正确性论证报告

tags： 面向对象课程文档 Java

elevator

抽象对象得到有效实现论证

elevator 类的 overview 交代了电梯保存的运动方向和所处楼层信息，明确了抽象对象；类的 rep 能够通过抽象函数映射到相应的抽象对象

对象有效性论证

1. 构造方法
   elevator 提供了一个构造方法 elevator()，它会初始化所有的 rep，repOk 的运行结果显然返回 true
2. 对象状态更改方法：set_floor()
   - 假设 set_floor() 方法执行时，repOk() 为 true
   - set_floor() 方法会根据输入的 f 的大小和 floor 作比较，无论比较结果如何，direction 都会进行一遍赋值，direction 不会为 null，repOk 依然为真
   - 因此，该方法的执行不会导致 repOk 为假，不违背表示不变式
3. 对象查询方法：get_direction()
   - 假设 get_direction() 方法执行时，repOk() 为 true
   - get_direction() 返回 direction 字符串，返回后的修改不会影响direction本身，direction不变
   - 因此，该方法的执行不会导致 repOk 为假，不违背表示不变式
4. 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

5. 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

1. set_floor()方法

```
public void set_floor(int f){
    /*
      @MODIFIES: this
      @EFFECTS: (f>this.floor)==>(this.direction=="UP")&&(this.floor==f)
                (f<this.floor)==>(this.direction=="DOWN")&&(this.floor==f)
                (f==this.floor)==>(this.direction=="STILL")&&(this.floor==f)
      */
```

根据上述过程规格，可以划分为

- `<direction="UP" and floor=f> with <f > floor>`
- `<direction="DOWN" and floor=f> with <f < floor>`
- `<direction="STILL" and floor=f> with <f == floor>`

1）方法首先将 f 和 floor 进行了比较，然后对 direction 进行赋值
2）之后方法会将 f 赋值给 floor
3）综上满足了三个划分

2. get_floor()方法

```
public int get_floor(){
    /*
      @MODIFIES: None
      @EFFECTS: \result==this.floor
      */
```

方法没有输入，只是返回 floor 的值

3. get_direction()方法

```
public String get_direction(){
    /*
      @MODIFIES: None
      @EFFECTS: \result==this.direction
      */
```

方法没有输入，只是返回 direction 的值

4. toString()方法

```
public String toString(double t){
    /*
      @MODIFIES: None
      @EFFECTS: \result=="("+floor+","+direction+","+t+")"
      */
```

方法没有输入，只是返回 floor 和 direction 组成的一个字符串

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

request_list

抽象对象得到有效实现论证

request_list 类的 overview 交代了类保存的请求队列 Vector 信息，明确了抽象对象；类的 rep 能够通过抽象函数映射到相应的抽象对象

对象有效性论证

1. 构造方法
   request_list 提供了一个构造方法 request_list()，它会初始化所有的 rep，repOk 的运行结果显然返回 true
2. 对象状态更改方法：add_req(request req)，remove_req(int i)，remove_req(request req)
   2.1 add_req(request req) 方法
   - 假设 add_req(request req) 方法执行时，repOk() 为 true
   - add_req(request req) 方法会根据输入的 req 进行有效性的判断，如果 req 满足加入请求队列的条件，那么 req_list 中会加入该 req，否则 req_list 不变，无论如何，req_list 都不会为null，repOk 依然为真
   - 因此，该方法的执行不会导致 repOk 为假，不违背表示不变式

   2.2 remove_req(int i) 方法
   - 假设 remove_req(int i) 方法执行时，repOk() 为 true
   - remove_req(int i) 方法会根据输入的 i 进行有效性的判断，如果 i 有效，那么 req_list 会移除掉其中第 i+1 个元素，变化的只是 req_list 的长度，无论如何，req_list 都不会为null，repOk 依然为真
   - 因此，该方法的执行不会导致 repOk 为假，不违背表示不变式

   2.3 remove_req(request req) 方法
   - 假设 remove_req(request req) 方法执行时，repOk() 为 true
   - remove_req(request req) 方法会根据输入的 req 进行移除，如果 req_list 包含该 req ，那么 req_list 会移除掉它，否则不作改变，此处变化的只是 req_list 的长度，无论如何，req_list 都不会为null，repOk 依然为真
   - 因此，该方法的执行不会导致 repOk 为假，不违背表示不变式
3. 对象查询方法 left_req()，get_req(int i)
   3.1 left_req()
   - 假设 left_req() 方法执行时，repOk() 为 true
   - left_req() 返回 req_list 的长度，返回后的修改不会影响 req_list 本身，req_list 不变
   - 因此，该方法的执行不会导致 repOk 为假，不违背表示不变式

   3.2 get_req(int i)
   - 假设 get_req(int i) 方法执行时，repOk() 为 true
   - get_req(int i) 返回 req_list 中第 i+1 个元素，返回后的修改不会使得 req_list变为 null
   - 因此，该方法的执行不会导致 repOk 为假，不违背表示不变式
4. 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
5. 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

1. add_req(request req)

```
public void add_req(request req){
    /*@REQUIRES: req!=null;
      @MODIFIES: this
      @EFFECTS: (req.judge()&&((this.left_req==0&&req.req_t==0&&(format of req is
[FR,1,UP,0]))||(req.req_t>=(last req in \old(req_list)).req_t)))
                    ==> (req_list.contains(req)==true)&&(req_list.size ==
\old(req_list).size+1)
                (otherwise) ==> (print error information)
      */
```

根据上述过程规格，可以划分为

- `<do nothing> with <req==null>`
- `<print INVALID and req's information> with <req.judge()==false>`
- `<print INVALID and req's information> with <req.judge()==true&&req is the first request&&format of req is not [FR,1,UP,0]>`
- `<print INVALID and req's information> with <req.judge()==true&&req is not the first request&&req's time is less than request before>`
- `< add req to req\_list> with < req.judge()==true&&((req is the first request && format of req is [FR,1,UP,0])||(req is not the first request && req's time is not less than request before))>`
  1）方法先对 req 进行判断，如果 req 为 null 则直接返回，不作处理
  2）之后方法调用了 req 的 judge 方法，判断 req 的格式是否符合规范，如果不符合，req.judge() 返回 false，方法则会输出 INVALID 和 req 的信息，结束处理
  3）如果 req.judge() 返回 true，则方法会继续判断 req 是否为程序的第一条请求，即 req_list 此时长度是否为 0，如果是的话，方法会判断 req 的格式是否满足 [FR,1,UP,0]，如果满足，则会将 req 加入 req_list，否则方法会输出 INVALID 和 req 的信息，结束处理
  4）如果 req.judge() 返回 true，而且方法判断 req 不是程序的第一条请求，那么方法会继续判断该条请求和之前的请求相比，其时间是不是相等或增加的，如果是的话，则会将 req 加入 req_list，否则方法会输出 INVALID 和 req 的信息，结束处理
  5）综上满足了五个划分

2. remove_req(int i)

```
public void remove_req(int i){
    /*@REQUIRES: i>=0&&i<req_list.size
      @MODIFIES: this
      @EFFECTS: (req_list.contains(req_list[i])==false)&&(req_list.size ==
\old(req_list).size-1)
      */
```

根据上述过程规格，可以划分为

- `<do nothing> with <i<0||i>=req_list.size>`
- `<remove (i+1)th element from req\_list> with <i>=0&&i<req_list.size>`
  1）方法先对 i 的范围进行了判断，如果 i<0 或 i>=req_list.size，则直接返回，不作处理

2）之后方法调用了 Vector 的 remove(int i) 方法，移除掉 req_list 中的第 i+1 个元素
3）综上满足了两个划分

3. remove_req(request req)

```
public void remove_req(request req){
 /*@REQUIRES: req_list.contains(req)==true
   @MODIFIES: this
   @EFFECTS: (req_list.contains(req)==false)&&(req_list.size == \old(req_list).size-1)
   */
```

根据上述过程规格，可以划分为

- `<do nothing> with <req_list.contains(req)==false>`
- `<remove req from req\_list> with <req_list.contains(req)==true>`
  1）方法调用了 Vector 的 remove(Object o) 方法，对于不包含在 req_list 中的 req 不做处理，对于包含在 req_list 中的 req，则将其从 req_list 中移除
  2）综上满足了两个划分

4. left_req()

```
public int left_req(){
    /*
      @MODIFIES: None
      @EFFECTS: \result == req_list.size
      */
```

方法没有输入，只是返回 req_list 的长度

5. get_req(int i)

```
public request get_req(int i){
    /*@REQUIRES: i>=0&&i<req_list.size
      @MODIFIES: None
      @EFFECTS: (i<0||i>=req_list.size) ==> \result == null
                (i>=0&&i<req_list.size) ==> \result == req_list[i]
      */
```

根据上述过程规格，可以划分为

- `<\result==null> with <i<0||i>=req_list.size>`
- `<\result == req_list[i]> with <i>=0&&i<req_list.size>`
  1）方法首先判断了 i 的范围，如果 i<0 或 i>=req_list.size，那么方法会返回 null
  2）之后方法会会返回 req_list 第 i+1 个元素
  3）综上满足了两个划分

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

scheduler

抽象对象得到有效实现论证

scheduler 类的 overview 交代了类保存的电梯系统时间，电梯以及进行傻瓜调度时的请求队列迭代量信息，明确了抽象对象；类的 rep 能够通过抽象函数映射到相应的抽象对象

对象有效性论证

1. 构造方法
   scheduler 提供了一个构造方法 scheduler()，它会初始化所有的 rep，repOk 的运行结果显然返回 true
2. 对象状态更改方法：schedule(request_list req_list)，command(request_list req_list)
   2.1 schedule(request_list req_list) 方法
   - 假设 schedule() 方法执行时，repOk() 为 true
   - schedule() 方法会根据传入的请求队列对电梯进行调度，在调度的过程中会改变电梯的楼层和方向信息，但是不会改变电梯本身的值，无论如何，ele 不会为 null，repOk 依然为真
   - 因此，该方法的执行不会导致 repOk 为假，不违背表示不变式

   2.1 command(request_list req_list) 方法
   - 假设 command() 方法执行时，repOk() 为 true
   - command() 方法只是当请求迭代量小于请求队列长度时进行调用 schedule() 方法进行调度，由于 schedule() 方法不会改变 repOk，所以在本方法中 ele 也不会为 null，repOk 依然为真
   - 因此，该方法的执行不会导致 repOk 为假，不违背表示不变式
3. 对象查询方法：类中没有对象查询方法
4. 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
5. 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

1. schedule(request_list req_list) 方法

```
public void schedule(request_list req_list){
    /*@REQUIRES: req_list!=null
      @MODIFIES: this, req_list
      @EFFECTS: (request req == req_list[iterator]; req is not the same req with before;
(make ele move to destination) &&
                    (change ele's floor and direction) && (t change when moving finished)
&& (iterator == \old(iterator)+1)) &&
                (\all requset req; (req in req_list)&&(req is same with another req which
is responsed); req_list.contains(req)==false && req be ignored)
      */
```

   根据上述过程规格，可以划分为
   - `<do nothing> with <req_list == null>`
   - `<remove req_list[iterator] from req_list> with <req_list!=null && (\any int i;i>=0 && i<iterator && request req == req_list[i] && req's type, destination and direction are the same with req_list[iterator] && req[iterator].t<=req.finish_t)>`
   - `<response req_list[iterator] and move ele and iterator == \old(iterator)+1> with <req_list!=null && (\all int i;i>=0 && i<iterator && request req == req_list[i] && (req's type, destination and direction are not the same with req_list[iterator] || req[iterator].t>req.finish_t))>`

1）方法先对 req_list进行了判断，如果 req_list 为 null，则直接返回，不作处理
2）之后方法遍历 req_list 在 iterator 之前的请求，比较 req_list[iteartor] 和每个请求之间是否相同，如果类型，目标楼层和方向都相同，再比较 req_list[iteartor] 的时间和这些请求的完成时间，如果 req_list[iteartor] 的请求时间小于等于完成请求的时间，说明是相同请求，则把 req_list[iteartor] 从 req_list 中移除并返回
3）如果方法不是相同请求（要求形式和时间都满足），那么方法会响应该请求，调整系统时间，并把电梯移动到对应楼层，修改电梯的属性，同时将 iterator 增加 1
4）综上满足了三个划分

2. command(request_list req_list) 方法

```
public void command(request_list req_list){
    /*@REQUIRES: req_list!=null
      @MODIFIES: this, req_list
      @EFFECTS: response all reqs in req_list && iterator == req_list.size &&
                (\all requset req; (req in req_list)&&(req is same with another req which
is responsed); req_list.contains(req)==false && req be ignored)
      */
```

根据上述过程规格，可以划分为

- `<do nothing> with <req_list == null>`
- `<always excute function schedule(req_list)> with <iterator < req_list.size>`

1）方法先对 req_list进行了判断，如果 req_list 为 null，则直接返回，不作处理
2）之后方法执行一个循环，循环条件是 `iterator < req_list.left_req()`，当迭代量小于请求队列长度时，调用 schedule() 方法取出一条请求进行判断和调度，req_list != null，满足了 schedule() 的调度条件
4）综上满足了两个划分

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

als_scheduler

抽象对象得到有效实现论证

scheduler 类的 overview 交代了类保存的电梯系统时间，电梯以及父类的请求队列迭代量信息，同时还有进行捎带需要的主请求 主楼层 临时方向和捎带队列等信息，明确了抽象对象；类的 rep 能够通过抽象函数映射到相应的抽象对象

对象有效性论证

1. 构造方法
   als_scheduler 提供了一个构造方法 als_scheduler()，它调用了父类的构造函数，会初始化所有的 rep，repOk 的运行结果显然返回 true
2. 对象状态更改方法：schedule(request_list req_list)，command(request_list req_list)，Move2Des(request_list req_list)，ArriveAtDes(request_list req_list)，ResetMainReq(request_list req_list)
   2.1 Move2Des(request_list req_list) 方法
   - 假设 Move2Des() 方法执行时，repOk() 为 true
   - Move2Des() 方法会根据传入的请求队列和此时系统的主请求以及电梯的状态选择可以捎带的请求，对电梯进行调度，在调度的过程中会改变电梯的楼层和方向信息，但是不会改变电梯本身的值，无论如何，

ele 不会为 null，同时方法也会根据电梯的运动改变系统的时间，在这个过程中，along_req 会加入新元素，但是最终会清空，但是 along_req 不会变为 null，所以 repOk 依然为真
    - 因此，该方法的执行不会导致 repOk 为假，不违背表示不变式

2.2 ArriveAtDes(request_list req_list) 方法

- 假设 ArriveAtDes() 方法执行时，repOk() 为 true
- ArriveAtDes() 方法会根据传入的请求队列和此时系统的主请求以及电梯的状态选择到达目标楼层时可以捎带的请求，进行合适的响应，在响应过程中电梯不会发生改变，ele 不会为 null，同时方法也会根据电梯的运动改变系统的时间，在这个过程中，along_req 会加入新元素，但是最终会清空，但是 along_req 不会变为 null，所以 repOk 依然为真
- 因此，该方法的执行不会导致 repOk 为假，不违背表示不变式

2.3 ResetMainReq(request_list req_list) 方法

- 假设 ResetMainReq() 方法执行时，repOk() 为 true
- ResetMainReq() 方法会根据传入的请求队列和此时系统的主请求以及电梯的状态重置主请求，在响应过程中电梯不会发生改变，ele 不会为 null，同时方法也没有改变 e_direction 和 along_req，所以 repOk 依然为真
- 因此，该方法的执行不会导致 repOk 为假，不违背表示不变式

2.4 schedule(request_list req_list) 方法

- 假设 schedule() 方法执行时，repOk() 为 true
- schedule() 方法会根据电梯楼层和主请求目标楼层设置 e_direction，e_direction 不会为 null，之后会调用 Move2Des()，ArriveAtDes()，ResetMainReq() 方法，而这些方法也不会改变 reqOk，repOk 依然为真
- 因此，该方法的执行不会导致 repOk 为假，不违背表示不变式

2.5 command(request_list req_list) 方法

- 假设 command() 方法执行时，repOk() 为 true
- command() 方法只是当请求队列长度不为 0 时调用 schedule() 方法进行调度，由于schedule() 方法不会改变 repOk，所以在本方法中 ele，e_direction，along_req 也不会为 null，repOk 依然为真
- 因此，该方法的执行不会导致 repOk 为假，不违背表示不变式

3. 对象查询方法：类中没有对象查询方法
4. 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
5. 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

1. Move2Des(request_list req_list) 方法

```
public void Move2Des(request_list req_list){
    /*@REQUIRES: req_list!=null
      @MODIFIES: this, req_list
      @EFFECTS: (\all requset req; (req in req_list)&&(req can be picked up when move to
destination); req_list.contains(req)==false && req be response)&&
                (\all requset req; (req in req_list)&&(req is same with another req which
is responsed); req_list.contains(req)==false && req be ignored)&&
                (this.ele.floor == main_floor)&&(t change with the main_floor)
      */
```

根据上述过程规格，可以划分为

- `<do nothing> with <req_list == null>`
- `<remove req from req_list> with <req_list!=null && (\all request req; req in req\_list && req is same with another req which is responsed)>`
- `<response req and move ele to main_floor and remove req from req_list and t change with the main_floor> with <req_list!=null && (\all request req; req in req\_list && req can be picked up when move to destination)>`

  1）方法先对 req_list进行了判断，如果 req_list 为 null，则直接返回，不作处理

  2）之后方法在电梯每移动一层就遍历一遍请求队列中剩余的请求，查看有没有目标楼层在当前楼层以及请求时间和系统时间不冲突的请求，将请求加入捎带队列 along_req，在加入捎带队列的同时就会将请求和捎带队列中已经存在的请求进行比较，看是否是相同请求还是同质请求，同质请求需要加入捎带队列并响应，但是相同请求则会直接忽略，不会加入捎带队列并输出相同信息

  3）方法对于捎带队列中的请求会响应，移动电梯和设置系统时间并把请求从请求队列 req_list 中移除，对于相同请求也会把请求从请求队列 req_list 中移除

  4）综上满足了三个划分

2. ArriveAtDes(request_list req_list) 方法

```
public void ArriveAtDes(request_list req_list) {
    /*@REQUIRES: req_list!=null
      @MODIFIES: this, req_list
      @EFFECTS: (\all requset req; (req in req_list)&&(req can be picked up when arrived at destination); req_list.contains(req)==false && req be response)&&
                (\all requset req; (req in req_list)&&(req is same with another req which is responsed); req_list.contains(req)==false && req be ignored)&&
                (req_list.contains(main_req)=false)&&(t = \old(t)+1)
      */
```

根据上述过程规格，可以划分为

- `<do nothing> with <req_list == null>`
- `<remove req from req_list> with <req_list!=null && (\all request req; req in req\_list && req is same with another req which is responsed)>`
- `<response req and move ele to main_floor and remove req from req_list and t change with the main_floor> with <req_list!=null && (\all request req; req in req\_list && req can be picked up when arrived at destination)>`

  1）方法先对 req_list进行了判断，如果 req_list 为 null，则直接返回，不作处理

  2）之后方法在电梯到达主请求目标楼层后遍历一遍请求队列中剩余的请求，查看有没有目标楼层在当前楼层以及请求时间和系统时间不冲突的请求，将请求加入捎带队列 along_req，在加入捎带队列的同时就会将请求和捎带队列中已经存在的请求进行比较，看是否是相同请求还是同质请求，同质请求需要加入捎带队列并响应，但是相同请求则会直接忽略，不会加入捎带队列并输出相同信息

  3）方法对于捎带队列中的请求会响应，移动电梯和设置系统时间并把请求从请求队列 req_list 中移除，对于相同请求也会把请求从请求队列 req_list 中移除

  4）综上满足了三个划分

3. ResetMainReq(request_list req_list) 方法

```
public void ResetMainReq(request_list req_list) {
    /*@REQUIRES: req_list!=null
      @MODIFIES: this, req_list
      @EFFECTS: set main_req after ele arrive at destination
      */
```

根据上述过程规格，可以划分为

- `<do nothing> with <req_list == null>`
- `<main_req == req\_list[i]> with <req_list!=null&&(\all int i;i>=0 && i<req_list.size && request==req[i] && req.t<t-1 && req.type is ER && req.des\_floor match e\_direction)>`
- `<main_req == req\_list[0]> with <req_list!=null && (\any request req;!(req in req\_list && req can be picked up when arrived at destination))>`

  1）方法先对 req_list进行了判断，如果 req_list 为 null，则直接返回，不作处理

  2）之后方法遍历请求队列，判断有没有满足要求的 ER 类型的请求能够作为新的主请求，如果存在则设置为主请求

  3）如果不存在，就把请求队列中的剩余的第一个请求作为新的主请求

  4）综上满足了三个划分

4. schedule(request_list req_list) 方法

```
/*@REQUIRES: req_list!=null
      @MODIFIES: this, req_list
      @EFFECTS: (response main_req and other reqs which can be picked up moving to destionation) && (make ele move)
      */
```

根据上述过程规格，可以划分为

- `<do nothing> with <req_list == null>`
- `<e_direction == "UP" && Move2Des(req_list) && ArriveAtDes(req_list) && ResetMainReq(req_list)> with <req_list!=null && ele.get_floor()<main_floor>`
- `<e_direction == "DOWN" && Move2Des(req_list) && ArriveAtDes(req_list) && ResetMainReq(req_list)> with <req_list!=null && ele.get_floor()>main_floor>`
- `<e_direction == "STILL" && Move2Des(req_list) && ArriveAtDes(req_list) && ResetMainReq(req_list)> with <req_list!=null && ele.get_floor()==main_floor>`

  1）方法先对 req_list进行了判断，如果 req_list 为 null，则直接返回，不作处理

  2）之后方法判断了电梯此时的楼层和主请求楼层来设置 e_direction，满足了三种情况下的设置

  3）之后方法调用 Move2Des(req_list)，ArriveAtDes(req_list)，ResetMainReq(req_list) 方法，此时 req_list 不是 null

  4）综上满足了四个划分

5. command(request_list req_list) 方法

```
public void command(request_list req_list){
    /*@REQUIRES: req_list!=null
      @MODIFIES: this, req_list
      @EFFECTS: response all reqs in req_list&&(req_list.size == 0)
      */
```

根据上述过程规格，可以划分为

- `<do nothing> with <req_list == null>`
- `<do nothing> with <req_list.size == 0>`
- `<always excute function schedule(req_list)> with <req_list.size!=0>`

1）方法先对 req_list 进行了判断，如果 req_list 为 null，则直接返回，不作处理
2）之后方法会判断 req_list 的长度，如果长度为0，则不会有实际执行效果
2）之后方法执行一个循环，循环条件是 `req_list.left_req()!=0`，当请求队列长度不为0时，调用 schedule() 方法对主请求和捎带请求进行判断和调度，req_list != null，满足了 schedule() 的调度条件
4）综上满足了三个划分

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