第14次OO作业-电梯有效性分析报告

一、Controller类

**1.Overview论证**

Overview书写如下：

/\*\*

\* **@OVERVIEW**:

\* 这个类是前一次作业调度器类，存储有当前系统时间，上一次指令执行时间和各个按钮亮灯情况等信息。

\* 可以通过构造方法就上述信息进行初始化操作。

\*/

其中说到了该抽象类具有的功能是储存当前系统时间、上一次指令执行完成时间、各个按钮量等情况信息，相应的代码中的实现如下：

**protected** **double** now\_time = 0;

**protected** **double** last\_time = 0;

**protected** **double** ele[] = **new** **double**[Element.***Floor\_number***];

**protected** **double** floor\_up[] = **new** **double**[Element.***Floor\_number***];

**protected** **double** floor\_down[] = **new** **double**[Element.***Floor\_number***];

其中now\_time表示当前系统时间，last\_time表示上一次指令完成时间，ele[11]表示电梯各楼层按钮最近一次灭灯时间，floor\_up表示各个楼层上行按钮最近一次灭灯时间，floor\_down表示各个楼层下行按钮最近一次灭灯时间。这5个数据类型能够表示Overview中的抽象对象所有可能的取值，同时由于指导书中有对楼层的数量限制—10层，和系统时间的限制—小于等于U\_INT的最大取值，同时有0.5S的需求，double类型均能满足。

该类的函数为：

/\*\*

\* 注释.......

\* **@REQUIRES**: None;

\* **@MODIFIES** : None;

\* **@EFFECTS** : now\_time == 0;

\* last\_time == 0;

\* (\all int i;0 <= i <= 10) ==> ele[i] == -1;

\* (\all int i;0 <= i <= 10) ==> floor\_up[i] == -1;

\* (\all int i;0 <= i <= 10) ==> floor\_down[i] == -1;

\*/

**public** Controller(){

...

}

可以满足初始化上述数据的功能，为了满足判断按钮时间左闭右闭，将按钮初始时间置为-1而非0，系统时间和上一次指令完成时间均为0。因此，抽象函数结果覆盖了Overview中描述的抽象对象。

**2.对象有效性论证**

先看看这个类的RepOK：

/\*\*

\* 注释.......

\* **@REQUIRES**: None;

\* **@MODIFIES** : None;

\* **@EFFECTS** :

\* (now\_time >= 0 && last\_time >= 0 && ele != null && floor\_up != null && floor\_down != null) ==> \result == true;

\* !(now\_time >= 0 && last\_time >= 0 && ele != null && floor\_up != null && floor\_down != null) ==> \result == false;

\*/

**public** **boolean** repOK(){

**boolean** result = **false**;

**if**(now\_time >= 0 && last\_time >= 0 && ele != **null** && floor\_up != **null** && floor\_down != **null**)

result = **true**;

**return** result;

}

其中，repOK为真的条件是系统时间大于等于0，上一次指令完成时间大于等于0，同时三个按钮组有相应的存储方式不为空；其余情况则返回false。

下面看各个方法是否会改变repOK的正确性，由于该类只有构造方法，所以只对构造方法进行论证，代码如下：

**public** Controller(){

now\_time = 0;

last\_time = 0;

**int** i;

**for**(i = 0; i < Element.***FLOOR\_NUMBER***; i++){

ele[i] = -1;

floor\_up[i] = -1;

floor\_down[i] = -1;

}

}

构造方法无参数传入，所以不存在直接共享传入的对象的可能。同时，将ele、floor\_up、floor\_down三个按钮组的引用到实例的操作是在实例化Controller类的时候就完成了的，所以在这里只是将其中的数据就实现需要进行赋值，并未修改引用，所以三个按钮组不会为null。而系统当前时间初始化为0，上一次指令完成时间初始化为0，同样满足repOK的要求。所以，综上，构造方法不会使repOK为false。

**3.方法实现的正确性**

构造方法的规格和实现如下：

/\*\*

\* 注释.......

\* **@REQUIRES**: None;

\* **@MODIFIES** : this.now\_time, this.last\_time, this.ele[], this.floor\_up[], this.floor\_down[];

\* **@EFFECTS** : now\_time == 0;

\* last\_time == 0;

\* (\all int i;0 <= i <= 10) ==> ele[i] == -1;

\* (\all int i;0 <= i <= 10) ==> floor\_up[i] == -1;

\* (\all int i;0 <= i <= 10) ==> floor\_down[i] == -1;

\*/

**public** Controller(){

now\_time = 0;

last\_time = 0;

**int** i;

**for**(i = 0; i < Element.***FLOOR\_NUMBER***; i++){

ele[i] = -1;

floor\_up[i] = -1;

floor\_down[i] = -1;

}

}

首先，判断规格是否正确有效。构造方法没有异常抛出，所以没有体现；前置条件为空，后置条件均可判定；且modifies的范围可确定，而且与具体实现吻合。综上，应判断出规格是合理的。

之后，由于没有对前置条件的要求，且无参数传入，所以每次方法执行的结果都是相同的而且是可预知的，同样，是可以满足后置条件的。而且因为modifies覆盖全面，所以不存在会修改副作用之外的非局部变量的可能。因为没有前置条件的要求，所以就没有依据前置条件和后置条件划分找出分类树，并分析每个划分对应后置条件得到决策表了。所以可见，该方法的实现是正确的。

综上，由于Controller类的抽象正确，对象实例化保证repOK正确，各个方法的实现符合规格正确，所以该类是正确的。

二、Begin类

**1.Overview的论证**

Overview书写如下：

/\*\*

\* **@OVERVIEW**:

\* 这个类是程序启动类，负责管理请求队列、电梯、楼层和调度器，同时具有读取数据

\* 并将合理请求添加进请求队列的功能。

\*/

其中说到了该抽象类具有的功能之一是统一管理请求队列、电梯、楼层和调度器，相应的代码中的实现如下：

**static** **private** RequestList *requestList* = **new** RequestList();

**static** **private** Elevator *elevator* = **new** Elevator();

**static** **private** Floor *floor* = **new** Floor();

**static** **private** Scheduler *scheduler* = **new** Scheduler();

整个程序中只有这个类中实例化了这四个对象，之后所有的操作都是这四个类之间相互调度，并通过不同的Request来实现不同的操作的。

Overview中还说道需要具有读取数据并将合理的请求添加进入请求队列的功能，所以实现了read方法：

/\*\*

\* 注释.......

\* **@REQUIRES**: requestList != null;

\* **@MODIFIES**: requestList;

\* **@EFFECTS** :

\* (System.in.matches("FR,%d1,UP,%d2") == true && 1 <= %d1 < 10 && 2^32 - 1 >= %d2 >= 0 && 新请求满足指令数限制要求 && 满足时间顺序递增) ==> (floor.make\_Request() == true && requestList.add\_Request() == true)

\* (System.in.matches("FR,%d1,DOWN,%d2") == true && 1 < %d1 <= 10 && 2^32 - 1 >= %d2 >= 0 && 新请求满足指令数限制要求 && 满足时间顺序递增) ==> (floor.make\_Request() == true && requestList.add\_Request() == true)

\* (System.in.matches("ER,%d1,UP,%d2") == true && 1 <= %d1 < 10 && 2^32 - 1 >= %d2 >= 0 && 新请求满足指令数限制要求 && 满足时间顺序递增) ==> (floor.make\_Request() == true && requestList.add\_Request() == true)

\* (System.in.matches("ER,%d1,DOWN,%d2") == true && 1 < %d1 <= 10 && 2^32 - 1 >= %d2 >= 0 && 新请求满足指令数限制要求 && 满足时间顺序递增) ==> (floor.make\_Request() == true && requestList.add\_Request() == true)

\* (!System.in.matches("ER/FR,%d,UP/DOWN,%d") == true && !System.in.matches("RUN")) ==> Invalid(System.in) == true;

\* (!System.in.matches("ER/FR,%d,UP/DOWN,%d") == true && System.in.matches("RUN")) ==> 停止输入 == true;

\*/

**public** **static** **void** read(RequestList requestList){

...

}

通过read的格式和参数可以看出，该方法会对控制台的输入进行正则匹配，并将合理的请求加入到requestList中。因此，抽象函数结果覆盖了Overview中描述的抽象对象。

**2.对象有效性的论证**

先看看这个类的RepOK：

/\*\*

\* 注释.......

\* **@REQUIRES**: None;

\* **@MODIFIES** : None;

\* **@EFFECTS** :

\* (requestList != null && elevator != null && floor != null && scheduler != null) ==> \result == true;

\* !(requestList != null && elevator != null && floor != null && scheduler != null) ==> \result == false;

\*/

**public** **boolean** repOK(){

**if**(*requestList* != **null** && *elevator* != **null** && *floor* != **null** && *scheduler* != **null**)

**return** **true**;

**else** {

**return** **false**;

}

}

其中，repOK为真的条件是Begin类四个引用都有对象而非空，说明Begin类的静态变量成功实例化。

下面看各个方法是否会改变repOK的正确性，由于该类有两个方法，所以对两个方法分别进行论证，代码如下：

/\*\*

\* 注释.......

\* **@REQUIRES**: None;

\* **@MODIFIES**: this.requestList;

\* **@EFFECTS** : None;

\*/

**public** **static** **void** main(String args[]){

**try**{

Begin.*read*(*requestList*);

**while**(*requestList*.is\_Empty() != **true**){

*scheduler*.next\_step(*requestList*, *elevator*);

}

}**catch** (Exception e) {

System.***out***.println("ERROR");

}

}

这是第一个函数，main主函数。其中，因为requestList、elevator、floor和scheduler都是private类型，所以只有scheduler类可能对requestList和elevator两个对象进行修改，还有this.read可能修改requestList，下面让我们来看看这两个方法的规格。

/\*\*

\* 注释.......

\* **@REQUIRES**: requestList != null;

\* **@MODIFIES**: requestList;

\* **@EFFECTS** :

\* (System.in.matches("FR,%d1,UP,%d2") == true && 1 <= %d1 < 10 && 2^32 - 1 >= %d2 >= 0 && 新请求满足指令数限制要求 && 满足时间顺序递增) ==> (floor.make\_Request() == true && requestList.add\_Request() == true)

\* (System.in.matches("FR,%d1,DOWN,%d2") == true && 1 < %d1 <= 10 && 2^32 - 1 >= %d2 >= 0 && 新请求满足指令数限制要求 && 满足时间顺序递增) ==> (floor.make\_Request() == true && requestList.add\_Request() == true)

\* (System.in.matches("ER,%d1,UP,%d2") == true && 1 <= %d1 < 10 && 2^32 - 1 >= %d2 >= 0 && 新请求满足指令数限制要求 && 满足时间顺序递增) ==> (floor.make\_Request() == true && requestList.add\_Request() == true)

\* (System.in.matches("ER,%d1,DOWN,%d2") == true && 1 < %d1 <= 10 && 2^32 - 1 >= %d2 >= 0 && 新请求满足指令数限制要求 && 满足时间顺序递增) ==> (floor.make\_Request() == true && requestList.add\_Request() == true)

\* (!System.in.matches("ER/FR,%d,UP/DOWN,%d") == true && !System.in.matches("RUN")) ==> Invalid(System.in) == true;

\* (!System.in.matches("ER/FR,%d,UP/DOWN,%d") == true && System.in.matches("RUN")) ==> 停止输入 == true;

\*/

**public** **static** **void** read(RequestList requestList){

...

}

这里论证需要假设其他代码都是正确的，是按照规格来实现的。

我们发现，this.read中只是对requestList进行添加成员操作，没有修改引用对象。

/\*\*

\* 注释.......

\* **@REQUIRES**: list != null && elevator != null;

\* **@MODIFIES** : this.main\_request, this.picking\_request, this.list, this.elevator;

\* **@EFFECTS** :

\* (main\_request == -1) ==> this.main\_request == search\_main\_one(list, elevator);

\* (main\_request == -1 && check\_duplicate(list, main\_request)) ==> (ignore(list , main\_request) == true && main\_request == -1);

\* (main\_request == -1 && !check\_duplicate(list, main\_request)) ==> change\_elevator\_dirction(list, elevator, main\_request) == true;

\* (main\_request != -1) ==> this.picking\_request = search\_picking\_one(list, elevator);

\* (main\_request != picking\_request && picking\_request != -1 && check\_duplicate(list, picking\_request)) ==> ignore(list, picking\_request) == true;

\* (main\_request != picking\_request && picking\_request != -1 && !check\_duplicate(list, picking\_request)) ==> excecute(list, elevator, picking\_request) == true;

\* !(main\_request != picking\_request && picking\_request != -1) ==> excecute(list, elevator, main\_request) == true;

\*/

**public** **void** next\_step(RequestList list , Elevator elevator){

...

}

同理，如果next\_step也是按照规格来实现的的话，发现只是修改elevator的状态，删除requestList中的元素，没有修改引用的操作。

所以，可以得出，静态变量实例化之后，Begin类repOK恒为true。

**3.方法实现的正确性**

read方法的规格和实现如下：

/\*\*

\* 注释.......

\* **@REQUIRES**: requestList != null;

\* **@MODIFIES**: requestList;

\* **@EFFECTS** :

\* (如果是第一个 == true && System.in == "FR,1,UP,0") ==> (floor.make\_Request() == true && requestList.add\_Request() == true)

\* (如果是第一个 == false && System.in.matches("FR,%d1,UP,%d2") == true && 1 <= %d1 < 10 && 2^32 - 1 >= %d2 >= 0 && 新请求满足指令数限制要求 && 满足时间顺序递增) ==> (floor.make\_Request() == true && requestList.add\_Request() == true)

\* (如果是第一个 == false && System.in.matches("FR,%d1,DOWN,%d2") == true && 1 < %d1 <= 10 && 2^32 - 1 >= %d2 >= 0 && 新请求满足指令数限制要求 && 满足时间顺序递增) ==> (floor.make\_Request() == true && requestList.add\_Request() == true)

\* (如果是第一个 == false && System.in.matches("ER,%d1,UP,%d2") == true && 1 <= %d1 < 10 && 2^32 - 1 >= %d2 >= 0 && 新请求满足指令数限制要求 && 满足时间顺序递增) ==> (floor.make\_Request() == true && requestList.add\_Request() == true)

\* (如果是第一个 == false && System.in.matches("ER,%d1,DOWN,%d2") == true && 1 < %d1 <= 10 && 2^32 - 1 >= %d2 >= 0 && 新请求满足指令数限制要求 && 满足时间顺序递增) ==> (floor.make\_Request() == true && requestList.add\_Request() == true)

\* (!System.in.matches("ER/FR,%d,UP/DOWN,%d") == true && !System.in.matches("RUN") || (如果是第一个 == true && !System.in == "FR,1,UP,0")) ==> Invalid(System.in) == true;

\* (!System.in.matches("ER/FR,%d,UP/DOWN,%d") == true && System.in.matches("RUN")) ==> 停止输入 == true;

\*/

首先，判断规格是否正确有效。read方法没有异常抛出，所以没有体现；前置条件为空，后置条件均可判定；且modifies的范围可确定，而且与具体实现吻合。综上，应判断出规格是合理的。

之后，该方法没有显示的参数传入，但是有System.in作为系统控制台输入，所以，现在对后置条件根据System.in进行分类：

①<Floor.make\_request() && requestLIst.add\_request()> with <System.in == “FR,des,UP/DOWN,tim” && 1 <= des <= 10 && 0 <= tim <= 2^32 – 1 && 按照时间递增 && 小于100条 && !是第一个>

②<Elevator.make\_request() && requestLIst.add\_request()> with <System.in == “ER,des,UP/DOWN,tim” && 1 <= des <= 10 && 0 <= tim <= 2^32 – 1 && 按照时间递增 && 小于100条 && !是第一个>

③<Invalid> with <(!(System.in == “FR,des,UP/DOWN,tim”) || !(1 <= des <= 10) || !(0 <= tim <= 2^32 – 1) || !按照时间递增 || !小于100条) && System.in != “RUN” || 是第一个 && System.in != “FR,1,UP,0”>

④<Stop Input> with <System.in == “RUN”>

**public** **static** **void** read(RequestList requestList){

...

}

根据上述四种分类，发现

①如果合理如<FR,1,UP,0>，符合正则表达，同时是FR指令，所以走向第一个判断的第一个分支，第二个判断的第二个分支。之后，对des、dir和tim进行分析并判断，发现均合理，最后执行操作new\_Request = floor.make\_Request(Element.FLOOR, des, dir, time)、requestList.add\_Request(new\_Request)，并且更新last\_time = time。

②如<ER,1,UP,0>，符合正则表达，同时是ER指令，所以走向第一个判断的第一个分支，第二个判断的第一个分支。之后，对des、dir和tim进行分析并判断，发现均合理，最后执行操作new\_Request = elevator.make\_Request(Element.FLOOR, des, dir, time)、requestList.add\_Request(new\_Request)，并且更新last\_time = time。

③如<ERR,1,UP,0>，不符合正则表达，所以走向第一个判断的第二个分支，之后又与RUN进行判断，发现同样不匹配，所以被当做Invalid输入，System.out.printf("INVALID[%s]\n" , processed\_line)。

④RUN，不符合正则表达，所以走向第一个判断的第二个分支，之后又与“RUN”进行匹配，发现成功，所以is\_over = true, input.close()，停止输入。

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

三、Element类

由于这个类是单独的用于存放类似于宏定义的常量的，所以没有任何方法，只有static final静态常量，所以就没有写Overview和repOK了，这个类无条件正确。

四、Elevator\_method接口

由于这个接口没有局部变量，所以没有repOK，就简要对Overview进行分析。

/\*\*

\* **@OVERVIEW**:

\* 这个类是电梯接口，希望电梯可以实现产生电梯请求、移动、改变自身状态，

\* 获取自身状态、输出打印信息等操作。

\*/

**public** **interface** Elevator\_method {

**public** Request make\_Request(**int** sym , **int** des , **int** dir , **long** tim);

**public** **void** move\_to(**int** des);

**public** **int** get\_floor();

**public** **void** set\_time(**double** time);

**public** **double** get\_time();

**public** **void** set\_dirction(**int** dir);

**public** **int** get\_direction();

**public** String print\_direction();

**public** String toString();

}

应Overview要求，Elevator类应该可以产生请求，所以有make\_request方法，三对获取更改自身状态的方法，move\_to、get\_floor，set\_time、get\_time，set\_direction、get\_direction，和一堆负责输出信息的方法print\_direction和toString，完全满足了Overview中对接口的抽象，所以该接口实现正确。

五、Elevator类

**1.Overview论证**

Overview书写如下：

/\*\*

\* **@OVERVIEW**:

\* 这个类是电梯类，继承电梯运动接口，储存有当前楼层，当前运行时间和运动方向。

\* 可以进行产生电梯请求，移动等操作。

\*/

其中说到了该抽象类具有的功能之一是存储电梯所在的当前楼层，当前为止的运行时间和电梯的运动方向，相应的代码中的实现如下：

**Private** **int** now\_floor = 1;

**private** **double** now\_time = 0;

**private** **int** direction = element.***elevator\_still***;

电梯的初始状态为在一层，系统时间为0，状态为Still静止状态。

Overview中说到了需要继承elevator\_method接口，其中需要实现的操作都在接口中定义了，关于接口的论证上面已经论证过，所以Overview中的抽象合理且被实现。

**public** **class** Elevator **implements** Elevator\_method{

**...**

**}**

**2.对象有效性的论证**

先看看Elevator类的RepOK：

/\*\*

\* 注释.......

\* **@REQUIRES**: None;

\* **@MODIFIES**: None;

\* **@EFFECTS** :

\* (now\_floor >= 1 && now\_floor <= 10 && now\_time >= 0 && (direction == Element.Elevator\_down || direction == Element.Elevator\_still || direction == Element.Elevator\_up)) ==> \result == true;

\* !(now\_floor >= 1 && now\_floor <= 10 && now\_time >= 0 && (direction == Element.Elevator\_down || direction == Element.Elevator\_still || direction == Element.Elevator\_up)) ==> \result == false;

\*/

**public** **boolean** repOK(){

**boolean** result = **false**;

**if**(now\_floor >= 1 && now\_floor <= 10 && now\_time >= 0 && (direction >= 0 && direction <= 2))

result = **true**;

**return** result;

}

其中，repOK为真的条件电梯的楼层在1到10楼之间，系统当前时间大于等于0，方向为向下、向上或静止。

下面看各个方法是否会改变repOK的正确性，由于该类有八个方法，为了简化论证，我们先看看每个方法都是做什么的，因为这个类只有三个private变量，所以只需要在自己类内，看看对这三个变量的修改即可，我们根据modifies即可删掉那些没有修改这三个变量的方法以简化，发现只有三个方法modifies不为None，代码如下：

/\*\*

\* 注释.......

\* **@REQUIRES**: 1 <= des <= 10;

\* **@MODIFIES** : this.now\_floor;

\* **@EFFECTS** :

\* this.now\_floor == des;

\*/

**public** **void** move\_to(**int** des){

now\_floor = des;

}

第一个是电梯的移动方法，这个方法是在用参数传递进来的des，修改this.now\_floor。在requires中我们可以看到，对des的限制为1 <= des <= 10，这说明this.now\_floor >= 1 && this.now\_floor <= 10，不会使repOK为false。

/\*\*

\* 注释.......

\* **@REQUIRES**: time >= 0;

\* **@MODIFIES** : this.now\_time;

\* **@EFFECTS** :

\* this.now\_time == time;

\*/

**public** **void** set\_time(**double** time){

now\_time = time;

}

第二个是电梯的修改系统时间方法，这个方法是在用参数传递进来的time，修改this.now\_time。在requires中我们可以看到，对time的限制为time >= 0，这说明this.now\_time >= 0，不会使repOK为false。

/\*\*

\* 注释.......

\* **@REQUIRES**: 0 <= dir <= 2;

\* **@MODIFIES** : this.direction;

\* **@EFFECTS** :

\* this.direction == dir;

\*/

**public** **void** set\_dirction(**int** dir){

**this**.direction = dir;

}

第三个是电梯的修改方向方法，这个方法是在用参数传递进来的dir，修改this.direction。在requires中我们可以看到，对time的限制为0 <= direction <= 2，这说明this.direction >= 0 && this.direction <= 2，即this.direction == Element.ELEVATOR\_UP || this.direction == ELEMENT.ELEVATOR\_DOWN || this.direction == ELEMENT.ELEVATOR\_STILL，不会使repOK为false。

其余方法均没有改变this，所以Elevator类任意对象始终保持着对象的有效性。

**3.方法实现的正确性**

九种方法的规格和实现如下：

①产生请求

/\*\*

\* 注释.......

\* **@REQUIRES**: 0 <= sym <= 1 && 1 <= des <= 10 && 0 == dir && 2^32 - 1 >= tim >= 0;

\* **@MODIFIES** : None;

\* **@EFFECTS** :

\* \result == new Request(sym, des, dir, tim);

\*/

**public** Request make\_Request(**int** sym , **int** des , **int** dir , **long** tim){

Request new\_Request = **new** Request(sym, des, dir, tim);

**return** new\_Request;

}

首先，判断规格是否正确有效。Make\_Request方法没有异常抛出，所以没有体现；前置条件与后置条件均可判定；且不对任何变量进行修改。综上，应判断出规格是合理的。

之后，该方法有四个参数作为输入，并且均有要求。所以根据前置条件和后置条件进行分类，在合理的情况下，仅有一种情况：

<make\_request()> with < 0 <= sym <= 1 && 1 <= des <= 10 && 0 == dir && 2^32 - 1 >= tim >= 0>

所以输入合理的情况下，会根据输入new一个Request，并且return，与规格吻合。

②移动方法

/\*\*

\* 注释.......

\* **@REQUIRES**: 1 <= des <= 10;

\* **@MODIFIES** : this.now\_floor;

\* **@EFFECTS** :

\* this.now\_floor == des;

\*/

**public** **void** move\_to(**int** des){

now\_floor = des;

}

首先，判断规格是否正确有效。Move\_to方法没有异常抛出，所以没有体现；前置条件与后置条件均可判定；且对指定变量now\_floor进行修改。综上，应判断出规格是合理的。

之后，该方法有一个参数作为输入，且有要求。所以根据前置条件和后置条件进行分类，在合理的情况下，仅有一种情况：

<now\_floor = des> with <1 <= des <= 10>

所以在des合理的情况下，执行now\_floor = des，符合规格要求。

③设置时间

/\*\*

\* 注释.......

\* **@REQUIRES**: time >= 0;

\* **@MODIFIES** : this.now\_time;

\* **@EFFECTS** :

\* this.now\_time == time;

\*/

**public** **void** set\_time(**double** time){

now\_time = time;

}

首先，判断规格是否正确有效。Set\_time方法没有异常抛出，所以没有体现；前置条件与后置条件均可判定；且对指定变量now\_time进行修改。综上，应判断出规格是合理的。

之后，该方法有一个参数作为输入，且有要求。所以根据前置条件和后置条件进行分类，在合理的情况下，仅有一种情况：

<now\_time = time> with <time >= 0>

所以在time合理的情况下，执行now\_time = time，符合规格要求。

④获取时间

/\*\*

\* 注释.......

\* **@REQUIRES**: None;

\* **@MODIFIES** : None;

\* **@EFFECTS** :

\* \result == this.now\_time;

\*/

**public** **double** get\_time(){

**return** now\_time;

}

首先，判断规格是否正确有效。Get\_time方法没有异常抛出，所以没有体现；前置条件为None，后置条件可判定；且未对this进行修改。综上，应判断出规格是合理的。

之后，该方法没有参数作为输入。所以根据前置条件和后置条件进行分类，在合理的情况下，仅有一种情况：

<\result = now\_time> with <nothing>

所以无论如何，会返回this.now\_time，符合规格要求。

⑤获取楼层

/\*\*

\* 注释.......

\* **@REQUIRES**: None;

\* **@MODIFIES** : None;

\* **@EFFECTS** :

\* \result == this.now\_floor;

\*/

**public** **int** get\_floor(){

**return** now\_floor;

}

首先，判断规格是否正确有效。Get\_floor方法没有异常抛出，所以没有体现；前置条件为None，后置条件可判定；且未对this进行修改。综上，应判断出规格是合理的。

之后，该方法没有参数作为输入。所以根据前置条件和后置条件进行分类，在合理的情况下，仅有一种情况：

<\result = now\_floor> with <nothing>

所以无论如何，会返回this.now\_floor，符合规格要求。

⑥设置方向

/\*\*

\* 注释.......

\* **@REQUIRES**: 0 <= dir <= 2;

\* **@MODIFIES** : this.direction;

\* **@EFFECTS** :

\* this.direction == dir;

\*/

**public** **void** set\_dirction(**int** dir){

**this**.direction = dir;

}

首先，判断规格是否正确有效。Set\_direction方法没有异常抛出，所以没有体现；前置条件与后置条件均可判定；且对指定变量direction进行修改。综上，应判断出规格是合理的。

之后，该方法有一个参数作为输入，且有要求。所以根据前置条件和后置条件进行分类，在合理的情况下，仅有一种情况：

<this.direction = dir> with <0 <= dir <= 2>

所以在dir合理的情况下，执行this.direction = dir，符合规格要求。

⑦获取方向

/\*\*

\* 注释.......

\* **@REQUIRES**: None;

\* **@MODIFIES** : None;

\* **@EFFECTS** :

\* \result == this.direction;

\*/

**public** **int** get\_direction(){

**return** **this**.direction;

}

首先，判断规格是否正确有效。Get\_direction方法没有异常抛出，所以没有体现；前置条件为None，后置条件可判定；且未对this进行修改。综上，应判断出规格是合理的。

之后，该方法没有参数作为输入。所以根据前置条件和后置条件进行分类，在合理的情况下，仅有一种情况：

<\result = this.direction> with <nothing>

所以无论如何，会返回this.direction，符合规格要求。

⑧得到方向的字符串

/\*\*

\* 注释.......

\* **@REQUIRES**: None;

\* **@MODIFIES** : None;

\* **@EFFECTS** :

\* (direction == 0) ==> (\result == "UP");

\* (direction == 1) ==> (\result == "DOWN");

\* (direction != 0 && direction != 1) ==> (\result == "STILL");

\*/

**public** String print\_direction(){

**if**(direction == 0)

**return** "UP";

**else** **if**(direction == 1)

**return** "DOWN";

**else** {

**return** "STILL";

}

}

首先，判断规格是否正确有效。print\_direction方法没有异常抛出，所以没有体现；前置条件为None，后置条件可判定；且未对this进行修改。综上，应判断出规格是合理的。

之后，该方法没有参数作为输入。所以根据前置条件和后置条件进行分类，在合理的情况下，有三种情况：

<\result = “UP”> with <direction == 0>

<\result = “DOWN”> with <direction == 1>

<\result = “STILL”> with <direction == 2>

因为direction范围是0到2，所以else即为direction == 2，发现当direction为012中不同数值时，返回值确实为Effects中规定，符合规格要求。

⑨电梯信息转字符串

/\*\*

\* 注释.......

\* **@REQUIRES**: None;

\* **@MODIFIES** : None;

\* **@EFFECTS** :

\* \result == String.format("(%d,%s,%.1f)", now\_floor , print\_direction() , now\_time);

\*/

**public** String toString(){

**return** String.*format*("(%d,%s,%.1f)", now\_floor , print\_direction() , now\_time);

}

首先，判断规格是否正确有效。toString方法没有异常抛出，所以没有体现；前置条件为None，后置条件可判定；且未对this进行修改。综上，应判断出规格是合理的。

之后，该方法没有参数作为输入。所以根据前置条件和后置条件进行分类，在合理的情况下，有一种情况：

<\result = String.format("(%d,%s,%.1f)", now\_floor , print\_direction() , now\_time)> with <nothing>

因为对象始终有效，所以打印出的String也有效，符合规格要求。

五、Floor类

**1.Overview论证**

Overview书写如下：

/\*\*

\* **@OVERVIEW**:

\* 这个类是楼层类，不负责数据存储，只负责产生楼层请求。

\*/

Overview中说到了该类的唯一功能就是产生楼层请求。

/\*\*

\* 注释.......

\* **@REQUIRES**: 0 <= sym <= 1 && 1 <= des <= 10 && 0 <= dir <= 1 && 2^32 - 1 >= tim >= 0;

\* **@MODIFIES** : None;

\* **@EFFECTS** :

\* \result == new Request(sym, des, dir, tim);

\*/

**public** Request make\_Request(**int** sym , **int** des , **int** dir , **long** tim){

...

}

确实有方法实现了产生请求的作用，而且没有任何成员变量，Overview的抽象正确。

**2.对象有效性的论证**

因为这个类没有成员变量，所以对象有效性恒为true。

**3.方法实现的正确性**

Make\_request方法的规格和实现如下：

/\*\*

\* 注释.......

\* **@REQUIRES**: 0 <= sym <= 1 && 1 <= des <= 10 && 0 <= dir <= 1 && 2^32 - 1 >= tim >= 0;

\* **@MODIFIES** : None;

\* **@EFFECTS** :

\* \result == new Request(sym, des, dir, tim);

\*/

**public** Request make\_Request(**int** sym , **int** des , **int** dir , **long** tim){

Request new\_Request = **new** Request(sym , des , dir , tim);

**return** new\_Request;

}

首先，判断规格是否正确有效。Make\_Request方法没有异常抛出，所以没有体现；前置条件与后置条件均可判定；且不对任何变量进行修改。综上，应判断出规格是合理的。

之后，该方法有四个参数作为输入，并且均有要求。所以根据前置条件和后置条件进行分类，在合理的情况下，仅有一种情况：

<make\_request()> with < 0 <= sym <= 1 && 1 <= des <= 10 && 0 <= dir <= 1 && 2^32 - 1 >= tim >= 0>

所以输入合理的情况下，会根据输入new一个Request，并且return，与规格吻合。

综上，Floor类的实现正确。

六、Request类

**1.Overview论证**

Overview书写如下：

/\*\*

\* **@OVERVIEW**:

\* 这个类是请求类，负责存储电梯or楼层请求，请求位置，请求方向，请求时间。

\* 只能通过构造方法初始化上述数据，其余时刻只能获取不能修改。

\*/

Overview中说到了该类负责存储请求类型即symbol，请求位置即destination，请求方向即direction，请求时间即time。

**private** **int** symbol;

**private** **int** destination;

**private** **int** direction;

**private** **long** time;

并且只能通过构造方法进行初始化上述数据，发现只有构造方法的modifies不为None：

/\*\*

\* 注释.......

\* **@REQUIRES**: 0 <= sym <= 1 && 1 <= des <= 10 && 0 <= dir <= 1 && 2^32 - 1 >= tim >= 0;

\* **@MODIFIES** : this.symbol, this.destination, this.direction, this.time;

\* **@EFFECTS** :

\* \result == new Request(sym, des, dir, tim);

\*/

**public** Request(**int** sym , **int** des , **int** dir , **long** tim){

...

}

而且有see\_direction、see\_time、see\_destination和see\_symbol来获取对象信息，所以Overview的抽象正确。

**2.对象有效性的论证**

我们先来看看request类的repOK：

/\*\*

\* 注释.......

\* **@REQUIRES**: None;

\* **@MODIFIES** : None;

\* **@EFFECTS** :

\* (0 <= symbol <= 1 && 1 <= destination <= 10 && 0 <= direction <= 1 && 2^32 - 1 >= time >= 0) ==> \result == true;

\* !(0 <= symbol <= 1 && 1 <= destination <= 10 && 0 <= direction <= 1 && 2^32 - 1 >= time >= 0) ==> \result == false;

\*/

**public** **boolean** repOK(){

**if**(0 <= symbol && symbol <= 1 && 1 <= destination && destination <= 10 && 0 <= direction && direction <= 1 && Element.***MAX\_INT*** >= time && time >= 0)

**return** **true**;

**else**

**return** **false**;

}

其中，如果symbol为电梯或者楼层，楼层数在1到10层间，方向为向上或向下，请求时间在0 到2^32 – 1之间，repOK就为true。

由于Overview中说到，只有Constructor会修改成员变量，所以仅对构造方法进行分析：

/\*\*

\* 注释.......

\* **@REQUIRES**: 0 <= sym <= 1 && 1 <= des <= 10 && 0 <= dir <= 1 && 2^32 - 1 >= tim >= 0;

\* **@MODIFIES** : this.symbol, this.destination, this.direction, this.time;

\* **@EFFECTS** :

\* \result == new Request(sym, des, dir, tim);

\*/

**public** Request(**int** sym , **int** des , **int** dir , **long** tim){

symbol = sym;

destination = des;

direction = dir;

time = tim;

}

该构造方法是在用参数传递进来的sym,des,dir,tim，修改symbol,destination,direction,time。在requires中我们可以看到，对sym的限制为0 <=sym <= 1,对des的限制为1 <= des <= 10,对dir的限制为1 <= dir <= 2，对tim的限制为0 <= tim <= 2^32 – 1。

所以，相应的有0 <=symbol <= 1, 1 <= destination <= 10, 1 <= direction <= 2， 0 <= time <= 2^32 – 1，不会使repOK为false。

而且，其余方法均没有改变this，所以Request类任意对象始终保持着对象的有效性。

**3.方法实现的正确性**

四种获取信息方法的规格和实现如下：

/\*\*

\* 注释.......

\* **@REQUIRES**: None;

\* **@MODIFIES** : None;

\* **@EFFECTS** :

\* \result == this.symbol;

\*/

**public** **int** see\_symbol(){

**return** symbol;

}

首先，判断规格是否正确有效。See\_symbol方法没有异常抛出，所以没有体现；前置条件为None，后置条件可判定；且未对this进行修改。综上，应判断出规格是合理的。

之后，该方法没有参数作为输入。所以根据前置条件和后置条件进行分类，在合理的情况下，仅有一种情况：

<\result =this.symbol> with <nothing>

所以无论如何，会返回this.symbol，符合规格要求。

/\*\*

\* 注释.......

\* **@REQUIRES**: None;

\* **@MODIFIES** : None;

\* **@EFFECTS** :

\* \result == this.destination;

\*/

**public** **int** see\_destination(){

**return** destination;

}

首先，判断规格是否正确有效。See\_destination方法没有异常抛出，所以没有体现；前置条件为None，后置条件可判定；且未对this进行修改。综上，应判断出规格是合理的。

之后，该方法没有参数作为输入。所以根据前置条件和后置条件进行分类，在合理的情况下，仅有一种情况：

<\result = this.destination> with <nothing>

所以无论如何，会返回this.destination，符合规格要求。

/\*\*

\* 注释.......

\* **@REQUIRES**: None;

\* **@MODIFIES** : None;

\* **@EFFECTS** :

\* \result == this.direction;

\*/

**public** **int** see\_direction(){

**return** direction;

}

首先，判断规格是否正确有效。see\_direction方法没有异常抛出，所以没有体现；前置条件为None，后置条件可判定；且未对this进行修改。综上，应判断出规格是合理的。

之后，该方法没有参数作为输入。所以根据前置条件和后置条件进行分类，在合理的情况下，仅有一种情况：

<\result = this.direction> with <nothing>

所以无论如何，会返回this.direction，符合规格要求。

/\*\*

\* 注释.......

\* **@REQUIRES**: None;

\* **@MODIFIES** : None;

\* **@EFFECTS** :

\* \result == this.time;

\*/

**public** **long** see\_time(){

**return** time;

}

首先，判断规格是否正确有效。See\_time方法没有异常抛出，所以没有体现；前置条件为None，后置条件可判定；且未对this进行修改。综上，应判断出规格是合理的。

之后，该方法没有参数作为输入。所以根据前置条件和后置条件进行分类，在合理的情况下，仅有一种情况：

<\result = this.time> with <nothing>

所以无论如何，会返回this.time，符合规格要求。

综上，Request类的实现正确。

七、requestList类

**1.Overview论证**

/\*\*

\* **@OVERVIEW**:

\* 这个类是请求队列类，通过ArrayList进行存储。

\* 可以执行增删取请求、与判空、返回length等操作。

\*/

这个类以ArrayList作为唯一成员变量储存数据，所有的方法均是使用库函数，所以类的抽象正确。

**2.对象的有效性论证**

/\*\*

\* 注释.......

\* **@REQUIRES**: None;

\* **@MODIFIES** : None;

\* **@EFFECTS** :

\* (list != null) ==> \result == true;

\* !(list != null) ==> \result == false;

\*/

**public** **boolean** repOK(){

**boolean** result = **false**;

**if**(*list* != **null**)

result = **true**;

**return** result;

}

所有方法均没有对list的引用对象进行修改，而且list的引用赋值是类创建时就赋值的，所以repOK恒为true。

**3.方法的实现正确性论证**

因为这个类的方法都是使用现成的库函数实现的，即使用ArrayList自带方法实现的，所以正确性可以保证。综上，RequestList类的实现正确。

八、scheduler类

**1. Overview论证**

/\*\*

\* **@OVERVIEW**:

\* 这个类是调度器类，继承前一次作业调度器类，存储有主请求，当前请求和各按钮亮灯情况等信息。

\* 可以根据请求队列RequestList内的请求和Elevator的状态，自动找到下一个主请求、可捎带请求，并因此控制电梯的运动以正确完成各个请求。

\*/

因为是继承前一次的调度器类，所以部分存储的数据信息不需要再重新论证，额外的新数据是主请求标号和捎带请求的标号，声明如下：

**public** **int** main\_request = -1;

**public** **int** picking\_request = -1;

抽象方法为根据请求队列和电梯状态，自动找下一个主请求和可捎带请求，自动完成寻找并完成下一个请求的方法的规格如下：

/\*\*

\* 注释.......

\* **@REQUIRES**: list != null && elevator != null;

\* **@MODIFIES** : this.main\_request, this.picking\_request, this.list, this.elevator;

\* **@EFFECTS** :

\* (main\_request == -1) ==> this.main\_request == search\_main\_one(list, elevator);

\* (main\_request == -1 && check\_duplicate(list, main\_request)) ==> (ignore(list , main\_request) == true && main\_request == -1);

\* (main\_request == -1 && !check\_duplicate(list, main\_request)) ==> change\_elevator\_dirction(list, elevator, main\_request) == true;

\* (main\_request != -1) ==> this.picking\_request = search\_picking\_one(list, elevator);

\* (main\_request != picking\_request && picking\_request != -1 && check\_duplicate(list, picking\_request)) ==> ignore(list, picking\_request) == true;

\* (main\_request != picking\_request && picking\_request != -1 && !check\_duplicate(list, picking\_request)) ==> excecute(list, elevator, picking\_request) == true;

\* !(main\_request != picking\_request && picking\_request != -1) ==> excecute(list, elevator, main\_request) == true;

\*/

**public** **void** next\_step(RequestList list , Elevator elevator){

...

}

这个方法是将其他小方法统一起来的实现。意思就是说，给一个请求队列和电梯作为参数，就可以自动根据电梯的当前状态，找到List中应该执行的那个请求，并完成。

综上，Overview中的抽象均可实现。

**2.对象有效性的论证**

我们先来看看Scheduler类的repOK：

/\*\*

\* 注释.......

\* **@REQUIRES**: None;

\* **@MODIFIES** : None;

\* **@EFFECTS** :

\* (super.repOK() && main\_request >= -1 && main\_request < RequestList.get\_Number() && picking\_request >= -1 && picking\_request <= RequestList.get\_Number()) ==> \result == true;

\* !(super.repOK() && main\_request >= -1 && main\_request < RequestList.get\_Number() && picking\_request >= -1 && picking\_request <= RequestList.get\_Number()) ==> \result == false;

\*/

**public** **boolean** repOK(){

**boolean** result = **false**;

**if**(**super**.repOK() && main\_request >= -1 && main\_request < RequestList.*get\_Number*() && picking\_request >= -1 && picking\_request <= RequestList.*get\_Number*())

result = **true**;

**return** result;

}

因为scheduler类继承了前一次的调度器类，所以如果父类的repOK为true，并且主请求为空或小于总请求数，捎带请求为空或小于总请求数，那么repOK就是true。

首先，回顾一下父类的repOK：

/\*\*

\* 注释.......

\* **@REQUIRES**: None;

\* **@MODIFIES** : None;

\* **@EFFECTS** :

\* (ele != null && floor\_up != null && floor\_down != null) ==> \result == true;

\* !(ele != null && floor\_up != null && floor\_down != null) ==> \result == false;

\*/

对应的，看一看scheduler类中的modifies，所有的方法最多只是修改ele等数组的内容，并未修改引用的对象，所以保证父类的repOK的正确性。

之后看所有修改main\_request和picking\_request的地方，通过modifies的查找，发现只有next\_step这个方法有对上述两个变量的修改：

/\*\*

\* 注释.......

\* **@REQUIRES**: list != null && elevator != null;

\* **@MODIFIES** : this.main\_request, this.picking\_request, this.list, this.elevator;

\* **@EFFECTS** :

\* (main\_request == -1) ==> this.main\_request == search\_main\_one(list, elevator);

\* (main\_request == -1 && check\_duplicate(list, main\_request)) ==> (ignore(list , main\_request) == true && main\_request == -1);

\* (main\_request == -1 && !check\_duplicate(list, main\_request)) ==> change\_elevator\_dirction(list, elevator, main\_request) == true;

\* (main\_request != -1) ==> this.picking\_request = search\_picking\_one(list, elevator);

\* (main\_request != picking\_request && picking\_request != -1 && check\_duplicate(list, picking\_request)) ==> ignore(list, picking\_request) == true;

\* (main\_request != picking\_request && picking\_request != -1 && !check\_duplicate(list, picking\_request)) ==> excecute(list, elevator, picking\_request) == true;

\* !(main\_request != picking\_request && picking\_request != -1) ==> excecute(list, elevator, main\_request) == true;

\*/

我们来分析一下这个方法，如果主请求为-1，也就是没有主请求，那么主请求main\_request = search\_main\_one(list, elevator)，找一个主请求。而查看search\_main\_one的规格可以发现：

/\*\*

\* 注释.......

\* **@REQUIRES**: list != null && elevator != null;

\* **@MODIFIES** : None;

\* **@EFFECTS** :

\* (\exist int i; 0 <= i < list.get\_Number();list.get\_Request(i).see\_symbol() == Element.ELE && elevator.get\_direction() == Element.ELEVATOR\_UP && elevator.get\_floor() < list.get\_Request(i).see\_destination() && (double)(elevator.get\_time() - Element.DOOR\_OPENING) > list.get\_Request(i).see\_time()) ==> mark == i;

\* (\exist int i; 0 <= i < list.get\_Number();list.get\_Request(i).see\_symbol() == Element.ELE && elevator.get\_direction() == Element.ELEVATOR\_DOWN && elevator.get\_floor() > list.get\_Request(i).see\_destination() && (double)(elevator.get\_time() - Element.DOOR\_OPENING) > list.get\_Request(i).see\_time()) ==> mark == i;

\* (mark == -1) ==> \result == 0;

\* mark != -1 ==> \result == mark;

\*/

Search\_main\_one的返回值为int mark，其中-1 <= mark <= list.length。所以此处执行过后，repOK仍然为true。

之后，对主请求进行重复性检测，如果发现是重复的，就直接删除，并且将main\_request = -1，仍然为true。

如果发现不是重复的，则根据主请求改变电梯的运动方向和系统时间，此时repOK仍然为true。第一个分支完毕。

如果有主请求，即main\_request != -1，那么找下一个捎带请求，picking\_request = search\_picking\_one(list, elevator)，这是我们再来看看search\_picking\_one的返回值：

/\*\*

\* 注释.......

\* **@REQUIRES**: list != null && elevator != null;

\* **@MODIFIES** : None;

\* **@EFFECTS** :

\* (\exist int i; 0 <= i < list.get\_Number();elevator.get\_direction() == Element.ELEVATOR\_UP && elevator.get\_floor() < list.get\_Request(i).see\_destination() && list.get\_Request(i).see\_destination() < list.get\_Request(main\_request).see\_destination() && ((double)(list.get\_Request(i).see\_destination() - elevator.get\_floor()) / 2 + elevator.get\_time()) > list.get\_Request(i).see\_time()) ==> \result == i;

\* (\exist int i; 0 <= i < list.get\_Number();elevator.get\_direction() == Element.ELEVATOR\_DOWN && elevator.get\_floor() > list.get\_Request(i).see\_destination() && list.get\_Request(i).see\_destination() > list.get\_Request(main\_request).see\_destination() && ((double)(elevator.get\_floor() - list.get\_Request(i).see\_destination()) / 2 + elevator.get\_time()) > list.get\_Request(i).see\_time()) ==> \result == i;

\* (\exist int i; 0 <= i < list.get\_Number();elevator.get\_direction() == Element.ELEVATOR\_UP && list.get\_Request(i).see\_direction() == Element.FLOOR\_UP && elevator.get\_floor() < list.get\_Request(i).see\_destination() && list.get\_Request(i).see\_destination() < list.get\_Request(main\_request).see\_destination() && list.get\_Request(main\_request).see\_destination() >= list.get\_Request(i).see\_destination() && ((double)(list.get\_Request(i).see\_destination() - elevator.get\_floor()) / 2 + elevator.get\_time()) > list.get\_Request(i).see\_time()) ==> \result = i;

\* (\exist int i; 0 <= i < list.get\_Number();elevator.get\_direction() == Element.ELEVATOR\_DOWN && list.get\_Request(i).see\_direction() == Element.FLOOR\_DOWN && elevator.get\_floor() > list.get\_Request(i).see\_destination() && list.get\_Request(i).see\_destination() > list.get\_Request(main\_request).see\_destination() && list.get\_Request(main\_request).see\_destination() <= list.get\_Request(i).see\_destination() && ((double)(elevator.get\_floor() - list.get\_Request(i).see\_destination()) / 2 + elevator.get\_time()) > list.get\_Request(i).see\_time()) ==> \result = i;

\* (else == true) ==> \result = -1;

\*/

通过阅读Effects发现，\result要不就是-1，要不就是i，再看看i的取值范围：\exist int i; 0 <= i < list.get\_Number()，所以，search\_picking\_one的返回值也是-1 <= \result < list.length，所以repOK仍然为true。

之后判断，如果捎带的请求存在并且不是主请求的话，那么先检查其是不是同质请求，如果是同质请求的话，删除，同时判断，如果此时主请求大于捎带请求，那么主请求-1：

**if**(main\_request > picking\_request)

main\_request--;

保证主请求指向的请求相对位置不变，而且数值依然小于等于list的长度。

如果不是同质请求的话，那么先执行，再删除，依然做如上的判断，保证主请求指向的请求相对位置不变，而且数值依然小于等于list的长度。

如果捎带请求存在并且是主请求的话，那么不用再检查同质了，因为选主请求的时候检查过，直接执行，并且将main\_request = -1，说明此时需要重新寻找主请求。

这个方法就结束了，其中几处会修改main\_request和picking\_request的地方均正确，所以综上，可以证明Scheduler类的对象有效性为真。

**3.方法实现的正确性**

该类一共8个方法，其中对构造方法、next\_step、search\_main\_one和search\_picking\_one方法，在上面论证有效性的时候已经论证分析过了，就不再赘述。

①查重方法：

/\*\*

\* 注释.......

\* **@REQUIRES**: list != null && 0 <= now\_request < list.length;

\* **@MODIFIES** : None;

\* **@EFFECTS** :

\* (list.get\_Request(now\_request).see\_symbol() == Element.ELE && ele[list.get\_Request(now\_request).see\_destination()] >= list.get\_Request(now\_request).see\_time()) ==> \result == true;

\* (list.get\_Request(now\_request).see\_symbol() == Element.FLOOR) && (floor\_up[list.get\_Request(now\_request).see\_destination()] >= list.get\_Request(now\_request).see\_time()) && (floor\_up[list.get\_Request(now\_request).see\_destination()] >= list.get\_Request(now\_request).see\_time()) ==> \result == true;

\* (list.get\_Request(now\_request).see\_symbol() == Element.FLOOR) && !(list.get\_Request(now\_request).see\_direction() == Element.FLOOR\_UP) && (floor\_down[list.get\_Request(now\_request).see\_destination()] >= list.get\_Request(now\_request).see\_time())) ==> \result == true;

\* (Else == true) ==> \result == false;

\*/

首先，判断规格是否正确有效。Check\_duplicate方法没有异常抛出，所以没有体现；前置条件与后置条件均可判定；且未对this进行修改。综上，应判断出规格是合理的。

之后，该方法有两个参数list和now\_request作为输入。所以根据前置条件和后置条件进行分类，在合理的情况下，有四种情况：

<\result = true> with <request.symbol == “Elevator” && ele[request.destination] >= request.time>

<\result = true> with < request.symbol == “Floor” && request.direction == UP && ele[request.destination] >= request.time >

<\result = true> with < request.symbol == “Floor” && request.direction == DOWN && ele[request.destination] >= request.time >

<\result = false> with <Other Situation>

情况A，比如Ele[5] = 10，此时来了一个（ER，5，8），那么进入第一个分支判断的第一个分支，又因为8<10，所以第二个判断成立，输出#SAME，并且返回true。

情况B，比如floor\_up[3] = 11，此时来了一个（FR，3，UP，7），那么进入第一个分支判断的第二个分支，又因为7<11，所以第二个判断成立，输出#SAME，并且返回true。

情况C，比如floor\_down [7] = 9，此时来了一个（FR，7，DOWN，3），那么进入第一个分支判断的第三个分支，又因为3<9，所以第二个判断成立，输出#SAME，并且返回true。

情况D，比如Ele[1] = 0，此时来了一个（ER，1，5），那么进入第一个分支判断的第一个分支，又因为5>0，所以第二个判断不成立，跳出判断，并返回false。

四种情况均按照规格正确实现，所以check\_duplicate实现正确。

②执行方法

/\*\*

\* 注释.......

\* **@REQUIRES**: list != null && elevator != null && 0 <= next\_request < list.length;

\* **@MODIFIES** : this.list, this.elevator, this.ele, this.floor\_up, this.floor\_down;

\* **@EFFECTS** :

\* (elevator.get\_direction() != Element.ELEVATOR\_STILL && elevator.get\_floor() < list.get\_Request(next\_request).see\_destination()) &&

\* elevator.set\_time(elevator.get\_time() + (double)(list.get\_Request(next\_request).see\_destination() - elevator.get\_floor())/2) &&

\* elevator.move\_to(list.get\_Request(next\_request).see\_destination()) ==> elevator.move\_to(list.get\_Request(next\_request).see\_destination())== true && elevator.set\_time(elevator.get\_time() + Element.DOOR\_OPENING);

\*(elevator.get\_direction() == Element.ELEVATOR\_STILL && elevator.get\_floor() < list.get\_Request(next\_request).see\_destination()) &&

\* elevator.set\_time(elevator.get\_time() + (double)(list.get\_Request(next\_request).see\_destination() - elevator.get\_floor())/2) &&

\* elevator.move\_to(list.get\_Request(next\_request).see\_destination()) ==> elevator.move\_to(list.get\_Request(next\_request).see\_destination())== true && elevator.set\_time(elevator.get\_time() + Element.DOOR\_OPENING);

\*/

这个方法是在选定了要执行某一个请求的情况下，根据电梯的自身状态，改变电梯使其运动以完成该请求，并给出相应输出。

首先，判断规格是否正确有效。excecute方法没有异常抛出，所以没有体现；前置条件与后置条件均可判定；modifies确定，对list、elevator、ele、floor\_up和floor\_down进行修改。综上，应判断出规格是合理的。

之后，该方法有两个参数list、elevator和next\_request作为输入。所以根据前置条件和后置条件进行分类，在合理的情况下，根据电梯运动方向和请求的类型可以分为有九种情况，但通过同类型合并可以简化为三种：

<根据被选中的下一个请求，将同层请求全部执行，并给出向上运动的输出> with <request.symbol == “Elevator” && request.destination >= elevator.destination || request.symbol == “Floor” && request.direction == “UP” && request.destination >= elevator.destination || request.symbol == “Floor” && request.direction == “DOWN” && request.destination >= elevator.destination >

<根据被选中的下一个请求，将同层请求全部执行，并给出向下运动的输出> with <request.symbol == “Elevator” && request.destination <= elevator.destination || request.symbol == “Floor” && request.direction == “UP” && request.destination <= elevator.destination || request.symbol == “Floor” && request.direction == “DOWN” && request.destination <= elevator.destination >

<根据被选中的下一个请求，将同层请求全部执行，并给出静止不动的输出> with <request.symbol == “Elevator” && request.destination == elevator.destination || request.symbol == “Floor” && request.direction == “UP” && request.destination == elevator.destination || request.symbol == “Floor” && request.direction == “DOWN” && request.destination == elevator.destination>

情况A：比如电梯在1层，请求为(ER，5，0)和(FR，5，UP，0)，那么先判断电梯状态，为向上。下一个请求是（ER，5,0），之后从list中从第一个请求开始，逐个判断，如果有同层请求的话全部执行。在这个样例中，先执行（ER，5，0），改变电梯此时的位置和时间，然后执行（FR，5，UP，0），找同层请求的同时需要对同质请求进行判断。找到（ER，5,0）之后，根据向上运动给出相应的输出，找到同层请求（FR，5，UP，0）的时候，同样根据向上运动给出相应的输出，所以说可以合并。

情况B：比如电梯在10层，请求为(ER，5，0)和(FR，5，DOWN，0)，那么先判断电梯状态，为向下。下一个请求是（ER，5,0），之后从list中从第一个请求开始，逐个判断，如果有同层请求的话全部执行。在这个样例中，先执行（ER，5，0），改变电梯此时的位置和时间，然后执行（FR，5，DOWNP，0），找同层请求的同时需要对同质请求进行判断。找到（ER，5，0）之后，根据向下运动给出相应的输出，找到同层请求（FR，5，DOWN，0）的时候，同样根据向下运动给出相应的输出，所以说可以合并。

情况C：比如电梯在5层，请求为(ER，5，0)和(FR，5，UP，0)，那么先判断电梯状态，为静止不动。下一个请求是（ER，5，0），之后从list中从第一个请求开始，逐个判断，如果有同层请求的话全部执行。在这个样例中，先执行（ER，5，0），改变电梯此时的位置和时间，然后执行（FR，5，UP，0），找同层请求的同时需要对同质请求进行判断。找到（ER，5,0）之后，根据静止不动给出相应的输出，找到同层请求（FR，5，UP，0）的时候，同样根据静止不动给出相应的输出，所以说可以合并。

综上，说明excecute方法实现是正确的。

③改变电梯方向方法

/\*\*

\* 注释.......

\* **@REQUIRES**: list != null && elevator != null && 0 <= next\_main\_request < list.length;

\* **@MODIFIES** : elevator;

\* **@EFFECTS** :

\* (list.get\_Request(next\_main\_request).see\_destination() > elevator.get\_floor()) ==> elevator.set\_dirction(Element.ELEVATOR\_UP);

\* (list.get\_Request(next\_main\_request).see\_destination() < elevator.get\_floor()) ==> elevator.set\_direction(Element.ELEVATOR\_DOWN);

\* list.get\_Request(next\_main\_request).see\_destination() == elevator.get\_floor()) ==> elevator.set\_dirction(Element.ELEVATOR\_STILL);

\*/

首先，判断规格是否正确有效。Change\_elevator\_direction方法没有异常抛出，所以没有体现；前置条件与后置条件均可判定；modifies确定，对elevator进行修改。综上，应判断出规格是合理的。

之后，该方法有三个参数list、elevator和next\_main\_request作为输入。所以根据前置条件和后置条件进行分类，在合理的情况下，可以分为三种情况：

< elevator.set\_dirction(Element.ELEVATOR\_UP) > with <request.destination \\> elevator.now\_floor>

< elevator.set\_dirction(Element.ELEVATOR\_DOWN) > with <request.destination \\< elevator.now\_floor>

< elevator.set\_dirction(Element.ELEVATOR\_STILL) > with <request.destination == elevator.now\_floor>

根据目标楼层和电梯当前楼层进行选择，改变电梯的运动方向为向上、向下或者静止，所以Change\_elevator\_direction的实现符合规格要求。

④忽略同质请求

/\*\*

\* 注释.......

\* **@REQUIRES**: list != null && 0 <= which\_one < list.length;

\* **@MODIFIES** : list;

\* **@EFFECTS** :

\* list.remove(which\_one) == true;

\*/

首先，判断规格是否正确有效。ignore方法没有异常抛出，所以没有体现；前置条件与后置条件均可判定；modifies确定，对list进行修改。综上，应判断出规格是合理的。

之后，该方法有两个参数list、which\_one作为输入。所以根据前置条件和后置条件进行分类，在合理的情况下，仅有一种情况：

< list.remove\_Request(which\_one)) > with <nothing>

只要满足requires，无条件执行删除list，所以ignore的实现符合规格要求。

综综上，本次作业论证完毕。