类实现正确性论证

Requeset_list 类

1、抽象对象得到了有效实现论证

```
/**
*@OVERVIEW: request_list is a class like arraylist, which has some basic function
*such as add, remove, set&get function to manage requests.
*/
```

2、对象有效性论证

首先本类的所有属性声明为 private，不包含继承。

private request[] list;

private int size;

(a) 构造方法

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

(b) 更改方法

Request_List 提供了三个状态更新方法：add(),remove(),top()，下面逐个进行论证：

•假设 add(request req)方法开始执行时，repOK 为 true。

1) Add 方法首先根据其私有属性 size 的大小判断是否是首条请求，如果是首条请求且不满足首条请求要求(FR,1,UP,0)则直接输出报错信息，没有改变 size 和 list，显然不改变 repOK 的取值；如果满足首条请求要求，则通过 size 获取请求队列尾的位置，将 req 加入队列，并将队列大小 size 计数加 1，由于初始化时队列大小为 2，size 初始为 0 所以 size 计数加 1 不会导致 repOK 为 false。
2) 倘若不是首条请求，则首先判断其与队尾的时间的大小关系，若小于则说明请求时间乱序，输出报错信息。没有改变 size 和 list，显然不改变 repOK 的取值；如果满足时间要求，将 req 加入队列，并将队列大小 size 计数加 1，并判断是否容量达到上限，若达到上限则新建一个大小满足要求的 request 数组并将队列前面的请求都复制到新队列，并改变 list 指向新队列。由于每次加入新请求都会进行队列大小检查和扩充，所以新加入一个 req 时不会导致 repOK 为 false。
3) 该方法没有返回值，所以结束方法执行时 repOK 不会改变，不违背表示不变式。

•假设 remove(int index)方法开始执行时，repOK 为 true。

1) 首先如果 index 不满足要求，例如小于 0 或者大于等于 size，则直接 return，方法结束，未改变任何表示对象，不会导致 repOK 为假；
2) 根据 index 的下标将所有位于下标之后的请求全部前移，过程中删除下标所代表的请求，最后将队列大小 size 计数减 1，remove 操作只会将 size 减小，所以不存在判断其大于队列大小的情况。
3) 该方法没有返回值，所以结束方法执行时 repOK 不会改变，不违背表示不变式。

•假设 top(int index)方法开始执行时，repOK 为 true。

1) index 不满足要求，例如小于 0 或者大于等于 size，则直接 return，方法结束，未改变任何表示对象，不会导致 repOK 为假；
2) 如果 index 满足要求，则新建临时对象 a 指向下标为 index 的请求，且从下标遍历到队列头，将所有请求后移，最后将队列头指向该临时对象，以达到将下标为 index 的请求“升级”到队列头部的目的。由于单纯的移动请求不会导致 size 的变化，所以不会导致 repOK 为假。
3) 该方法没有返回值，所以结束方法执行时 repOK 不会改变，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或为基础的 get/set 方法，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) Add(request req):

```
/**
*@REQUIRES:req!=null;req.time!=99999999;req.floor!=11;req.type!="NULL";type.direction!="NULL";
*@MODIFIES:this;this.size;system.out;
*@EFFECTS:
*          (the request is in time sort and right format)==>(list.contains(r)&&this.size ==\old(this.size)+1);
*          !(the request is in time sort and right format)==>(output error info);
*          (not enough space)==>(make a new list)
*/
```

根据上述过程规格，获得如下的划分：

<output error info>with<not in time sort || not qualify for the first request>

<add into the list ,size == \old size +1>with<in time sort && (qualify for the first request || not the first request) && \old size +1< list.length>

<add into the list ,size == \old size +1 and enlarge the request array>with< in time sort && qualify for the first request && \old size +1 == list.length>

√方法首先检查确认 req 是否为第一条请求，如果是且不满足格式，则输出错误信息。如果不是第一条请求但是不满足时间非递减要求，同样输出错误信息，满足

<output error info>with<not in time sort || not qualify for the first request>

√req 为第一条请求且满足要求时，将其加入队列，并更改 size 大小。当 req 不为第一条请求时，若其满足时间非递减要求，则加入队列，并更改 size 大小。且 size 小于 list.length 时不进行队列大小的扩充，在满足前置条件要求下，add 方法满足

<add into the list ,size == \old size +1>with<in time sort && (qualify for the first request || not the first request) && \old size +1< list.length>

√req 不为第一条请求时，若其满足时间非递减要求，则加入队列，并更改 size 大小。此时如果 size 等于 list.length 则意味着队列即将容量满，则新建一个大小满足要求的 request 数组并将队列前面的请求都复制到新队列，在满足前置条件要求下，add 方法满足

<add into the list ,size == \old size +1 and enlarge the request array>with< in time sort && qualify for the first request && \old size +1 == list.length>

(b) Remove(int index):

```
/**
 *@REQUIRES:index!=null;0<=index<\old(this.size);
 *@MODIFIES:this;this.size;
 *@EFFECTS:
 *          !list.contains(\old(list).get(index))
 *          this.size==\old(this.size)-1;
 */
```

根据上述过程规格，获得如下的划分:

<do nothing> with<index<0 || index >=size>

<remove the request with position index, size == \old size -1> with <0 <= index <size>

√方法首先确认 index 的合法性，如果不满足要求，直接 return 结束，满足

<do nothing> with<index<0 || index >=size>

√如果 index 合法，则根据 index 的下标将所有位于下标之后的请求全部前移，过程中删除下标所代表的请求，最后将队列大小 size 计数减 1，在满足前置条件要求下，remove 方法满足

<remove the request with position index, size == \old size -1> with <0 <= index <size>

(c) Top(int index):

```
/**
 *@REQUIRES:index!=null;0<=index<\old(this.size));
 *@MODIFIES:this;;
 *@EFFECTS:
 *          change the order of the list , and make a request to the top;
 */
```

根据上述过程规格，获得如下的划分：

<do nothing > with < index<0 || index >=size >

<top the request with position index > with <0 <= index <size >

√方法首先确认 index 的合法性，如果不满足要求，直接 return 结束，满足

<do nothing> with<index<0 || index >=size>

√如果 index 合法，则新建临时对象 a 指向下标为 index 的请求，且从下标遍历到队列头，将所有请求后移，最后将队列头指向该临时对象，以达到将下标为 index 的请求“升级”到队列头部的目的，在满足前置条件要求下，top 方法满足

<top the request with position index > with <0 <= index <size >

(d) repOK()

```
 /**
 * @Effects: \result==invariant(this);
 */
```

RepOK 的实现很显然，针对不同不符合程序要求的情况，比如私有对象为 null，队列大小为负，队列大小大于实际大小，时间乱序均会返回 false，其余情况返回 true。所以很明显 repOK 的实现是正确的。

综上所述，所有方法的事先都满足规格，从而可以推断，request_list 的实现是正确的，即满足规格要求。

Requeset 类

1、抽象对象得到了有效实现论证

```
/**
*@OVERVIEW: request is a class contains basic parameter of a request
*                 such as time, type and others, it has the function to check the format and remove repeat requests from list.
 */
```

2、对象有效性论证

首先本类的所有属性声明为 private，不包含继承。

```
private String str;
 private double time;
 private String type;
 private String direction;
 private int floor;
 private final String regEx1="^\\(FR,\\+?[0-9]{1,5},((UP)|(DOWN)),\\+?[0-9]{1,10} \\ ) $";
 private final String regEx2 = "^\\(ER,\\+?[0-9]{1,5},\\+?[0-9]{1,10}\\)$";
 private int order;
```

（a） 构造方法

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

（b） 更改方法

Request 提供了两个状态更新方法：check(),remove_repeat()，下面逐个进行论证：

•假设 check()方法开始执行时，repOK 为 true。

1) Check 方法根据其私有属性与两个 pattern 进行正则匹配，匹配过程中不会对私有属性进行改变，未改变任何表示对象，不会导致 repOK 为假；
2) 该方法返回值会匹配结果，倘若有匹配成功则返回 true，其余情况返回 false，结束方法执行时也没有改变私有属性，repOK 不会改变，不违背表示不变式。

•假设 remove_repeat(elevator e,request_list r,int flag)方法开始执行时，repOK 为 true。

1) 首先如果 flag 不满足要求，例如小于 0，则直接 return，方法结束，未改变任何表示对象，不会导致 repOK 为假；
2) 根据传入的 request 相关属性，将 request_list 中的同质请求去除，这个行为只调用这个类的相关 get 方法，不会对表示对象产生改变，自然不会改变 repOK 的返回

值。

3) 该方法没有返回值，所以结束方法执行时 repOK 不会改变，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或为基础的 get/set 方法，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) check():

```
/**
*@EFFECTS:
*        (this request matches patter 1||this request matches patter 1)==>\result==true;
*        \result==false;
*/
```

根据上述过程规格，获得如下的划分：

<\result == true>with<request matches pattern 1 || request matches pattern 2>

<\result == false>with<request does not matches pattern 1 && request does not matches pattern 2>

<\result == false, output error info>with<request is invalid>

√方法首先检查确认 str 是否为 null，如果是则输出错误信息,满足

<\result == false, output error info>with<request is invalid>

√ str 不为 null 且可以匹配上 p1 或 p2 时，即 m1.maches()==true 或 m2.maches()==true 时，代表其匹配上了两个正则表达式之一，返回 true，满足

<\result == true>with<request matches pattern 1 || request matches pattern 2>

√str 不能匹配上 p1 和 p2 时,即 m1.maches()==false 且 m2.maches()==false 时，代表其不能匹配上任意正则表达式，格式错误，返回 fasle，满足

<\result == false>with<request does not matches pattern 1 && request does not matches pattern 2>

(b) remove_repeat(elevator e,request_list r,int flag):

```
/**
*@REQUIRES: e!=null;r!=null;flag!=null;flag>=0;
*@MODIFIES: r;
```

*@EFFECTS:

*　　(\all int i=1;i<r.get_size()&&(in time range))==>(judge the requests in list)==>(remove the repeat requests);

*/

根据上述过程规格，获得如下的划分:

<do nothing> with<flag<0>

<remove the repeat request, change the size of request_list and output info> with <flag>=0>

√方法首先确认 flag 的合法性，如果不满足要求，直接 return 结束，满足

<do nothing> with<flag<0>

√如果 flag 合法，则根据 flag 设定查询时间的下界，即 finishtime 截止时间，在这个范围内判断是否是同质请求，通过调用 get 方法选出同质请求，根据下标调用 request_list 的 remove()方法进行去除，并且输出同质信息，满足

<remove the repeat request, change the size of request_list and output info> with <flag>=0>

(c) repOK()

/**

* @Effects: \result==invariant(this);

*/

RepOK 的实现很显然，针对不同不符合程序要求的情况，比如私有对象为 null，时间为负，楼层不为 1-10 层之一，请求类型不明，方向不明均会返回 false，其余情况返回 true。所以很明显 repOK 的实现是正确的。

综上所述，所有方法的事先都满足规格，从而可以推断，request 的实现是正确的，即满足规格要求。

floor 类

1、抽象对象得到了有效实现论证

```
/**
*@OVERVIEW: floor is a class that can make a request
*/
```

2、对象有效性论证

首先本类的所有属性声明为 private，不包含继承。

private String str;

private String[] element;

(a) 构造方法

floor 类提供了两个构造方法，floor(String str)和 floor()，它初始化全部的 rep，repOK 的运行结果显然返回 true。

(b) 更改方法

floor 提供了两个状态更新方法：check(),analysis()，下面逐个进行论证：

•假设 check()方法开始执行时，repOK 为 true。

1) Check 首先根据私有属性的字符串,将其分割为不同的属性存在 element 数组中，包括楼层，时间，方向，这个过程中只对私有对象进行分析而非改变，所以不会改变 repOK 的返回值。
2) 接下来通过判断楼层，方向，时间的合法性，分别返回 check 的结果，true 或者 false，返回值的决定来自于对 element 的判断，同样不会改变私有对象，所以也不会导致 repOK 返回值为 false。

•假设 analysis()方法开始执行时，repOK 为 true。

1) Analysis 方法首先创建一个新的 request 实例，不包括在本类的私有对象中，属于即用即 new 的对象，不会导致 repOK 返回值改变。
2) 由于 analysis 方法调用一定在 check 过后，即一定是先 check 方法中对 element 的各个元素进行了赋值后才对 request 对象的各个属性进行赋值，并没有改变私有对象，自然也不会改变 repOK 的返回值。
3) 本方法的返回值为即时构造的 request 对象，所以也不会导致 repOK 返回值为 false。

(c) 该类的其他几个方法的执行皆不改变对象状态或为基础的 get/set 方法，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) check():

```
/**
*@MODIFIES:this;this.element;
*@EFFECTS:
*          true==>(check the range of the time and set the value of element);
*          (floor out of range or time out of range)==>(\result==false);
*          \result==true;
*/
```

根据上述过程规格，获得如下的划分：

<\result == true>with<request is valid>

<\result == false>with<request has invalid floor/time/direction>

√首先 element 各个元素进行赋值，例如楼层，方向和时间大小，当楼层超过范围，例如小于 1，大于 10，或是 1 层 DOWN，10 层 UP，或是时间大于 int 可以表示的最大数字时，返回值为 false，满足

<\result == false>with<request has invalid floor/time/direction>

√在楼层符合范围，时间符合范围，并且方向没有特殊情况下，返回值为 true，满足

<\result == true>with<request is valid>

(b) request analysis():

```
/**
*@REQUIRES:element[1]!=null;element[2]!=null;
element[3]!=null;this.check()==true;
*@MODIFIES:this.p;
*@EFFECTS:
*          true==>(create a request, set its basic value and return it);
*/
```

根据上述过程规格，获得如下的划分:

<create a request, set basic parameters and return it> with<every situation>

√方法直接构造一个新的 request 对象，利用其 set 方法设定 request 的属性，然后直接返回。由于前置条件要求 element 各个数组元素不为 null，且 check 返回值为 true 的情况下才会调用 analysis 方法，所以在满足前置条件情况下，方法执行结果才会满足后置条件，其他情况均为不合法的调用，所以满足

<create a request, set basic parameters and return it> with<every situation>

（c）repOK()

```
/**
* @Effects: \result==invariant(this);
*/
```

RepOK 的实现很显然，针对不同不符合程序要求的情况，比如私有对象为 null 会返回 false，其余情况返回 true。所以很明显 repOK 的实现是正确的。

综上所述，所有方法的事先都满足规格，从而可以推断，floor 类的实现是正确的，即满足规格要求。

Elevator 类

1、抽象对象得到了有效实现论证

```
/**
*@OVERVIEW: elevator is a class that has some basic get&set function, we use it to simulate the movement of a elevator.
*/
```

(抽象函数论证)

2、对象有效性论证

首先本类的所有属性声明为 private，不包含继承。

```
private String str;
private double Time;
private int currentfloor;
private String direction;
private String[] element;
```

（a）构造方法

elevator 类提供了两个构造方法，elevator(String str)，和 elevator()它初始化全部的 rep，repOK 的运行结果显然返回 true。

(b) 更改方法

elevator 提供了四个状态更新方法：check(),analysis(),FindBest(request r,request_list rl,int index),run(request r)，下面逐个进行论证：

•假设 check()方法开始执行时，repOK 为 true。

1） Check 首先根据私有属性的字符串，将其分割为不同的属性存在 element 数组中，包括楼层，时间，这个过程中只对私有对象进行分析而非改变，所以不会改变 repOK 的返回值。

2） 接下来通过判断楼层，时间的合法性，分别返回 check 的结果，true 或者 false，返回值的决定来自于对 element 的判断，同样不会改变私有对象，所以也不会导致 repOK 返回值为 false

•假设 analysis()方法开始执行时，repOK 为 true。

1） Analysis 方法首先创建一个新的 request 实例，不包括在本类的私有对象中，属于即用即 new 的对象，不会导致 repOK 返回值改变。

2） 由于 analysis 方法调用一定在 check 过后，即一定是先 check 方法中对 element 的各个元素进行了赋值后才对 request 对象的各个属性进行赋值，并没有改变私有对象，自然也不会改变 repOK 的返回值。

3）本方法的返回值为即时构造的 request 对象，所以也不会导致 repOK 返回值为 false。

•假设 FindBest(request r,request_list rl,int index)方法开始执行时，repOK 为 true。

1） FindBest 方法首先判断队列中是否包含可捎带的请求，当队列大小为 1 时，不存在捎带，直接返回，没有改变私有对象，不会导致 repOK 返回值改变。

2） 根据电梯当前楼层和主请求所在楼层，设定当前的方向，这个过程不会将私有对象的值更改为任何非法的值，所以 repOK 返回值应当为 true，根据请求发出时间，计算到达时间，到达时间为下界，遍历队列中的所有请求，选择一个当前可捎带的最近的请求，通过 min 定义距离，best 表示这个请求的下标，最后返回这个 best 值，过程中只对请求进行了 get 方法和对私有对象的读操作，不会改变私有对象的属性，自然也就不会导致 repOK 返回值为 false。

3） 本方法的返回值为最优捎带的下标，所以也不会导致 repOK 的返回值为 false。

4） 另一 FindBest 方法虽然传参不同，但是根本实现和此方法相同，故不再重复论证。

•假设 run(request r)方法开始执行时，repOK 为 true。

1） run 方法首先通过当前楼层和要执行的请求楼层，更改自己的私有对象，但是不会更改为非法值，不会导致 repOK 返回值改变。

2） 然后通过判断时间优先级，计算出到达该楼层的时间，并更改相应时间，当前楼层和模拟开关门，所有私有对象的更改都依赖如 request 的属性正确，由于 request 类实现正确，所以可以认为此时的 repOK 也不会因此更改为非法值导致 repOK 返回值为 false。。

3） 本方法没有返回值，所以也不会导致 repOK 返回值为 false。

4） 另一 run 方法虽然传参不同，但是根本实现和此方法相同，故不再重复论证。

(c) 该类的其他几个方法的执行皆不改变对象状态或为基础的 get/set 方法，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) check():

```
/**
*@MODIFIES:this;this.element;
*@EFFECTS:
*          true==>(check the range of the time and set the value of element);
*          (floor out of range or time out of range)==>(\result==false);
*          \result==true;
*/
```

根据上述过程规格，获得如下的划分：

<\result == true>with<request is valid>

<\result == false>with<request has invalid floor/time >

√首先 element 各个元素进行赋值，例如楼层，方向和时间大小，当楼层超过范围，例如小于 1，大于 10，或是时间大于 int 可以表示的最大数字时，返回值为 false，满足

<\result == false>with<request has invalid floor/time >

√在楼层符合范围，时间符合范围，返回值为 true，满足

<\result == true>with<request is valid>

(b) request analysis():

```
/**
*@REQUIRES:element[1]!=null;element[2]!=null;this.check()==true;
*@MODIFIES:this.p;
*@EFFECTS:
*          true==>(create a request, set its basic value and return it);
*/
```

根据上述过程规格，获得如下的划分：

<create a request, set basic parameters and return it> with<every situation>

√方法直接构造一个新的 request 对象，利用其 set 方法设定 request 的属性，然后直接返回。由于前置条件要求 element 各个数组元素不为 null，且 check 返回值为 true 的情况下才会调用 analysis 方法，所以在满足前置条件情况下，方法执行结果才会满足后置条件，其他情况均为不合法的调用，所以满足

<create a request, set basic parameters and return it> with<every situation>

(c) FindBest(request r,request_list rl,int index):

```
/**
*@REQUIRES:r!=null;rl!=null;rl.size()>=1;index!=null;index>=0;index<rl.size();
*@EFFECTS:
*          (no shortcut)==>\result==999;
*          \result==(the index of the best shortcut);
*/
```

根据上述过程规格，获得如下的划分：

<\result == 999>with<no shortcut>

<\result == the index of the best shortcut >with<exist shortcut request>

√首先定义各个变量便于进行判断，包括最小楼层间距离 min，到达时间，结束时间，若请求队列大小为 1 或者遍历全部请求后不包含捎带请求，则过程中不会改变 best 的值，返回值 best 为初始的 999，代表没有捎带，满足

<\result == 999>with<no shortcut>

√通过计算出电梯运行的方向，电梯时间与请求发出时间比较，可以简单地计算出到达每个楼层的时间，通过判断方向是否满足捎带条件，以及是否在时间要求内，来更新最近捎带的请求下标和最小楼层间距离，遍历所有请求以寻找到最适当的捎带，作为下一次运行的主请求，并在方法最后返回其下标，满足

<\result == the index of the best shortcut >with<exist shortcut request>

(d) FindBest(request r,request_list rl,int index):

```
/**
*@REQUIRES:r!=null;
*@MODIFIES:this.direction;this.Time;this.currentfloor;
*@EFFECTS:
*        true==>(command a request and change the status of the elevator,such as time, current floor and direction);
*/
```

根据上述过程规格，获得如下的划分：

<command a request and update basic parameters>with<every situation>

√首先本方法为单纯的执行一条请求，即不去考虑请求的任何有效性，是否含有捎带等等，这一切都是通过别的方法提前考虑过的，比如 request 类的正确实现，比如在 run 之前一定已经通过 FindBest 选出了最佳捎带，本方法首先通过当前楼层和目标楼层更新方向信息，再通过时间先后计算出到达目标楼层的时间，更新时间和当前楼层，以达到模拟电梯的移动过程，最后进行信息的输出和开关门的模拟，方法结束，满足

<command a request and update basic parameters>with<every situation>

(e) repOK()

```
/**
* @Effects: \result==invariant(this);
*/
```

RepOK 的实现很显然，针对不同不符合程序要求的情况，比如私有对象为 null，时间为负，楼层不为 1-10 层之一均会返回 false，其余情况返回 true。所以很明显 repOK

的实现是正确的。

综上所述，所有方法的事先都满足规格，从而可以推断，elevator 的实现是正确的，即满足规格要求。

Scheduler 类

1、抽象对象得到了有效实现论证

```
/**
*@OVERVIEW: scheduler is a class that can decide the order to finish the request;
*/
```

2、对象有效性论证

首先本类的所有属性声明为 private，不包含继承。

```
private floor[] f_list;
private elevator e;
private request_list r_list;
private int order;
```

(a) 构造方法

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

(b) 更改方法

Scheduler 类提供了两个个状态更新方法：shcedule(),command()下面逐个进行论证：

•假设 schedule()方法开始执行时，repOK 为 true。

1） schedule 方法类似于 get 方法，返回私有对象的首个需要进行执行的请求，过程中进对私有对象进行查找而非改变，所以不会改变 repOK 的返回值。

2） 返回值为队列首请求，同样不会改变私有对象，所以也不会导致 repOK 返回值为 false。

•假设 command()方法开始执行时，repOK 为 true。

1） command 方法首先去除当前请求的所有同质请求，构造请求 p，调用私有对象的 FindBest 方法，然后在队列中寻找最佳捎带 p，将其提升为最先执行的请求，去除相应同质，然后再次寻找可以同时完成的捎带，由于同时最多可以有三个请求同时完成，则记录下这三个请求，并从队列中移除，这个过程中只涉及 request_list 类的操作，由于 request_list 类实现正确，所以这个过程不会导致 repOK 返回值改变。

2） 经过筛选，我们记录了主请求和捎带请求，此时调用电梯的 run 方法执行该主请求并完成相应的捎带请求，然后将没有完成的捎带升级为主请求，去除同质请求。再次调用 run 执行主请求，完成相应输出，每执行完一条请求便进行 remove 操作等，由于 elevator 类，request_list 类执行正确，这个过程不涉及自身方法的调用，所以也不会导致 repOK 返回值为 false。

3） 当首条请求和其相关的所有捎带完成后，其会再次调用自身直至所有请求完成，这个过程也不会导致 repOK 返回值改变。

(c) 该类的其他几个方法的执行皆不改变对象状态或为基础的 get/set 方法，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) schedule():

```
/**
*@REQUIRES:r_list!=null;r_list.size()>=1;
*@EFFECTS:
*          \result==this.r_list.get(0);
*/
```

根据上述过程规格，获得如下的划分：

<\result == the first request>with<every situation>

√由于这个方法的调用一定是在 size 大于等于 1 的情况下调用了，在满足前置条件的情况下，一定能够满足后置条件，返回值为队列的首条请求，满足

<\result == the first request>with<every situation>

(b) command():

```
/**
*@REQUIRES:r!=null;r_list!=null;r_list.size()>=1;e!=null;
*@MODIFIES:this;this.f_list;this.e;this.r_list;this.order;
*@EFFECTS:
*          true==>(command all the request, using the shortcut and other judgment, until all the requests are finished);
*/
```

根据上述过程规格，获得如下的划分：

<command a request and its shortcut> with<the request list is not empty>

<quit and do nothing> with <empty request list>

√方法通过确定主请求，去除相应同质请求，并同时完成所有的捎带，包括当前楼层和目标楼层之间的，以及目标楼层之外的捎带，升级为主请求并迭代完成相应操作。知道一条主请求相关的所有请求完成，满足

<command a request and its shortcut> with<the request list is not empty>

√在队列为空的情况下，该方法不会再次调用自身，退出方法，满足

<quit and do nothing> with <empty request list>

(c) repOK()

```
/**
* @Effects: \result==invariant(this);
*/
```

RepOK 的实现很显然，针对不同不符合程序要求的情况，比如私有对象为 null，或私有对象的 repOK 不满足为 true 均会返回 false，其余情况返回 true。所以很明显 repOK 的实现是正确的。

综上所述，所有方法的事先都满足规格，从而可以推断，scheduler 的实现是正确的，即满足规格要求。

InputHandler 类

1、抽象对象得到了有效实现论证

/**

*@OVERVIEW: InputHandler is a class that can read input from console and transfer them into request list.

*/

2、对象有效性论证

首先本类的所有属性声明为 private，不包含继承。

private Scanner s;

private request_list rl;

(a) 构造方法

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

(b) 更改方法

Scheduler 类提供了一个状态更新方法：read()下面进行论证：

•假设 read()方法开始执行时，repOK 为 true。

1） read 方法首先初始化各个内部对象，比如请求，以及初始化一个电梯对象，一个楼层对象来模拟两类请求发出来源，然后程序进入读取循环，这一部分不涉及对两个私有对象的修改，所以不会导致 repOK 返回值为 false。

2） 读取阶段，将每一个读取到的字符串初始化为一个请求，首先调用自身的 check 方法判断基本格式是否正确，然后根据其类型 FR/ER 设定楼层或者电梯再次进行更细致的格式检查，检查通过的发出请求并加入请求队列，不通过的忽略并输出错误信息，这个过程调用的分别是 elevator 类，floor 类，request 类和 request_list 类的方法，由于以上类实现正确，所以这一系列操作也不会导致 repOK 返回值为 false。

3） 当所有的请求读取完毕，对请求队列中的所有请求进行编号，调用 request 类和 request_list 类的方法，由于其实现正确，所以也不会导致 repOK 返回值的改变。

(c) 该类的其他几个方法的执行皆不改变对象状态或为基础的 get/set 方法，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) read():

/**

*@REQUIRES:s!=null;rl!=null;

*@MODIFIES:this;this.rl;

*@EFFECTS:

*　　true==>(read all the request, delete invalid ones and add other into request list in order);

*/

根据上述过程规格，获得如下的划分：

<delete invalid request>with<wrong format>

<add into the request list in order>with<right format>

√通过方法可以发现，其在将读取到的字符串赋给 request 后，调用 request 类的 check 方法进行正则匹配，筛掉基本格式不匹配的请求，然后模拟楼层或电梯发出请求，调用自身的 check 方法检查格式，比如楼层范围，1 层 DOWN，10 层 DOWN 等，将错误的再次筛除，并不进行 add 操作，满足

<delete invalid request>with<wrong format>

√对于其他的满足条件的方法，调用私有对象 request_list 的 add 方法，将其加入请求队列以便后面进行调度，并且在读取结束后进行编号，便于后面进行输出，满足

<add into the request list in order>with<right format>

（b）repOK()

/**

* @Effects: \result==invariant(this);

*/

RepOK 的实现很显然，针对不同不符合程序要求的情况，比如私有对象为 null 或私有对象的 repOK 不满足为 true 均会返回 false，其余情况返回 true。所以很明显 repOK 的实现是正确的。

综上所述，所有方法的事先都满足规格，从而可以推断，InputHandlerr 的实现是正确的，即满足规格要求。