# 类正确性论证报告

## Main类

### 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。

### 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。

### 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("INVALID["+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。

## 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

### 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。

### 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。

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。

### 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。