正确性论证

1. 请求队列类（RequestQueue）

1.1 抽象对象有效实现论证

```
/*
 * OVERVIEW: 请求队列类，构建并维护了一个容量为 1000 的请求队列
 * 表示对象：ArrayList<Request> queue, int front, int rear, int count;
 * 抽象函数：AF(c) = {queue, front, rear, count} where c.queue == queue, c.front == front, c.rear == rear, c.count == count;
 * 不变式：(c.front >= 0) && (c.rear >= -1) && (c.count >= 0 && c.count <= 1000) && (c.queue != null);
 */
```

表示对象为 queue，front，rear，count，通过抽象函数映射为带有队头索引，队尾索引和队内成员数量的请求队列。

1.2 对象有效性论证

① RequestQueue()是构造方法，为 queue 新建了 1 个 ArrayList 对象，并将 front 初始化为 0，rear 初始化为-1，count 初始化为 0，这样类的所有对象都能保证其 repOK 为真，repOK 的运行结果显然返回 true，对象的初始状态满足不变式，即 repOK 为真。

② RequestQueue 提供了 2 个状态更新方法 enQueue，deQueue。

（a）假设 enQueue(Request r)方法开始执行时，repOK 为 true。

enQueue 方法在队列未满（count<1000）时将请求加入 queue 尾部，并将 rear 和 count 加 1，从而使 rear 大于-1，不会使 queue 等于 null，同时因为原 count 小于 1000，所以也不会使 count 加 1 后大于 1000，所以方法结束时 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。

（b）假设 deQueue()方法开始执行时，repOK 为 true。

deQueue 方法在队列非空（count>0）时将 queue 头部的元素返回，并将 front 加 1，count 减 1，从而使 front 大于 0，不会使 queue 等于 null，同时因为原 count 大于 0，所以也不会使 count 减 1 后小于 0，所以方法结束时 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 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。

④ 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。

1.3 方法实现正确性论证

① **构造方法 RequestQueue()**

```
/**
 * @REQUIRES: None;
 * @MODIFIES: None;
 * @EFFECTS: (this != null) && (this.front == 0) && (this.rear == -1) && (this.count == 0) && (this.queue != null);
```

```
        */
```

此方法的规格可以划分为：

<(this != null) && (this.front == 0) && (this.rear == -1) && (this.count == 0) && (this.queue != null)> with <any>

方法构建一个 RequestQueue 对象，使 this != null，并对其中的四个属性进行初始化，将 front 赋值为 0，rear 赋值为-1，count 赋值为 0，最后对 queue 赋值 new ArrayList<Request>()，使 queue!=null，因此满足<(this != null) && (this.front == 0) && (this.rear == -1) && (this.count == 0) && (this.queue != null)> with <any>。

② **repOK()**

```
        /**
         * @REQUIRES: None;
         * @MODIFIES: None;
         * @EFFECTS: ((front >= 0) && (rear >= -1) && (count >= 0 && count <= 1000) && (queue != null)) ==> (\result == true);
         * (!((front >= 0) && (rear >= -1) && (count >= 0 && count <= 1000) && (queue != null))) ==> (\result == false);
           */
```

此方法的规格可以划分为：

<\result==true> with <(front >= 0) && (rear >= -1) && (count >= 0 && count <= 1000) && (queue != null)>

<\result==false> with <!((front >= 0) && (rear >= -1) && (count >= 0 && count <= 1000) && (queue != null))>

由于方法直接返回了(front >= 0) && (rear >= -1) && (count >= 0 && count <= 1000) && (queue != null)，因此只有当(front >= 0)，(rear >= -1)，(count >= 0)，(count <= 1000)，(queue != null)这几个条件同时满足时，该表达式为真，方法返回 true，因此满足<\result=true> with <(front >= 0) && (rear >= -1) && (count >= 0 && count <= 1000) && (queue != null)>。

(front >= 0)，(rear >= -1)，(count >= 0)，(count <= 1000)，(queue != null)这几个条件只要有一个条件不满足时，(front >= 0) && (rear >= -1) && (count >= 0 && count <= 1000) && (queue != null)表达式为假，方法返回 false，因此满足<\result=false> with <!((front >= 0) && (rear >= -1) && (count >= 0 && count <= 1000) && (queue != null))>。

③ **enQueue(Request r)**

```
        /**
         * @REQUIRES: r != null;
         * @MODIFIES: this;
         * @EFFECTS: (!this.isFull()) ==> (this.rear == (\old(this.rear)+1)) && (this.count == \old(this.count)+1) && (this.queue.size() == \old(this.queue).size()+1) && (this.queue.get(this.rear) == r) && (this.result == true);
         * (this.isFull()) ==> (this.result == false);
         */
```

此方法的规格可以划分为：

<\result==false> with <r != null && this.isFull()>

<(this.rear == (\old(this.rear)+1)) && (this.count == \old(this.count)+1) && (this.queue.size() == \old(this.queue).size()+1) && (this.queue.get(this.rear) == r) && (this.result == true)> with <r != null && !this.isFull()>

显然，执行 enQueue 方法时满足 isFull 方法的前置条件（None）要求，isFull 方法的返回值将满足其后置条件。

isFull 方法返回 true 时表明队列已满，不能将 r 加入 queue，此时方法直接返回 false，满足<\result==false> with < this.isFull()>。

isFull 方法返回 false 时表明队列未满，此时将请求 r 加入 queue 尾部，使 queue.size()增大 1，并将 rear 和 count 加 1，从而使 rear 指向 r 在 queue 中的位置，最后方法返回 true，因此满足<(this.rear == (\old(this.rear)+1)) && (this.count == \old(this.count)+1) && (this.queue.size() == \old(this.queue).size()+1) && (this.queue.get(this.rear) == r) && (this.result == true)> with <!this.isFull()>。

④ **deQueue()**

```
/**
 * @REQUIRES: None;
 * @MODIFIES: this;
 * @EFFECTS: (!this.isEmpty()) ==> ((this.front == (\old(this.front)+1)) && (this.count == \old(this.count)-1) && (this.result == this.queue.get(\old(this.front))));
 *              (this.isEmpty()) ==> (this.result == null);
 */
```

此方法的规格可以划分为：

< this.result == null > with < this.isEmpty()>

<(this.front == (\old(this.front)+1)) && (this.count == \old(this.count)-1) && (this.result == this.queue.get(\old(this.front)))> with <!this.isEmpty()>

显然，执行 enQueue 方法时满足 isEmpty 方法的前置条件（None）要求，isEmpty 方法的返回值将满足其后置条件。

isEmpty 方法返回 true 时表明队列已空，不能将队头元素出队，此时方法直接返回 null，满足< this.result == null > with < this.isEmpty()>。

isEmpty 方法返回 false 时表明队列未空，此时将先取出 front 指向的队头元素，并将 front 加 1，count 减 1，最后方法返回取出的队头元素，因此满足<(this.front == (\old(this.front)+1)) && (this.count == \old(this.count)-1) && (this.result == this.queue.get(\old(this.front)))> with <!this.isEmpty()>。

⑤ **isEmpty()**

```
/**
 * @REQUIRES: None;
 * @MODIFIES: None;
 * @EFFECTS: (this.count == 0) ==> (this.result == true);
 *              (this.count != 0) ==> (this.result == false);
```

```
     */
```

此方法的规格可以划分为：

<\result==true> with < this.count == 0>

<\result==false> with < this.count != 0>

由于方法直接返回了 count == 0，因此只有当满足 count 为 0 时，该表达式为真，方法返回 true，因此满足<\result==true> with < this.count == 0>。

当满足 count 不为 0 时，该表达式为假，方法返回 false，因此满足<\result==false> with < this.count != 0>。

⑥ **isFull()**

```
    /**
     * @REQUIRES: None;
     * @MODIFIES: None;
     * @EFFECTS: (this.count == 1000) ==> (this.result == true);
     *              (this.count != 1000) ==> (this.result == false);
     */
```

此方法的规格可以划分为：

<\result==true> with < this.count == 1000>

<\result==false> with < this.count != 1000>

由于方法直接返回了 count == 1000，因此只有当满足 count 为 1000 时，该表达式为真，方法返回 true，因此满足<\result==true> with < this.count == 1000>。

当满足 count 不为 1000 时，该表达式为假，方法返回 false，因此满足<\result==false> with < this.count != 1000>。

⑦ 该类中的其他方法都为 get 和 set 型方法，根据指导书的要求可忽略其正确性论证。

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

2. 电梯类(Elevator)

2.1 抽象对象有效实现论证

```
    /*
     * OVERVIEW: 电梯类，负责保存，获取，更新电梯的属性信息。
     * 表示对象：int currentFloor, double currentTime, String currentStatus;
     * 抽象函数：AF(c) = {currentFloor, currentTime, currentStatus} where c.currentFloor == currentFloor, c.currentTime == currentTime, c.currentStatus == currentStatus;
     * 不变式：(c.currentFloor >= 1 && c.currentFloor <= 10) && (c.currentTime >=0) && (c.currentStatus != null && (c.currentStatus.equals("UP") || c.currentStatus.equals("DOWN") || c.currentStatus.equals("STILL")));
       */
```

表示对象为 currentFloor，currentTime，currentStatus，通过抽象函数映射为有当前所在楼

层、当前时间、当前运行状态的电梯。

2.2 对象有效性论证

① Elevator()是构造方法，构造一个初始在 1 楼、时间为 0、状态为 STILL 的电梯，从而使类的所有三个对象都能保证其 repOK 为真，repOK 的运行结果显然返回 true，对象的初始状态满足不变式，即 repOK 为真。

② Elevator 类提供了 3 个状态更新方法 setFloor，setTime，setStatus。

（a）假设 setFloor(int floor)方法开始执行时，repOK 为 true。

setFloor(int floor)方法是把 floor 赋值为 currentFloor，由于前置条件保证了 1<=floor<=10，因此方法结束时 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。

（b）假设 setTime(double time)方法开始执行时，repOK 为 true。

setTime(double time)方法是把 time 赋值为 currentTime，由于前置条件保证了 0<=time<=4394967295L，因此方法结束时 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。

（c）假设 setStatus(String status)方法开始执行时，repOK 为 true。

setStatus(String status)方法是把 status 赋值为 currentStatus，由于前置条件保证了 status != null && (status.equals("UP") || status.equals("DOWN") || status.equals("STILL"))，因此方法结束时 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 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。

④ 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。

2.3方法实现正确性论证

① **构造方法 Elevator()**

```
/**
 * @REQUIRES: None;
 * @MODIFIES: this;
 * @EFFECTS: (this != null) && (this.currentFloor == 1) && (this.currentTime == 0) && (this.count == 0) && (this.currentStatus.equals("STILL");
 */
```

此方法的规格可以划分为：

<(this!=null)&&(this.currentFloor==1)&&(this.currentTime==0)&& (this.currentStatus.equals("STILL")> with <any>

方法构建一个 Elevator 对象，使 this != null，并对其中的 3 个属性进行初始化，将 currentFloor 赋值为 1，currentTime 赋值为 0，currentStatus 赋值为"STILL"，因此满足<(this != null) && (this.front == 0) && (this.rear == -1) && (this.count == 0) && (this.queue != null)> with <any>。

② **repOK()**

```
/**
```

```
        * @REQUIRES: None;
        * @MODIFIES: None;
        * @EFFECTS: ((this.currentFloor >= 1 && this.currentFloor <= 10) && (this.current
Time >=0) && (this.currentStatus != null && (this.currentStatus.equals("UP") || this.current
Status.equals("DOWN") || this.currentStatus.equals("STILL")))) ==> (\result == true);
        * (!((this.currentFloor >= 1 && this.currentFloor <= 10) && (this.currentTime >=0)
&& (this.currentStatus != null && (this.currentStatus.equals("UP") || this.currentStatus.equa
ls("DOWN") || this.currentStatus.equals("STILL"))))) ==> (\result == false);
        */
```

此方法的规格可以划分为：

<\result==true> with <(this.currentFloor >= 1 && this.currentFloor <= 10) && (this.curre ntTime >=0) && (this.currentStatus != null && (this.currentStatus.equals("UP") || this.curre ntStatus.equals("DOWN") || this.currentStatus.equals("STILL")))>

<\result==false> with <!((this.currentFloor >= 1 && this.currentFloor <= 10) && (this.cur rentTime >=0) && (this.currentStatus != null && (this.currentStatus.equals("UP") || this.cur rentStatus.equals("DOWN") || this.currentStatus.equals("STILL"))))>

由于方法直接返回了(this.currentFloor >= 1 && this.currentFloor <= 10) && (this.current Time >=0) && (this.currentStatus != null && (this.currentStatus.equals("UP") || this.current Status.equals("DOWN") || this.currentStatus.equals("STILL")))，因此只有当(this.currentFloor > = 1 && this.currentFloor <= 10)，(this.currentTime >=0)，(this.currentStatus != null)，(this. currentStatus.equals("UP") || this.currentStatus.equals("DOWN") || this.currentStatus.equals ("STILL"))这几个条件同时满足时，该表达式为真，方法返回 true，因此满足<\result==true> with <(this.currentFloor >= 1 && this.currentFloor <= 10) && (this.currentTime >=0) && (this.currentStatus != null && (this.currentStatus.equals("UP") || this.currentStatus.equals(" DOWN") || this.currentStatus.equals("STILL")))>。

(this.currentFloor >= 1 && this.currentFloor <= 10)，(this.currentTime >=0)，(this.current Status != null)，(this.currentStatus.equals("UP") || this.currentStatus.equals("DOWN") || thi s.currentStatus.equals("STILL"))这几个条件只要有一个条件不满足时，该表达式为假，方法返回 false，因此满足<\result==false> with <!((this.currentFloor >= 1 && this.currentFloor <= 10) && (this.currentTime >=0) && (this.currentStatus != null && (this.currentStatus.equals ("UP") || this.currentStatus.equals("DOWN") || this.currentStatus.equals("STILL"))))>。

③ **toString(Request req)**

```
        /**
        * @REQUIRES: req != null;
        * @MODIFIES: None;
        * @EFFECTS: \result.equals("[" + req + "]/(" + this.currentFloor + "," + this.curren
tStatus + "," + this.doubleFormat.format(this.currentTime) + ")");
        */
```

此方法的规格可以划分为：

<\result.equals("[" + req + "]/(" + this.currentFloor + "," + this.currentStatus + "," + thi s.doubleFormat.format(this.currentTime) + ")")> with <req != null>

由于方法直接返回了"[" + req + "]/(" + this.currentFloor + "," + this.currentStatus + "," + this.doubleFormat.format(this.currentTime) + ")"，因此满足<\result.equals("[" + req + "]/(" + this.currentFloor + "," + this.currentStatus + "," + this.doubleFormat.format(this.currentTime) + ")")> with <req != null>。

④ 该类中的其他方法都为 get 和 set 型方法，根据指导书的要求可忽略其正确性论证。

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。

3. 请求类（Request）

3.1 抽象对象有效实现论证

```
/*
 * OVERVIEW: 请求类，负责记录请求的详细信息等;
 * 表示对象：int floor, long time, String direction, boolean isSame, boolean isExecuted;
 * 抽象函数：AF(c) = {floor, time, direction, isSame, isExecuted} where c.floor == floor, c.time == time, c.direction == direction, c.isSame == isSame, c.isExecuted == isExecuted;
 * 不变式：(c.floor >= 1 && c.floor <= 10) && (0<=c.time<=4294967295L) && (c.direction != null && (c.direction.equals("UP") || c.direction.equals("DOWN") || c.direction.equals("STILL")));
 */
```

表示对象为 floor, time, direction, isSame, isExecuted，通过抽象函数映射为带有楼层，时间方向，是否为同质是否执行等信息的请求。

3.2 对象有效性论证

① Request(int floor ,long time, String direction)是构造方法，将传入的 floor 赋值给 this. floor，传入的 time 赋值给 this. time，将传入的 direction 赋值给 this. direction，由于前置条件保证了(floor >= 1 && floor <= 10) && (0<=time<=4294967295L) && (direction != null && (direction.equals("UP") || direction.equals("DOWN") || direction.equals("ER")))，这样类的所有对象都能保证其 repOK 为真，repOK 的运行结果显然返回 true，对象的初始状态满足不变式。

② Request 类提供了 2 个状态更新方法 setSame，setExecuted。

（a）假设 setSame()方法开始执行时，repOK 为 true。

setSame 方法将 isSame 属性赋值为 true，不会改变不变式，所以方法结束时 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。

（b）假设 setExecuted()方法开始执行时，repOK 为 true。

setExecuted 方法将 isExecuted 属性赋值为 true，不会改变不变式，所以方法结束时 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 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。

④ 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。

3.3 方法实现正确性论证

① **构造方法 Request(int floor ,long time, String direction)**

```
/**
 * @REQUIRES: (floor >= 1 && floor <= 10) && (0<=time<=4294967295L) && (direction != null && (direction.equals("UP") || direction.equals("DOWN") || direction.equals("ER")));
 * @MODIFIES: this;
 * @EFFECTS: (this != null) && (this.floor == floor) && (this.time == time) && (this.direction == direction) && (isSame == false) && (isExecuted == false);
 */
```

此方法的规格可以划分为：

<(this != null) && (this.floor == floor) && (this.time == time) && (this.direction == direction) && (isSame == false) && (isExecuted == false)> with <(floor >= 1 && floor <= 10) && (0<=time<=4294967295L) && (direction != null && (direction.equals("UP") || direction.equals("DOWN") || direction.equals("ER")))>

方法构建一个 Request 对象，使 this != null，并对其中的 5 个属性进行初始化，在满足前置条件的前提下将 this.floor 赋值为 floor，将 this.time 赋值为 time，将 this. direction 赋值为 direction，最后将 isSame 和 isExecuted 赋值为 true，因此满足<(this != null) && (this.floor == floor) && (this.time == time) && (this.direction == direction) && (isSame == false) && (isExecuted == false)> with <(floor >= 1 && floor <= 10) && (0<=time<=4294967295L) && (direction != null && (direction.equals("UP") || direction.equals("DOWN") || direction.equals("ER")))>。

② **repOK()**

```
/**
 * @REQUIRES: None;
 * @MODIFIES: None;
 * @EFFECTS: ((floor >= 1 && floor <= 10) && (0<=time<=4294967295L) && (direction != null && (direction.equals("UP") || direction.equals("DOWN") || direction.equals("ER")))) ==> (\result == true);
 *              (!((floor >= 1 && floor <= 10) && (0<=time<=4294967295L) && (direction != null && (direction.equals("UP") || direction.equals("DOWN") || direction.equals("ER"))))) ==> (\result == false);
 */
```

此方法的规格可以划分为：

<\result==true> with <(floor >= 1 && floor <= 10) && (0<=time<=4294967295L) && (direction != null && (direction.equals("UP") || direction.equals("DOWN") || direction.equals("ER")))>

<\result==false> with <!((floor >= 1 && floor <= 10) && (0<=time<=4294967295L) && (direction != null && (direction.equals("UP") || direction.equals("DOWN") || direction.equals("ER"))))>

由于方法直接返回了(floor >= 1 && floor <= 10) && (0<=time<=4294967295L) && (direction != null && (direction.equals("UP") || direction.equals("DOWN") || direction.equals("ER")))，因此只有当(floor >= 1 && floor <= 10)，(0<=time<=4294967295L)，(direction != null && (direction.equals("UP") || direction.equals("DOWN") || direction.equals("ER")))这几个条件同时满足时，该表达式为真，方法返回 true，因此满足<\result==true> with <(floor >= 1 && floor <= 10) && (0<=time<=4294967295L) && (direction != null && (direction.equals("UP") || direction.equals("DOWN") || direction.equals("ER")))>。

这几个条件只要有一个条件不满足时，(floor >= 1 && floor <= 10) && (0<=time<=4294967295L) && (direction != null && (direction.equals("UP") || direction.equals("DOWN") || direction.equals("ER")))表达式为假，方法返回 false，因此满足<\result==false> with <!((floor >= 1 && floor <= 10) && (0<=time<=4294967295L) && (direction != null && (direction.equals("UP") || direction.equals("DOWN") || direction.equals("ER"))))>。

③ **toString()**

```
/**
 * @REQUIRES: None;
 * @MODIFIES: None;
 * @EFFECTS: "ER".equals(direction) ==> \result.equals("ER,"+floor+","+time);
 * !"ER".equals(direction) ==> \result.equals("FR,"+floor+","+direction+","+time);
 */
```

此方法的规格可以划分为：

<\result.equals("ER,"+floor+","+time)> with <"ER".equals(direction)>

<\result.equals("FR,"+floor+","+direction+","+time)> with <!"ER".equals(direction)>

方法首先调用 java 提供的标准库函数检查请求 direction 是否为"ER"，若是则直接返回字符串("ER,"+floor+","+time)，因此满足<\result.equals("ER,"+floor+","+time)> with <"ER".equals(direction)>。

若 direction 不为"ER"，则直接返回字符串("FR,"+floor+","+direction+","+time)，因此满足<\result.equals("FR,"+floor+","+direction+","+time)> with <!"ER".equals(direction)>。

④ 该类中的其他方法都为 get 和 set 型方法，根据指导书的要求可忽略其正确性论证。

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

4. 调度器类（Scheduler）

4.1 抽象对象有效实现论证

```
/*
```

```
        * OVERVIEW: 调度器类，负责执行请求，更新电梯状态，检查同质请求，可捎带请求等;
        * 表示对象：RequestQueue reqQueue, Elevator elevator;
        * 抽象函数：AF(c) = {reqQueue, elevator} where c.reqQueue == reqQueue, c.elevator == elevator;
        * 不变式：(c.reqQueue != null) && (c.elevator != null);
        */
```

表示对象为 elevator，reqQueue，通过抽象函数映射为有被调度电梯对象，被调度请求队列的调度器。

4.2 对象有效性论证

① Scheduler(RequestQueue requestQueue,Elevator elevator)是构造方法，将传入的电梯对象赋值给 this.elevator，传入的请求队列对象赋值给 this.reqQueue，由于前置条件保证了 requestQueue != null && elevator != null，这样类的所有对象都能保证其 repOK 为真，repOK 的运行结果显然返回 true，对象的初始状态满足不变式。

② RequestQueue 提供了 5 个状态更新方法 updateStatus，checkSameRequest，checkPiggybacking，runStill，runUpOrDown。

（a）假设 updateStatus(int currentFloor, int targetFloor)方法开始执行时，repOK 为 true。

该方法通过调用 elevator.setStatus(status)方法来更改电梯状态，因为电梯类的实现是正确的，执行 elevator.setStatus 方法时满足方法的前置条件要求，则方法的返回值将满足其后置条件，所以调用 elevator.setStatus 方法不会使 elevator 等于 null，所以方法结束时 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。

（b）假设 checkSameRequest(Request req, double finishTime)方法开始执行时，repOK 为 true。

该方法通过调用 request 类的 setSame()方法来更改请求队列中的请求属性，从而改变 reqQueue，因为请求类的实现是正确的，执行 request 类的 setSame()方法时满足方法的前置条件要求，则方法的返回值将满足其后置条件，所以调用 request 类的 setSame()方法不会使 reqQueue 变为 null，所以方法结束时 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。

（c）假设 checkPiggybacking(Request req)方法开始执行时，repOK 为 true。

该方法通过调用 request 类的 setExecuted()方法和本类中的 checkSameRequest 方法来更改请求队列中的请求属性，从而改变 reqQueue。因为请求类的实现是正确的，执行 request 类的 setExecuted()方法时满足方法的前置条件要求，则方法的返回值将满足其后置条件，所以调用 setExecuted()方法不会使 reqQueue 变为 null，同时已经论证 checkSameRequest 方法的执行也不会使不变式为假，所以调用 checkSameRequest 方法不会使不变式为假。所以方法结束时 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。

（d）假设 runStill(Request request)方法开始执行时，repOK 为 true。

该方法通过调用 elevator.setTime 方法来更改电梯时间，因为电梯类的实现是正确的，执行 elevator.setTime 方法时满足方法的前置条件要求，则方法的返回值将满足其后置条件，所以调用 elevator.setTime 方法不会使 elevator 等于 null。同理该方法调用 checkSameRequest 检查队列中的同质请求，已经论证 checkSameRequest 方法的执行不会

使不变式为假，所以方法结束时 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。

（e）假设 runUpOrDown(Request request)方法开始执行时，repOK 为 true。

该方法通过调用 elevator.setTime 和 elevator.setFloor 方法来更改电梯时间，因为电梯类的实现是正确的，执行 elevator.setTime 和 elevator.setFloor 方法时满足方法的前置条件要求，则方法的返回值将满足其后置条件，所以调用 elevator.setTime 和 elevator.setFloor 方法不会使 elevator 等于 null。同理该方法调用 checkPiggybacking(request)检查队列中的可捎带请求，已经论证 checkPiggybacking 方法的执行不会使不变式为假，所以方法结束时 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 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。

④ 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。

4.3 方法实现正确性论证

① **构造方法 Scheduler(RequestQueue requestQueue,Elevator elevator)**

```
/**
 * @REQUIRES: requestQueue != null && elevator != null;
 * @MODIFIES: this;
 * @EFFECTS: this != null && this.elevator == elevator && this.reqQueue == requestQueue;
 */
```

此方法的规格可以划分为：

<this != null && this.elevator == elevator && this.reqQueue == requestQueue> with <requestQueue != null && elevator != null>

方法构建一个 Scheduler 对象，使 this != null。在前置条件保证了 requestQueue != null && elevator != null 的情况下，方法将传入的电梯对象赋值给 this.elevator，传入的请求队列对象赋值给 this.reqQueue，从而满足< this != null && this.elevator == elevator && this.reqQueue == requestQueue > with < requestQueue != null && elevator != null >。

② **repOK()**

```
/**
 * @REQUIRES: None;
 * @MODIFIES: None;
 * @EFFECTS: ((reqQueue != null) && (elevator != null)) ==> (\result == true);
 *           (!((reqQueue != null) && (elevator != null)) ==> (\result == false);
 */
```

此方法的规格可以划分为：

<\result==true> with <(reqQueue != null) && (elevator != null)>

<\result==false> with <!((reqQueue != null) && (elevator != null)>

由于方法直接返回了(reqQueue != null) && (elevator != null)，因此只有当(reqQueue !=

null)，(elevator != null)这 2 个条件同时满足时，该表达式为真，方法返回 true，因此满足<\result==true> with <(reqQueue != null) && (elevator != null)>。

(reqQueue != null)，(elevator != null)这 2 个条件只要有一个条件不满足时，(reqQueue != null) && (elevator != null)表达式为假，方法返回 false，因此满足<\result==false> with <!((reqQueue != null) && (elevator != null)>。

③ **updateStatus(int currentFloor, int targetFloor)**

```
/**
 * @REQUIRES: 1<=currentFloor<=10 && 1<=targetFloor<=10;
 * @MODIFIES: this;
 * @EFFECTS: (targetFloor > currentFloor) ==> (this.elevator.currentStatus.equals("UP"));
 *           (targetFloor < currentFloor) ==> (this.elevator.currentStatus.equals("DOWN"));
 *           (targetFloor == currentFloor) ==> (this.elevator.currentStatus.equals("STILL"));
 */
```

此方法的规格可以划分为：

<this.elevator.currentStatus.equals("UP")> with <1<=currentFloor<targetFloor<=10>

<this.elevator.currentStatus.equals("DOWN")> with <1<=targetFloor<currentFloor<=10>

<this.elevator.currentStatus.equals("STILL")> with <1<=currentFloor==targetFloor<=10>

方法首先判断 targetFloor 和 currentFloor 的大小关系，targetFloor > currentFloor 时将 status 设置为"UP"，然后调用 elevator.setStatus(status)方法更改电梯状态，此时满足该方法的前置条件要求，方法的返回值将满足其后置条件，因此满足<this.elevator.currentStatus.equals("UP")> with <1<=currentFloor<targetFloor<=10>。

targetFloor < currentFloor 时将 status 设置为"DOWN"，然后调用 elevator.setStatus(status)方法更改电梯状态，此时满足该方法的前置条件要求，方法的返回值将满足其后置条件，因此满足<this.elevator.currentStatus.equals("DOWN")> with <1<=targetFloor<currentFloor<=10>。

targetFloor == currentFloor 时将 status 设置为"STILL"，然后调用 elevator.setStatus(status)方法更改电梯状态，此时满足该方法的前置条件要求，方法的返回值将满足其后置条件，因此满足<this.elevator.currentStatus.equals("STILL")> with <1<=currentFloor==targetFloor<=10>。

④ **runStill(Request request)**

```
/**
 * @REQUIRES: request != null && this.elevator.getFloor() == request.getFloor();
 * @MODIFIES: this;request;System.out;
 * @EFFECTS: (this.elevator.currentTime == \old(this.elevator.currentTime) + 1) &&
 *           (request.getIsExecuted() == true) && System.out 有请求执行结果输出;
 */
```

此方法的规格可以划分为：

<(this.elevator.currentTime == \old(this.elevator.currentTime) + 1) && (request.getIsExecuted() == true) && System.out 有请求执行结果输出> with < request != null && this.elevator.getFloor() == request.getFloor()>

在满足方法的前置条件的情况下，方法先调用 elevator.setTime(elevator.getTime() + 1)来将电梯的时间加 1，elevator.getTime() + 1 显然满足 elevator.setTime 的前置条件要求，elevator.setTime 方法的返回值将满足其后置条件。

然后调用 request.setExecuted()方法改变请求的 isExecuted 属性使其为 true，显然此时满足 request.setExecuted()的前置条件要求，request.setExecuted()方法的执行结果将满足其后置条件。

然后调用 elevator.toString(request)方法获得格式化字符串信息并在 System.out 输出，因为 request != null 已经得到保证，显然满足 elevator.toString 的前置条件要求，elevator.toString 方法的返回值将满足其后置条件。

最后方法调用 checkSameRequest(request, elevator.getTime())检查同质请求，因为此时满足 elevator.getTime 的前置条件要求，elevator.getTime 方法的返回值将满足其后置条件，且 request != null 满足 checkSameRequest 的前置条件，checkSameRequest 方法的执行结果将满足其后置条件。

因此，方法结束时满足<(this.elevator.currentTime == \old(this.elevator.currentTime) + 1) && (request.getIsExecuted() == true) && System.out 有请求执行结果输出> with < request != null && this.elevator.getFloor() == request.getFloor()>。

⑤ **runUpOrDown(Request request)**

```
/**
 * @REQUIRES: request != null && this.elevator.getFloor() != request.getFloor();
 * @MODIFIES: this;
 * @EFFECTS: (this.elevator.currentTime > \old(this.elevator.currentTime)) &&
 *              (this.elevator.getFloor() == request.getFloor());
 */
```

此方法的规格可以划分为：

<(this.elevator.currentTime > \old(this.elevator.currentTime)) && (this.elevator.getFloor() == request.getFloor())> with <request != null && this.elevator.getFloor() != request.getFloor()>

在满足方法的前置条件的情况下，方法通过调用 request.getFloor()，elevator.getFloor()，elevator.getStatus()，elevator.getFloor()等方法来获取请求和电梯的相关信息，因为这些方法的前置条件均为 None，此时显然得到满足，方法的返回值也将满足其后置条件。

当电梯运行状态为"UP"时，其所处楼层必然小于目标楼层，当然也小于 10，所以 elevator.setFloor(elevator.getFloor() + 1)执行时必然满足 elevator.setFloor 的前置条件，方法的执行结果也将满足其后置条件。

当电梯运行状态为"DOWN"时，其所处楼层必然大于目标楼层，当然也大于 1，所以 elevator.setFloor(elevator.getFloor() - 1)执行时必然满足 elevator.setFloor 的前置条件，方法的执行结果也将满足其后置条件。

elevator.setTime 的前置条件为 time>=0，所以执行 elevator.setTime(elevator.getTime() + 0.5)和 elevator.setTime(elevator.getTime() + 1)时满足 elevator.setTime 的前置条件，方法的执行结果也将满足其后置条件。

执行 checkPiggybacking(request)方法时由于 request != null 满足 checkPiggybacking 方法的前置条件，方法的执行结果也将满足其后置条件。

方法在 request.getFloor() != elevator.getFloor()时会一直循环更新电梯所处楼层和时间，直到电梯所处楼层等于目标楼层为止，且因为电梯运行需要花费时间，方法结束时电梯时间大于原电梯时间，因此满足<(this.elevator.currentTime > \old(this.elevator.currentTime)) && (this.elevator.getFloor() == request.getFloor())> with <request != null && this.elevator.getFloor() != request.getFloor()>。

⑥ **checkSameRequest(Request req, double finishTime)**

```
/**
 * @REQUIRES: req != null;
 * @MODIFIES: this;System.out;
 * @EFFECTS: (!reqQueue.isEmpty() && finishTime >= elevator.getTime()) ==> (\all int i;(i >= reqQueue.getFront() && i <= reqQueue.getRear() && reqQueue.get(i).getTime() <= finishTime && !reqQueue.get(i).getIsSame() && !reqQueue.get(i).getIsExecuted() && reqQueue.get(i).getDirection().equals(req.getDirection()) && reqQueue.get(i).getFloor() == req.getFloor());(reqQueue.get(i).getIsSame() && System.out 输出该同质请求));
 * (reqQueue.isEmpty() || finishTime < elevator.getTime()) ==> do nothing;
 */
```

此方法的规格可以划分为：

<(\all int i;(i >= reqQueue.getFront() && i <= reqQueue.getRear() && reqQueue.get(i).getTime() <= finishTime && !reqQueue.get(i).getIsSame() && !reqQueue.get(i).getIsExecuted() && reqQueue.get(i).getDirection().equals(req.getDirection()) && reqQueue.get(i).getFloor() == req.getFloor());(reqQueue.get(i).getIsSame() && System.out 输出该同质请求))> with <request != null && !reqQueue.isEmpty() && finishTime >= elevator.getTime()>

<do nothing> with <request != null && (reqQueue.isEmpty() || finishTime < elevator.getTime())>

方法首先调用 reqQueue.isEmpty()方法判断请求队列是否为空，然后判断 finishTime 是否小于 elevator.getTime()，因为 reqQueue.isEmpty()和 elevator.getTime()方法的前置条件均为 None，显然满足，所以方法的返回结果也满足其后置条件，reqQueue 为空或 finishTime < elevator.getTime()时将直接返回，因此满足<do nothing> with <request != null && (reqQueue.isEmpty() || finishTime < elevator.getTime())>。

reqQueue 不为空且 finishTime >= elevator.getTime()时，方法遍历请求队列中的所有请求，逐个判断其是否满足同质请求的判断条件，遍历和判断过程中所调用的方法均为前置条件为 None 的 get 类方法和 Java 提供的标准库方法，所以可以保证这些方法的返回结果满足其后置条件。

如果满足同质请求的判断条件则通过 setSame()方法将该请求的 isSame 属性设为 true，setSame 前置条件为 None，显然满足，所以执行结果也满足其后置条件，然后再在 System.out 输出该同质请求，因此满足<(\all int i;(i >= reqQueue.getFront() && i <= reqQueue.getRear() && reqQueue.get(i).getTime() <= finishTime && !reqQueue.get(i).getIsSame() && !reqQueue.get(i).getIsExecuted() && reqQueue.get(i).getDirection().equals(req.getDirection()) && reqQueue.get(i).getFloor() == req.getFloor());(reqQueue.get(i).getIsSame() && System.out 输出该同质请求))> with <request != null && !reqQueue.isEmpty() && finishTime >= elevator.getTime()>。

⑦ **checkPiggybacking(Request req)**

```
/**
 * @REQUIRES: req != null;
 * @MODIFIES: this;System.out;
 * @EFFECTS: (\all int i;(i >= reqQueue.getFront()-1 && i <= reqQueue.getRear() &
& reqQueue.get(i).getTime() < elevator.getTime() && !reqQueue.get(i).getIsSame() && !req
Queue.get(i).getIsExecuted() && reqQueue.get(i).getFloor() == elevator.getFloor() && (reqQu
eue.get(i).getDirection().equals("ER") || reqQueue.get(i) == req || reqQueue.get(i).getDirecti
on().equals(elevator.getStatus()));(reqQueue.get(i).getIsExecuted() && System.out 有提示信息
输出));
 * (\result == (\exist int i;(i >= reqQueue.getFront()-1 && i <= reqQueue.getRear()
&& reqQueue.get(i).getTime() < elevator.getTime() && !reqQueue.get(i).getIsSame() && !re
qQueue.get(i).getIsExecuted() && reqQueue.get(i).getFloor() == elevator.getFloor() && (reqQ
ueue.get(i).getDirection().equals("ER") || reqQueue.get(i) == req || reqQueue.get(i).getDirec
tion().equals(elevator.getStatus()))));
 */
```

此方法的规格可以划分为：

<(\all int i;(i >= reqQueue.getFront()-1 && i <= reqQueue.getRear() && reqQueue.get(i).getTime() < elevator.getTime() && !reqQueue.get(i).getIsSame() && !reqQueue.get(i).getIsExecuted() && reqQueue.get(i).getFloor() == elevator.getFloor() && (reqQueue.get(i).getDirection().equals("ER") || reqQueue.get(i) == req || reqQueue.get(i).getDirection().equals(elevator.getStatus()));(reqQueue.get(i).getIsExecuted() && System.out 有提示信息输出))> with <request != null>

<\result == true> with <request != null && (\exist int i;(i >= reqQueue.getFront()-1 && i <= reqQueue.getRear() && reqQueue.get(i).getTime() < elevator.getTime() && !reqQueue.get(i).getIsSame() && !reqQueue.get(i).getIsExecuted() && reqQueue.get(i).getFloor() == elevator.getFloor() && (reqQueue.get(i).getDirection().equals("ER") || reqQueue.get(i) == req || reqQueue.get(i).getDirection().equals(elevator.getStatus())))>

<\result == false> with <request != null && !(\exist int i;(i >= reqQueue.getFront()-1 && i <= reqQueue.getRear() && reqQueue.get(i).getTime() < elevator.getTime() && !reqQueue.get(i).getIsSame() && !reqQueue.get(i).getIsExecuted() && reqQueue.get(i).getFloor() == elevator.getFloor() && (reqQueue.get(i).getDirection().equals("ER") || reqQueue.get(i) == req || reqQueue.get(i).getDirection().equals(elevator.getStatus())))>

方法遍历请求队列中的所有请求，逐个判断其是否满足同质请求的条件，遍历和判断过程中所调用的方法均为前置条件为 None 的 get 类方法和 Java 提供的标准库方法，所以可以保证这些方法的返回结果满足其后置条件。

对所有的满足可捎带条件的请求，通过调用 setExecuted()方法，将该请求的 isExecuted 属性设为 true，setExecuted 方法前置条件为 None，显然满足，所以执行结果也满足其后置条件，然后在控制台输出该请求的相关信息，reqQueue.get(i)!=null 显然成立满足 elevator.toString 方法的前置条件，方法的返回值也满足其后置条件，最后再调用 checkSameRequest 检查该可捎带请求的同质请求，reqQueue.get(i)!=null 满足 checkSameRequest 方法的前置条件，方法的执行结果当然也满足其后置条件。所以最后方法结束时满足<(\all int i;(i >= reqQueue.getFront()-1 && i <= reqQueue.getRear() && reqQueue.get(i).getTime() < elevator.getTi

me() && !reqQueue.get(i).getIsSame() && !reqQueue.get(i).getIsExecuted() && reqQueue.get(i).getFloor() == elevator.getFloor() && (reqQueue.get(i).getDirection().equals("ER") || reqQueue.get(i) == req || reqQueue.get(i).getDirection().equals(elevator.getStatus()));(reqQueue.get(i).getIsExecuted() && System.out 有提示信息输出))> with <request != null>。

方法先将 flag 变量设为 false，然后遍历过程中只要检测出可捎带请求，就会将 flag 变量设为 true，最后返回 flag 变量的值，所以有可捎带请求时方法返回值为 true，满足<(\result == true> with <request != null && (\exist int i;(i >= reqQueue.getFront()-1 && i <= reqQueue.getRear() && reqQueue.get(i).getTime() < elevator.getTime() && !reqQueue.get(i).getIsSame() && !reqQueue.get(i).getIsExecuted() && reqQueue.get(i).getFloor() == elevator.getFloor() && (reqQueue.get(i).getDirection().equals("ER") || reqQueue.get(i) == req || reqQueue.get(i).getDirection().equals(elevator.getStatus())))>。

没有可捎带请求时方法返回值为 false，满足<(\result == false> with <request != null && !(\exist int i;(i >= reqQueue.getFront()-1 && i <= reqQueue.getRear() && reqQueue.get(i).getTime() < elevator.getTime() && !reqQueue.get(i).getIsSame() && !reqQueue.get(i).getIsExecuted() && reqQueue.get(i).getFloor() == elevator.getFloor() && (reqQueue.get(i).getDirection().equals("ER") || reqQueue.get(i) == req || reqQueue.get(i).getDirection().equals(elevator.getStatus())))>。

⑧ **changeMainRequest(Request req)**

```
/**
 * @REQUIRES: req != null;
 * @MODIFIES: None;
 * @EFFECTS: (\exist min int i;(i >= reqQueue.getFront() && i <= reqQueue.getRear() && reqQueue.get(i).getTime() < elevator.getTime() - 1 && !reqQueue.get(i).getIsSame() && !reqQueue.get(i).getIsExecuted() && reqQueue.get(i).getDirection().equals("ER") && (elevator.getStatus().equals("UP") && reqQueue.get(i).getFloor() > elevator.getFloor() || elevator.getStatus().equals("DOWN") && reqQueue.get(i).getFloor() < elevator.getFloor())) ==> (\result == this.reqQueue.get(i));
 * !(\exist int i;(i >= reqQueue.getFront() && i <= reqQueue.getRear() && reqQueue.get(i).getTime() < elevator.getTime() - 1 && !reqQueue.get(i).getIsSame() && !reqQueue.get(i).getIsExecuted() && reqQueue.get(i).getDirection().equals("ER") && (elevator.getStatus().equals("UP") && reqQueue.get(i).getFloor() > elevator.getFloor() || elevator.getStatus().equals("DOWN") && reqQueue.get(i).getFloor() < elevator.getFloor())) ==> (\result == req);
 */
```

此方法的规格可以划分为：

<\result == this.reqQueue.get(i)> with <request != null && (\exist min int i;(i >= reqQueue.getFront() && i <= reqQueue.getRear() && reqQueue.get(i).getTime() < elevator.getTime() - 1 && !reqQueue.get(i).getIsSame() && !reqQueue.get(i).getIsExecuted() && reqQueue.get(i).getDirection().equals("ER") && (elevator.getStatus().equals("UP") && reqQueue.get(i).getFloor() > elevator.getFloor() || elevator.getStatus().equals("DOWN") && reqQueue.get(i).getFloor() < elevator.getFloor()))>

<\result == req> with <request != null && !(\exist int i;(i >= reqQueue.getFront() && i <= reqQueue.getRear() && reqQueue.get(i).getTime() < elevator.getTime() - 1 && !reqQueue.get(i).getIsSame() && !reqQueue.get(i).getIsExecuted() && reqQueue.get(i).getDirection().e

quals("ER") && (elevator.getStatus().equals("UP") && reqQueue.get(i).getFloor() > elevator.getFloor() || elevator.getStatus().equals("DOWN") && reqQueue.get(i).getFloor() < elevator.getFloor()))))>

方法遍历请求队列中的所有请求，逐个判断其是否可以由可捎带的请求升级为主请求，遍历和判断过程中所调用方法的均为前置条件为 None 的 get 类方法和 Java 提供的标准库方法，所以可以保证这些方法的返回结果满足其后置条件。

一旦发现有请求满足上述升级条件，则直接返回该请求，所以满足<\result == this.reqQueue.get(i)> with <request != null && (\exist min int i;(i >= reqQueue.getFront() && i <= reqQueue.getRear() && reqQueue.get(i).getTime() < elevator.getTime() - 1 && !reqQueue.get(i).getIsSame() && !reqQueue.get(i).getIsExecuted() && reqQueue.get(i).getDirection().equals("ER") && (elevator.getStatus().equals("UP") && reqQueue.get(i).getFloor() > elevator.getFloor() || elevator.getStatus().equals("DOWN") && reqQueue.get(i).getFloor() < elevator.getFloor()))))>

如果遍历完整个请求对列之后仍未发现有可升级为主请求的可捎带请求，则直接将参数中的请求返回，所以满足<\result == req> with <request != null && !(\exist int i;(i >= reqQueue.getFront() && i <= reqQueue.getRear() && reqQueue.get(i).getTime() < elevator.getTime() - 1 && !reqQueue.get(i).getIsSame() && !reqQueue.get(i).getIsExecuted() && reqQueue.get(i).getDirection().equals("ER") && (elevator.getStatus().equals("UP") && reqQueue.get(i).getFloor() > elevator.getFloor() || elevator.getStatus().equals("DOWN") && reqQueue.get(i).getFloor() < elevator.getFloor()))))>

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。

5. Main 类

5.1 抽象对象有效实现论证

```
/*
 * OVERVIEW: Main 类，负责处理输入请求，检查请求的有效性，以及控制调度器的正常工作等;
 * 表示对象：Boolean isFirst, long lastTime;
 * 抽象函数：AF(c) = {isFirst, lastTime} where c.isFirst == isFirst, c.lastTime == lastTime;
 * 不变式：(0<=c.lastTime<=4294967295L);
 */
```

表示对象为 isFirst, lastTime，通过抽象函数映射为带有是否可能是第一条有效请求，上一个请求的时间等信息的输入输出处理类。

5.2 对象有效性论证

① 该类无自定义的构造方法，属性 lastTime 在定义时被初始化为 0，repOK 为真，执行类的默认构造方法后 lastTime 不变，repOK 依然为真，所以对象的初始状态满足不变式。

② Main 类提供了 1 个与 lastTime 有关的状态更新方法 checkFormat。

（a）假设 checkFormat(int floor, long time, String direction)方法开始执行时，repOK 为 true。

checkFormat 方法只有在 time>=0 且 time<=4294967295L 的前提下时才有可能把 time 的值赋给 lastTime ，time<0 或 time>4294967295L 时方法直接返回 false，所以方法结束时 repOK 一定为真。

因此，该方法的执行不会导致 repOK 为假，不违背表示不变式。

③ 该类的其他几个方法的执行皆不改变 lastTime，因此这些方法执行前和执行后的 repOK 都为 true。

④ 综上，对该类任意方法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。

5.3 方法实现正确性论证

① **repOK()**

```
/**
 * @REQUIRES: None;
 * @MODIFIES: None;
 * @EFFECTS: ((lastTime >=0) && (lastTime<=4294967295L)) ==> (\result == true);
 * (!((lastTime >=0) && (lastTime<=4294967295L))) ==> (\result == false);
 */
```

此方法的规格可以划分为：

<\result==true> with <(lastTime >=0) && (lastTime<=4294967295L)>

<\result==false> with <!((lastTime >=0) && (lastTime<=4294967295L))>

由于方法直接返回了(lastTime >=0) && (lastTime<=4294967295L)，因此只有当(lastTime >=0)，(lastTime<=4294967295L)这 2 个条件同时满足时，该表达式为真，方法返回 true，因此满足<\result==true> with <(lastTime >=0) && (lastTime<=4294967295L)>。

(lastTime >=0)，(lastTime<=4294967295L)这 2 个条件只要有一个条件不满足时，(lastTime >=0) && (lastTime<=4294967295L)表达式为假，方法返回 false，因此满足<\result==false> with <!((lastTime >=0) && (lastTime<=4294967295L))>。

② **checkFormat(int floor, long time, String direction)**

```
/**
 * @REQUIRES: (direction != null && (direction.equals("UP") || direction.equals("DOWN") || direction.equals("ER")));
 * @MODIFIES: isFirst;lastTime;
 * @EFFECTS: (!(floor<1 || floor>10) && !(floor==1 && "DOWN".equals(direction) || floor==10 && "UP".equals(direction)) && !(time<0 || time>4294967295L) && isFirst==true && (floor==1 && "UP".equals(direction) && time==0)) ==> (\result == true && isFirst==false);
 *  (!(floor<1 || floor>10) && !(floor==1 && "DOWN".equals(direction) || floor==10 && "UP".equals(direction)) && !(time<0 || time>4294967295L) && isFirst==false && time >= lastTime) ==> (\result == true && lastTime == time);
 *  ((floor<1 || floor>10) || (floor==1 && "DOWN".equals(direction) || floor==10
```

```
&& "UP".equals(direction)) || (time<0 || time>4294967295L) || isFirst==true && !(floor==
1 && "UP".equals(direction) && time==0) || time < lastTime) ==> (\result == false);
        */
```

此方法的规格可以划分为：

<\result == true && isFirst==false> with <(direction != null && (direction.equals("UP") || direction.equals("DOWN") || direction.equals("ER")) && !(floor<1 || floor>10) && !(floor==1 && "DOWN".equals(direction) || floor==10 && "UP".equals(direction)) && !(time<0 || time>4294967295L) && isFirst==true && (floor==1 && "UP".equals(direction) && time==0)>

<\result == true && lastTime == time> with <(direction != null && (direction.equals("UP") || direction.equals("DOWN") || direction.equals("ER")) && !(floor<1 || floor>10) && !(floor==1 && "DOWN".equals(direction) || floor==10 && "UP".equals(direction)) && !(time<0 || time>4294967295L) && isFirst==false && time >= lastTime>

<\result == false> with <(direction != null && (direction.equals("UP") || direction.equals("DOWN") || direction.equals("ER")) && (floor<1 || floor>10) || (floor==1 && "DOWN".equals(direction) || floor==10 && "UP".equals(direction)) || (time<0 || time>4294967295L) || isFirst==true && !(floor==1 && "UP".equals(direction) && time==0) || (time < lastTime)>

方法会对请求的楼层，时间，方向等信息的范围和搭配作判断，若范围不满足要求或第一条有效请求不为(FR,1,UP,0)或请求的时间小于上一条请求的时间等情况都会直接返回 false，因此满足<\result == false> with <(direction != null && (direction.equals("UP") || direction.equals("DOWN") || direction.equals("ER")) && (floor<1 || floor>10) || (floor==1 && "DOWN".equals(direction) || floor==10 && "UP".equals(direction)) || (time<0 || time>4294967295L) || isFirst==true && !(floor==1 && "UP".equals(direction) && time==0) || (time < lastTime)>。

若请求的楼层，时间，方向等信息的范围和搭配满足要求，且该请求为第一条有效请求(FR,1,UP,0)，则方法将 isFirst 赋值为 false，然后返回 true，因此满足<\result == true && isFirst==false> with <(direction != null && (direction.equals("UP") || direction.equals("DOWN") || direction.equals("ER")) && !(floor<1 || floor>10) && !(floor==1 && "DOWN".equals(direction) || floor==10 && "UP".equals(direction)) && !(time<0 || time>4294967295L) && isFirst==true && (floor==1 && "UP".equals(direction) && time==0)>。

若请求的楼层，时间，方向等信息的范围和搭配满足要求，且该请求不为第一条有效请求，且该请求的时间不小于上一条请求的时间，则方法将 time 赋值给 lastTime，然后返回 true 因此满足<\result == true && lastTime == time> with <(direction != null && (direction.equals("UP") || direction.equals("DOWN") || direction.equals("ER")) && !(floor<1 || floor>10) && !(floor==1 && "DOWN".equals(direction) || floor==10 && "UP".equals(direction)) && !(time<0 || time>4294967295L) && isFirst==false && time >= lastTime>。

③ **inputHandle(RequestQueue reqQueue)**

```
/**
 * @REQUIRES: reqQueue != null;
 * @MODIFIES: reqQueue;System.out;
 * @EFFECTS: (\all Request request;request 为从 System.in 中读取到的字符串 str 对
```

```
应的有效的请求;reqQueue.contains(request));
        *                (\all String str;str 为从 System.in 中读取到的不合法输入;System.out 有对应的无效提示输出);
        */
```

该方法此方法的规格可以划分为：

<(\all Request request;request 为从 System.in 中读取到的字符串 str 对应的有效的请求;reqQueue.contains(request))> with <reqQueue != null>

<(\all String str;str 为从 System.in 中读取到的不合法输入;System.out 有对应的无效提示输出)> with <reqQueue != null>

在 reqQueue != null 的前置条件下，方法首先获取控制台输入的字符串，然后进行正则表达式匹配，当方法满足楼层请求或电梯请求的正则表达式时一定有(direction != null && (direction.equals("UP") || direction.equals("DOWN") || direction.equals("ER")))，满足 checkFormat 方法的前置条件，所以 checkFormat 方法的返回值也满足其后置条件。当 checkFormat 方法返回值为 true 时 floor, time, direction 一定满足 Request 构造方法的前置条件，所以 Request 构造方法的执行结果也满足其后置条件，即新建的 Request 对象不为 null，满足 reqQueue.enQueue 方法的前置条件，所以 reqQueue.enQueue 方法的执行结果也满足其后置条件。所以方法结束时满足<(\all Request request;request 为从 System.in 中读取到的字符串 str 对应的有效的请求;reqQueue.contains(request))> with <reqQueue != null>。

如果在上述的格式检查过程中检查到输入的字符串不合法，则方法会在 System.out 有对应的无效提示输出，因此满足<(\all String str;str 为从 System.in 中读取到的不合法输入;System.out 有对应的无效提示输出)> with <reqQueue != null>。

除上述提到的方法外，该方法调用的其他方法均为 java 提供的标准库函数，其正确性一定得到满足。

④ **main(String[] args)**

```
/**
 * @REQUIRES: None;
 * @MODIFIES: System.out;
 * @EFFECTS: 未 catch 到任何异常 ==> System.out 输出所有 System.in 获得请求的调度结果;
 *               catch 到有异常发生 ==> 程序退出;
 */
```

该方法此方法的规格可以划分为：

<程序退出> with <catch 到有异常发生>

<System.out 输出所有 System.in 获得请求的调度结果> with <未 catch 到任何异常>

方法首先新建一个 RequestQueue 对象，然后赋值给 reqQueue，因为 RequestQueue 构造方法的前置条件为 None，所以 RequestQueue 构造方法的执行结果也满足其后置条件，从而使 reqQueue 不为 null，满足 inputHandle 方法的前置条件，所以 inputHandle 方法的执行结果也满足其后置条件。

然后方法新建一个 Elevator 对象并赋值给 elevator 变量，因为 Elevator 构造方法的前置条件为 None，所以 Elevator 构造方法的执行结果也满足其后置条件，从而使 elevator 不为 null，同时 reqQueue 不为 null，满足 Scheduler 构造方法的前置条件，所以 Scheduler 构造方法

的执行结果也满足其后置条件。

方法在调度请求的过程中会不断取出队头请求，若该请求不为同质请求且未被执行则会将其升级为主请求然后执行，执行过程中不断更新调度器和电梯信息，并检查队列中的同质请求和可捎带请求，主请求执行完毕后若队列中还有可升级为主请求的可捎带请求，则将该请求升级为主请求继续执行，每执行完一条请求都会在 System.out 输出该请求的调度结果。

方法只有请求队列非空时才会不断取出队头请求，所以会保证 mainReq 不为 null，满足 changeMainRequest 方法的前置条件，方法的执行结果也满足其后置条件。同时也只有电梯所处楼层与请求楼层不同时会调用 runUpOrDown 方法，满足 runUpOrDown 方法的前置条件，方法的执行结果也满足其后置条件；只有电梯所处楼层与请求楼层相同时会调用 runStill 方法，满足 runStill 方法的前置条件，方法的执行结果也满足其后置条件。另外方法中调用的 updateStatus 方法和 setTime 方法的参数均来源于电梯对象或请求对象本身，因此也必定满足方法的后置条件，方法的执行结果也满足其后置条件。

除了上面提到的这些方法之外，该方法中调用的其他方法前置条件均为 None，显然满足，因此也必定满足方法的后置条件。

若方法在上述执行过程中未发生任何异常，则方法结束时会在 System.out 输出所有请求的调度结果。满足<System.out 输出所有 System.in 获得请求的调度结果> with <未 catch 到任何异常>。

若方法在上述执行过程中 catch 到任何异常，则程序退出，满足<程序退出> with <catch 到有异常发生>。

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