则 直接返回结束，满足<do nothing> with <i < 0 or i >= queue.size>

√ 如果输入的 i 满足 i>=0&&i<queue.size，则删除 queue 的第 i 个元素和 valid 的第 i 个元素。在此过程中修改了 this 所管理的 queue 和 valid，满足<queue.remove(i) and valid.remove(i)> with <i>=0&&i<queue.size>

（c）
```
public boolean repOK()
        /**@REQUIRES: None;
        @MODIFIES: None;
        @EFFECTS: \result == invariant(this);
        */
```

<return false> with <queue == null or valid == null>

<return true> with <queue != null && queue != null>

√若 queue 为 null 或 valid 为 null，返回 false，满足<return false> with <queue == null or valid == null>

√若 queue！=null 且 valid！=null，返回 true，满足<return true> with <queue != null && queue != null>

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