

# New Elevator Framework

---

## Table of Contents

Part 1: 设计思路..... 2

Part 2:程序结构..... 4

    1、 Loader..... 4

    2、 NewElevatorFramework ..... 4

Part 3:SimulateProgram..... 5

Part 4:Commons ..... 6

Part 5:Buttons and ButtonPanels..... 6

Part 6:Passenger ..... 7

Part 7:Elevator..... 8

Part 8:Scheduler..... 9

Part 9:Utility.....10

## Part 1: 设计思路

在笔者看来，整个电梯调度的过程中有三个组成部分：乘客群、电梯群、电梯调度器。

现在关键的问题是这三个部分之间是怎么交互的。

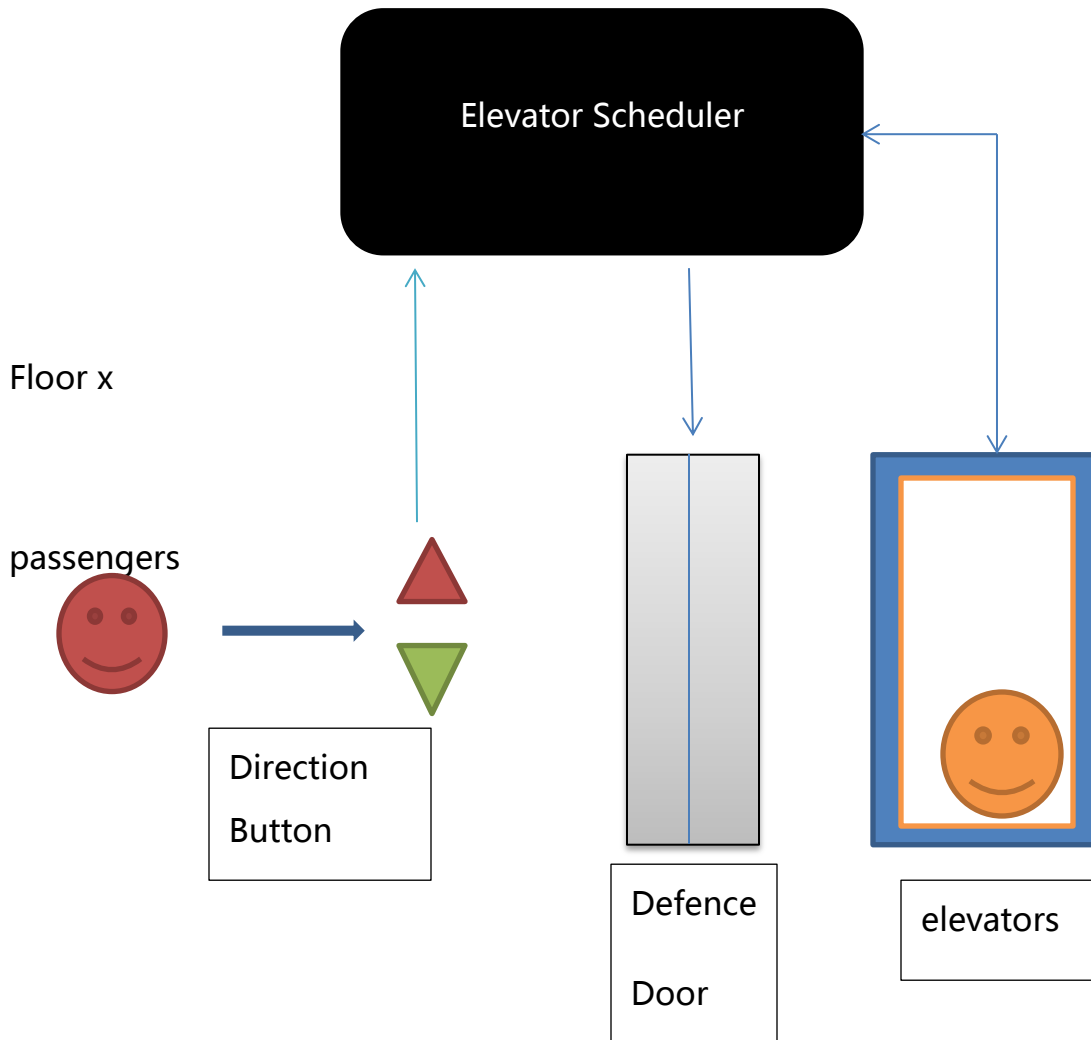
就电梯而言，电梯并没有智能，它并不知道有没有人进入它内部，或者说电梯根本不知道进入电梯的物体具体是什么。但是电梯可以知道目前在电梯内部的物体有多重（从很多电梯超重后会有警报也可以看出）。电梯要去哪也是有调度器来告诉它的。

就乘客而言，乘客只关注电梯到了没，去哪，他并不知道电梯的一切，而且也不知道有电梯调度器的存在。乘客和电梯的交互都是通过按钮来执行的。

可见：{乘客对按钮的操作}=>{调度器收到 task}=>{调度器把 task 分给电梯}=>{电梯完成目前 assign 给它的 task}。

具体一点，整个实体解构应该像下图：

（Defence door 是指保护门，当电梯还没到时保持关闭，避免乘客看到电梯的管道。）



如上图，乘客按下 direction button，elevator scheduler 就可以通过查看所有 button 的状态查看到最新按下的按钮，于是，scheduler 就通知一部适合的电梯前往 passenger 所在地楼层，当电梯到达相应的楼层是，电梯通知 scheduler，scheduler 打开相应的 defence door 让乘客进去。

在电梯内部的过程也一样，而且更简单，在这就不赘述了。

## Part 2:程序结构

“解决方案” NewElevatorFramework 包含两个项目：  
Loader,NewElevatorFramework。

### 1、Loader

所谓的 Loader 就是从保存 elevators 和 passengers 信息的 xml 文件中反序列化出相应的信息，用以初始化用于模拟调度的电梯和乘客数据。具体里面的代码不用深究，**主要注意的有两点**：

- 1、 在该文件夹下的 elevators.xml, passenger1,2,3(.xml),需要是作为输入的数据信息，在将解决方案 build 好后，运行的时候，输入文件就是这几个中，当然，也可以根据自身的测试需要，按照以上文件的格式生成自己的测试文件。（by the way，elevator.xml 里的电梯对 passengers.xml 里的某个乘客是无力的，你们可以自己看看....）
- 2、 这个项目内的源代码不用去看，主要注意它们从文件中读取信息后返回的结果结构就行了，这些结构会在项目 NewElevatorFramework 中用于初始电梯和乘客对象。

### 2、NewElevatorFramework

主程序在 SimulateProgram.cs 内，文件 SimulateProgram.cs 里只有定义了一个同名的类，顾名思义，class SimulateProgram 就是电梯调度的模拟程序。

Commons.cs 定义了整个项目的通用接口。

Buttons.cs 实现了 Commons 中的 IButton 接口，并定义了实际的两个类型：**电梯外地方向按钮类型**和**电梯内部的楼层按钮类型**

ButtonPanel.cs 实现了 Commons 中的 IButtonPanel 接口，这个类是为了继承出 SpecificallyButtonPanel.cs 中的两个具体类（**电梯外地按钮面板**和

电梯内的按钮面板) 而存在的；这个形式和 Button.cs 中的实现是一样的，只是 class ButtonPanel 写的有点大了，所有另分了一个文件写它的两个子类。

Passenger.cs、Elevator.cs、Scheduler.cs 这三个文件定义的是乘客、电梯、调度器的具体实现。

Utility.cs 中定义了一个静态类 Utility，提供一系列帮助调试的工具函数。

## Part 3: Simulate Program

在主程序中：

- ⇒ 会先生成一个 SimulateProgram 的对象；
- ⇒ 然后载入记录有电梯、乘车信息的 xml，完成对电梯对象，乘客的对象的生成和初始化；
- ⇒ 根据取得的信息构造一个电梯调度器；
- ⇒ 初始化模拟器的时钟，开始一模拟时钟地每个 tick 为单位进行电梯调度模拟

SimulateProgram 中定义了模拟过程中乘客的行为：

即 `passengerTakeActions` 函数，每次该函数调用，它都会遍历所有乘客，判断他们的状态，并根据他们所处的状态（刚到，已到达，未到达还在等电梯，未到达但已经在电梯内等），驱使乘客采取相应的操作。

`elevatorsRun` 函数，每次该函数调用，它都会督促所有电梯完成本个 tick 应该完成的事。

`schedulerDepatchTasks` 函数，每次调用它，都会对电梯进行任务分配。

## Part 4: Commons

Commons 中分成两个部分，第一部分是各种 enum 的声明，第二部分是各种接口的声明。

Enum 声明中，有两个可能不是很容易理解：

`IndexOfOutsidButton`，在具体程序实现中，每个按钮在按钮面板中都有一个序号，以方便按钮的查找（**实际上就是数组和其下标**）。

`MotionOfElevator`，这是模拟现实物理世界的运动限制而给出的量，`Velocity` 是电梯的最高运行时速，`DecelerationSpace` 是电梯的减速距离，即电梯在最高速的状态下开始减速后还要运行多远才会停下。

接口声明中，人、电梯、调度器中只有电梯定义了接口，因为在目前的调度结果下，只有电梯可能需要作为参数传来传去，本着减小一点耦合性的观念，给电梯定义了一个接口。

## Part 5: Buttons and ButtonPanels

这两部分中，`Button` 很好懂，`ButtonPanel` 中有一个属性：

`AllNewlyPressedButton`，这个属性是和

方法 `resetNewlyPressedButtonNotification` 结合在一起的，

所谓的 `newlyPressedButton`，就是从上一次 `resetNewlyPressedButtonNotification`（）执行后到调用 `newlyPressedButton` 过程中新按的按钮。（看一下具体实现）

```

public void resetNewlyPressedButtonNotification() {
    hasNewlyPressedButton = false;
    allNewlyPressedButton.Clear();
}

```

ButtonPanel 中的两个数组 buttonDisplay 和 buttons，其实看一下和 buttonDisplay 相对应的属性 ButtonDisplay 就知道，buttonDisplay 有两个意义：

```

public ButtonState[] ButtonDisplay {
    get {
        for (int i = 0; i < buttonCounts; i++) {
            buttonDisplay[i] = buttons[i].State;
        }
        return buttonDisplay;
    }
}

```

- 1、 在实际中可以认为是乘客可以看到的面板按钮的状态；
- 2、 在程序中是面板按钮状态的快速查找数组；

## Part 6:Passenger

这个类型即是模拟中的乘客。

在这个类型中可以看出，passenger 仅与各种 button 进行交互：

```

public void pressButtonInsideElev(IButtonPanel buttonPanel) ...
public void pressButtonOutsideElev(IButtonPanel buttonPanel) ...

```

除了按按钮之外，就只有进电梯，出电梯的行为：

```

public bool enterElevator(IElevator comingElev) ...
public bool leaveElevator(int leaveTick) ...

```

在模拟环境中具体的判断行为(合适进电梯，出电梯)，是在模拟环境下给出的，即 (SimulateProgram.cs)：

```

//detect passengers' actions
simulator.passengerTakeActions();

//whether the passenger is appeared during the global time scope
if (passenger.ComingTime <= globalTickCount)
{
    //the passenger is inside the elevator
    if (passenger.IsInsideTheElev)
    {
        actionsInsideElevator(passenger);
    }
    else
    { //the passenger is outside the elevator
        actionsOutsideElevator(passenger);
    }
}

```

⇒

```

void actionsInsideElevator(Passenger someOne) ...
void actionsOutsideElevator(Passenger someOne) ...

```

⇒

## Part 7:Elevator

```

public bool addWeight(int weight) ...
public bool subWeight(int weight) ...

public void openDoor() ...
public void closeDoor() ...

public bool setTargetFloor(int floorNumber) ...
public void resetTargetFloor() ...

public void resetDoorOpeningTime() ...

public void run() ...

```

对电梯来说，乘客的进出就是重量的变化，即 add、sub weight；



v-guil@microsoft.com

setTargetFloor 是确定电梯目前的目的地；resetTargetFloor，是电梯每到一个楼层，在调度器还没有给它新的目的地之前，所进行的操作，表示此时电梯没有要去的目的地。

Run 是电梯的运行。

这几个是电梯的基本方法，下面几个是电梯的事件相关的方法：

```
//add and remove event listener
public void addEventListener(EventType eventType, EventHandler handler)
public void removeEventListener(EventType eventType, EventHandler handler)
private void operationOnEventListener(EventType eventType, EventHandler handler)
```

这里模仿了 Javascript 的事件添加、移除接口；

这里的 EventType 指的是开门事件和关门时间，因为电梯开门关门的时候必须通知调度器，这样调度器才好把电梯外地 defensive door 给关上，并更新相应按钮的状态。

## Part 8:Scheduler

这是电梯调度器。

调度器的使用有点繁琐，首先它有个构造函数，可以通过个构造函数生成一个调度器；然后用函数 bindWithElevators（）将它与一组电梯绑定在一起，这样才能真正使用。

函数 despatchQueriesToElev()是重中之重，因为这里面包含的应该是调度算法。如果使用者愿意的话，可以不用对其他地方进行修改，只在这个函数内填入算法（辅助函数随意加），就可改变整个电梯调度。

目前调度器中自带一个 naïve 的 bus 算法，它是让 4 个电梯逢层就停（不管怎么说，这是个稳定而且一定可行的算法.....）。

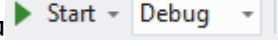
```
private void onElevatorDoorOpen(object sender, EventArgs e) ...
private void onElevatorDoorClose(object sender, EventArgs e) ...
private void abstractDoorEventHandler(object sender, EventArgs e, D
```

这是时间处理程序，是调度器对电梯开关门的处理。

10

## Part 9:Utility

以 log 开头的三个函数是用来记录日志的，在 Utility.cs 文件的最开头有两个宏定义，这两个宏定义可以控制日志对控制台的输出；

log 是普通日志在 Debug 模式（vs2012 中 ）下，  
`#define SHOW_DETAILS_IN_DEBUG` 存在时，会向控制台输出，字体为绿色；

logWarning 是警告日志，在 Debug 模式，且：  
`#define SHOW_DETAILS_IN_DEBUG`  
`#define SHOW_WARNING_IN_DEBUG` 时，向控制台输出，字体为黄色；

logError 是错误日志，一定会向控制台输出，字体为红色；

函数 saveLogRecord，在 debug 模式下会将日志保持到文件；

函数 outputAnalysisResult，是专门针对已经完成了的模拟，将模拟过程中乘客的信息，时间花费打印，并计算平均时间。

函数 howManyPassengerInTotal，返回模拟程序拥有的乘客数；

v-guil@microsoft.com

函数 `howManyPassengerArrived`, 返回已经达到的乘客人数;

函数 `howManyPassengerNonarrived`, 返回未到达目的地的乘客人数;