类实现正确性的论证

一、 电梯类 Elevator

1. 抽象对象得到了有效实现论证

抽象对象为现实生活中的电梯，类属性中包含了电梯的当前位置、状态和执行的主请求，满足真实电梯的要求，方法中有包含改变当前请求、重置请求、重置状态和向上运动、向下运动方法，满足电梯运行的需要。

2. 对象有效性论证

a) 针对构造方法论证初始对象的 repOK 为真。

Elevator 提供了一个构造方法，代入 repOK 中显然值为真。

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

Elevator 提供了 changemain、resetstate、resetmain、moveup、movedown 五个对象状态变更方法。下面逐个论证。

- 假设 changemain(Request re)方法开始执行前，repOK 为 true。该方法没有改变 pos 属性，因此执行结束后 pos>0 且 pos<10，并且 state == STILL || state == UP || state == DOWN，所以，执行完 changemain 方法后，repOK 仍然为 true。
- 假设 resetmain()方法开始执行前，repOK 为 true。此方法没有改变 pos 和 state，因此执行完之后 repOK 仍然为 true
- 假设 resetstate()方法开始执行前，repOK 为 true。此方法没有改变 pos，且有 state == STILL，因此，执行结束后，repOK 仍然为 true。
- 假设 moveup()方法开始执行前，repOK 为 true。根据 REQUIRES，pos！=10，即有，pos<10，所以执行完 moveup()方法之后，pos<=10 且 state == UP，所以 repOK 为 true。
- 假设 movedown()方法开始执行前，repOK 为 true。根据 REQUIRES，pos！=1，即有，pos>1 以执行完 moveup()方法之后，pos>1 state == UP，所以 repOK 为 true。

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) Elevator()

```
/**
 * @REQUIRES: None;
 * @MODIFIES: \this
 * @EFFECTS: pos == 1 && state == STILL;
 */
```

根据以上规格，获得以下划分：

<pos == 1 && state == STILL> with <none>

显然，代码满足该条件。

b) getstate()

```
/**
 * @REQUIRES: None
 * @MODIFIES: None
 * @EFFECTS: \result == state;
 */
```

根据以上规格，获得以下划分：

<\result == state> whith <none>

显然满足。

c) getmainreq()

```
/**
 * @REQUIRES: None
 * @MODIFIES: None
 * @EFFECTS: \result == mainreq;
 */
```

根据以上规格，获得以下划分：

<\result == mainreq> whith <none>

显然满足。

d) getpos()

```
/**
 * @REQUIRES: None
 * @MODIFIES: None
 * @EFFECTS: \result == pos;
 */
```

根据以上规格，获得以下划分：

<\result == pos> with <none>

显然满足。

e) changemain(Request)

```
/**
 * @REQUIRES: re != null;
 * @MODIFIES: \this
 * @EFFECTS: mianreq == re && (re.floor > pos ==>
state == UP && re.floor < pos ==> state == DOWN
 *          && re.floor == pos ==> state == STILL)
 */
```

根据以上规格，获得以下划分：

<mianreq == re && state == UP> with <re.floor > pos>

<mianreq == re && state == STILL> with <re.floor == pos>

<mianreq == re && state == DOWN> with <re.floor < pos>

✓ re.floor > pos 通过检验。根据代码显然满足。

✓ <re.floor == pos>通过检验。根据代码显然满足。

✓ <re.floor < pos>通过检验。根据代码显然满足。

f) resetstate()

```
/**
 * @REQUIRES: None
 * @MODIFIES: \this
 * @EFFECTS: state == STILL;
 */
```

根据以上规格，获得以下划分：

<state == STILL> with <none>

显然满足。

g) resetmain()

```
/**
 * @REQUIRES: None
 * @MODIFIES: \this
 * @EFFECTS: mainreq == null;
 */
```

获得以下划分：

<mainreq == null> with <none>

显然满足。

h) moveup()

```
/**
 * @REQUIRES: \this.pos != 10;
 * @MODIFIES: \this
 * @EFFECTS: pos == \old(pos) + 1 && state == UP;
 */
```

获得以下划分：

<pos == \old(pos) + 1 && state == UP> with <none>

显然满足。

i) movedown()

```
/**
 * @REQUIRES: \this.pos != 1;
 * @MODIFIES: \this
 * @EFFECTS: pos == \old(pos - 1) && state == DOWN;
 */
```

获得以下划分：

<pos == \old(pos - 1) && state == DOWN> with <none>

显然满足。

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

二、 请求队列 Reqlist

1. 抽象对象得到了有效实现论证

   Reqlist 类保存了请求队列相关的属性并提供了操作方法，满足抽象。

2. 对象有效性论证

   a) 针对构造方法论证初始对象的 repOK 为真。

   Reqlist 类有一个构造函数，将其结果代入 repOK 中，显然为 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。

   Reqlist 有 2 个状态改变方法：addterm(Request) 和 remove(Request)，下面分别论证：

   - 假设 addterm(Request)开始之前对象的 repOK 为 true。则经过 addterm 方法后，显然有 list!=null && size = list.size()，因此 repOK 的值为 true。
   - 假设 remove(Request)开始之前对象热 repOK 为 true。则经过 remove 方法后，显然仍旧为 true。

   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) Reqlist()

```
/**
 * @REQUIRES: None
 * @MODIFIES: None
 * @EFFECTS: create a new object of Reqlist
 */
```

根据规格，获得以下划分：

<list != null && size == list.size> with <none>

显然成立。

b) getsize()

```
/**
 * @REQUIRES: None
 * @MODIFIES: None
 * @EFFECTS: \result == size
 */
```

根据规格，获得以下划分：

<\result == size> with <none>

显然成立。

c) get(int)

```
/**
 * @REQUIRES: None
 * @MODIFIES: None
 * @EFFECTS: \result == list.get(i)
 */
```

根据规格，获得以下划分：

<\result == list.get(i)> with <none>

显然成立。

d) addterm(Request)

```
/**
 * @REQUIRES: \old(list).contains(re) == false;
 * @MODIFIES: None
 * @EFFECTS: list.contains(re) == true && size ==
\old(size) + 1;
 */
```

根据规格，获得以下划分：

<list.contains(re) == true && size == \old(size) + 1>
with <none>

显然成立。

e) remove(Request)

```
/**
 * @REQUIRES: \old(list).contains(re) == true;
 * @MODIFIES: None
 * @EFFECTS: list.contains(re) == false && size ==
\old(size) - 1;
 */
```

根据规格，获得以下划分：

<list.contains(re) == false && size == \old(size) - 1>
with <none>

显然成立。

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

三、 请求调度类 Scheduler

1. 抽象对象得到了有效实现论证

   Scheduler 类的属性包含请求队列和时间，满足抽象的需求。

   注：为了测试和论证方便，删去了 Scheduler 类中关于 FCFS 调度算法的部分方法。

2. 对象有效性论证

   a) 针对构造方法论证初始对象的 repOK 为真。

      显然，根据构造函数可知初始对象的 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。

      Scheduler 类没有提供改变类属性的方法。

   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) Scheduler()

```
/**
 * @REQUIRES: None
 * @MODIFIES: \this
 * @EFFECTS: create a object of scheduler;
 */
```

      根据规格，获得以下划分：

<timer != null && reqs != null> with <none>

      显然成立。

   b) errhandler(int, String)

```
/**
 * @REQUIRES: (code == 0 || code == 1) && str !=
null;
 * @MODIFIES: None;
 * @EFFECTS: handle errors according of code
 */
```

      根据规格，获得以下划分：

      <print(REE0 + str)> with <code == 0, str>;

      <print(ERR1 + str)> with <code == 1, str>;

      显然成立。

c) getreqs()

```
/**
 * @REQUIRES: None
 * @MODIFIES: None
 * @EFFECTS: \result = \this.reqs;
 */
```

根据规格，获得以下划分：

<\result == reqs> with <none>

显然成立。

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

四、 ALS 调度类 ALS_Scheduler

1. 抽象对象得到了有效实现论证

ALS_Scheduler 类继承自 Scheduler 类，在 Scheduler 的基础上增加了 ALS 调度算法。满足抽象。

2. 对象有效性论证

a) 针对构造方法论证初始对象的 repOK 为真。

显然，构造函数生成的初始对象的 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。

ALS_Scheduler 类有三个方法会改变对象的状态，分别为 setmain(Elevator)、schedule(Elevator)、command(Elevator)，下面分别论证。

- 假设 setmain(Elevator)之前对象的 repOK 为 true。
  1) 如果 Elevator.getmainreq != null，则不会改变对象状态，因此 repOK 仍然为 true。
  2) 如果 Elevator.getmainreq == null。如果 sdreqs.getsize != 0，则将 sdreqs 中时间最早的请求设为 ele 的 mainreq，并将其从 reqs 中移除，这个过程并不会导致 reqs 变为 null，所以对象的 repOK 为 true；如果 sdreqs.getsize == 0 但 reqs.getsize != 0，则将 reqs 中的第一个请求设为 ele 的 mainreq，并将其从 reqs 中移除，但并不会导致 reqs 变为 null，所以对象的 repOK 为 true；如果 sdreqs.getsize == 0 并且 reqs.getsize == 0，将 ele 的 mainreq 重置为 null 并返回，不改变状态，因此对象的 repOK 为 true。
  3) 综上，执行完 setmain(Elevator)之后对象的 repOK 仍然为 true。
- 假设 schedule(Elevator)之前对象的 reoOK 为 true。
  1) 对 reqs 中的每个请求，如果它与主请求实质上是相同的请

求，则将其从 reqs 中移除并输出。该过程不会导致 reqs 或者 sdreqs 变为 null，所以 repOK 为 true。

2) 对 reqs 中的每个请求，如果它可以被主请求捎带，则将其移入 sdreqs 中，此过程不会导致 reqs 或者 sdreqs 变为 null，所以 repOK 为 true。

3) 综上，执行完 schedule(Elevator)后对象的 repOK 仍然为 true。

- 假设执行 command(Elevator)方法之前，对象的 repOK 为 true。

1) command 方法调用 setmain(Elevator)方法和 schedule(Elevator)方法，根据调度后的结果，控制电梯运行。这个过程只会修改 reqs 和 sdreqs，而不会将其重置为 0，因此 repOK 为 true。

2) 综上，执行完 command(Elevator)之后，对象的 repOK 为 true。

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) ALS_Scheduler()

```
/**
 * @REQUIRES: None
 * @MODIFIES: None
 * @EFFECTS: create a new object of ALS_Scheduler;
 */
```

根据规格，获得以下划分：

b) pickcheck(Elevator, Request)

```
/**
 * @REQUIRES: request.time < timer.gettime
 *
 * @MODIFIES: None
 *
 * @EFFECTS: \result == true ==> (request.getkind() ==
FR && request.getdir() == ele.getstate() &&
 *          (request.getdir() == Enumstate.UP &&
request.getfloor() <= ele.getmainreq().getfloor()
 *          && request.getfloor() >= ele.getpos()) ||
 *          (request.getdir() == Enumstate.DOWN &&
request.getfloor() >= ele.getmainreq().getfloor()
 *          && request.getfloor() <= ele.getpos()));
 *          otherwise, \result == false;
 */
```

根据规格，获得以下划分：

<\result == true> with <request.getkind() == FR &&
request.getdir() == ele.getstate() &&(request.getdir() ==
Enumstate.UP && request.getfloor() <=
ele.getmainreq().getfloor() && request.getfloor() >=
ele.getpos()) || (request.getdir() == Enumstate.DOWN &&
request.getfloor() >= ele.getmainreq().getfloor() &&
request.getfloor() <= ele.getpos())>
<\result == false> with <other input>

由代码的逻辑可知，以上划分可以满足。

c) setmain(Elevator)

```
/**
 * @REQUIRES: ele != null
 * @MODIFIES: ele, \this
 * @EFFECTS: set ele.mainreq to proper request
 */
```

根据规格，获得以下划分：

<none> with <ele.getmainreq() != null>
<ele.getmianreq() == \old(sdreqs[min]) &&
sdreqs.contains(ele.getmainreq()) == false> with
<sdreqs.getsize() != 0>
<ele.getmianreq() == \old(reqs[0]) &&
reqs.contains(ele.getmainreq()) == false> with
<sdreqs.getsize() == 0 && reqs.getsize != 0>
<ele.getmianreq() == null> with <sdreqs.getsize() ==
0 && reqs.getsize == 0>

程序先判断sdreqs.getsize()是否为0，若等于0再判断reqs.getsize()是否为0，并分别执行相应操作。因此可以满足划分。

d) schedule(Elevator)

```
/**
 * @REQUIRES: None
 * @MODIFIES: ele
 * @EFFECTS: set ele.mainreq to proper request
 */
```

根据规格，获得以下划分：

int i, 0 <= i < reqs.size, re = \old(reqs).get(i)
<reqs.contains(re) == false> with
<ele.getmainreq().same(re)>
<reqs.contains(re) == false && sdreqs.contains(re) == true>
with <re.gettime < timer && pickcheck(ele, re) == true>

由代码可知，满足。

e) command(Elevator)

```
/**
 * @REQUIRES: ele != null
 * @MODIFIES: this
 * @EFFECTS: chose a request to handle accordding to
ALS algorithm
 */
```

此方法是一个过程控制方法，无法划分。对每个已经别分配的请求，该方法控制电梯运行并输出相应的信息。有过程控制可知，该方法满足规格描述。

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