处理机管理-电梯调度 设计方案报告

目录

开发环境

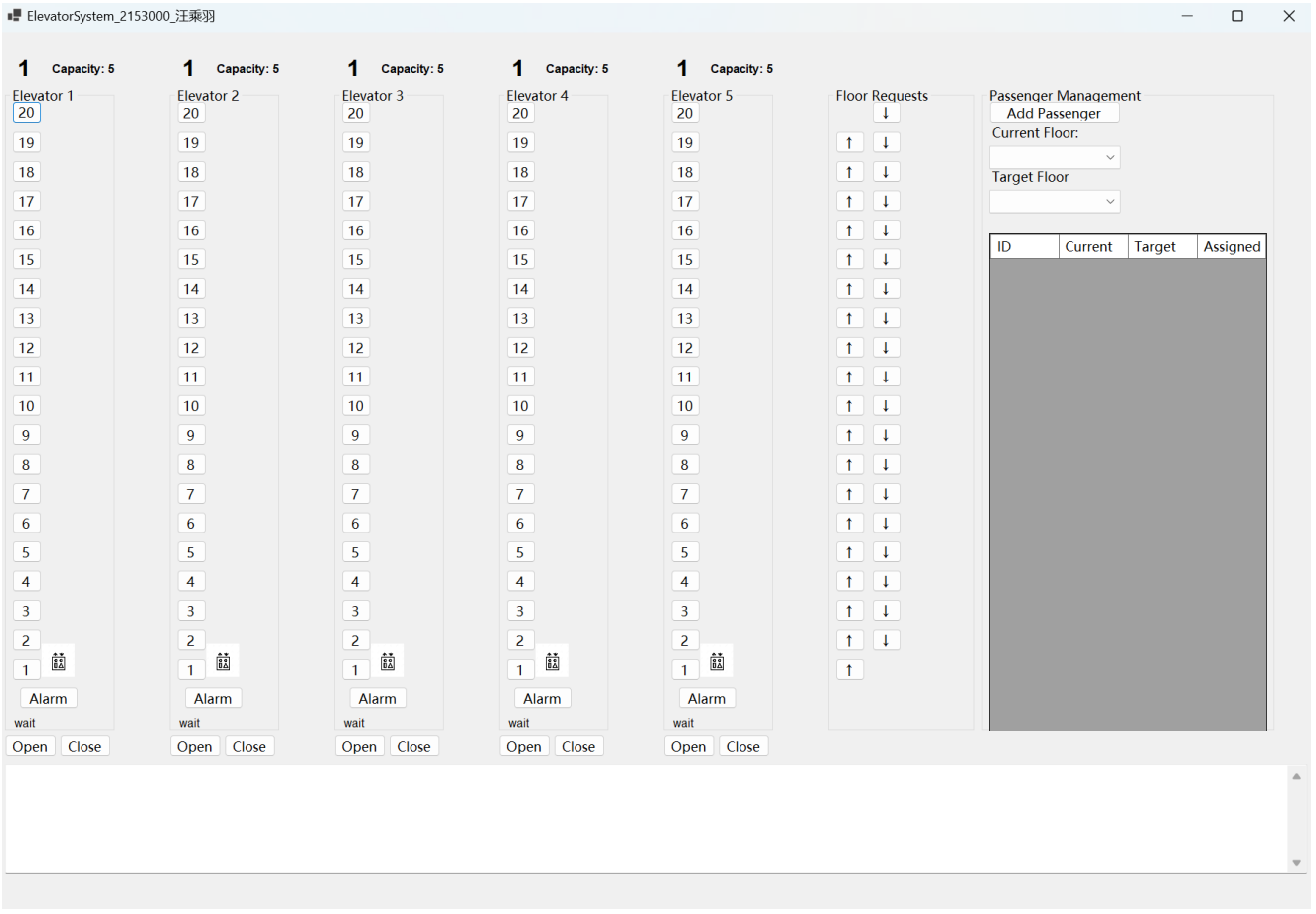
- 开发环境：Windows 11
- 开发软件：Visual Studio 2022
- 开发语言：C#
- 主要引用模块：
    - System
    - System.Threading

实现的功能

- 互联接的上下行按钮：当按下五部电梯门口的上下行按钮时，五部电梯都收到这个请求，根据调度算法决定具体由哪部电梯响应这个请求；
- 独立的楼层按钮：每部电梯都有各自的楼层按钮，当按下电梯内部的楼层按钮时，该请求由这部电梯响应，具体何时响应由调度算法决定；
- 添加乘客功能：支持添加乘客，需指定乘客起始楼层和目标楼层，调度算法决定由哪部电梯响应乘客的请求；
- 数码管显示电梯当前楼层：每部电梯都有一个数码管显示其当前所在楼层；
- 电梯状态显示：每部电梯都有一个标签显示电梯当前状态：有up，down，wait，alarm四种状态；
- 电梯容量显示：每部电梯都有一个标签显示电梯可用容量，初始为5；
- 开门、关门键：当电梯到达乘客所在楼层和目标楼层时，需要完成开门和关门操作；

- 提示信息框：在特定时候弹出提示信息，显示乘客被分配到了哪部电梯以及提示用户进行开门或关门操作；
- 乘客状态表：根据乘客状态实时更新乘客当前所在楼层，只显示未到达目标楼层的乘客；

界面



主要设计思路

实现的类

- Form1类：程序的主窗口，负责UI的显示与更新，处理UI的点击事件，管理与乘客有关的逻辑、订阅ElevatorController中定义的事件；
- ElevatorController类：实现了电梯的调度算法，管理电梯列表，定义了一些关键的事件；
- Elevator类：记录电梯的属性，包括电梯的当前楼层，目标楼层，电梯的状态，电梯的容量，提供了一些公用的方法，可以对电梯的这些属性进行设置和获取；
- Passenger类：记录每个乘客的ID，当前所在楼层和目标楼层，被分配到的电梯，以及请求是否已被处理；

类之间的联系

- 通过在Form1中引入ElevatorController的示例manage，实现两个类的交互，在Form1的初始化函数中完成对在ElevatorController中定义的事件的订阅：

```
 public Form1()
{
    InitializeComponent();
```

```
        InitializeElevatorUI();
        InitialMassageBox();
        InitializeFloorRequestButtons();
        InitializePassengerManagementUI();
        manage = new ElevatorController();
        manage.ElevatorMoved += Manage_ElevatorMoved;
        manage.ElevatorRequestProcessed += Manage_ElevatorRequestProcessed;
        manage.ElevatorExternalRequestProcessed +=
Manage_ExternalRequestProcessed;
        manage.OpenDoorPressed += Manage_OpenDoorPressed;
        manage.CloseDoorPressed += Manage_CloseDoorPressed;

        LoadImages();
        InitializeTimer();
    }
```

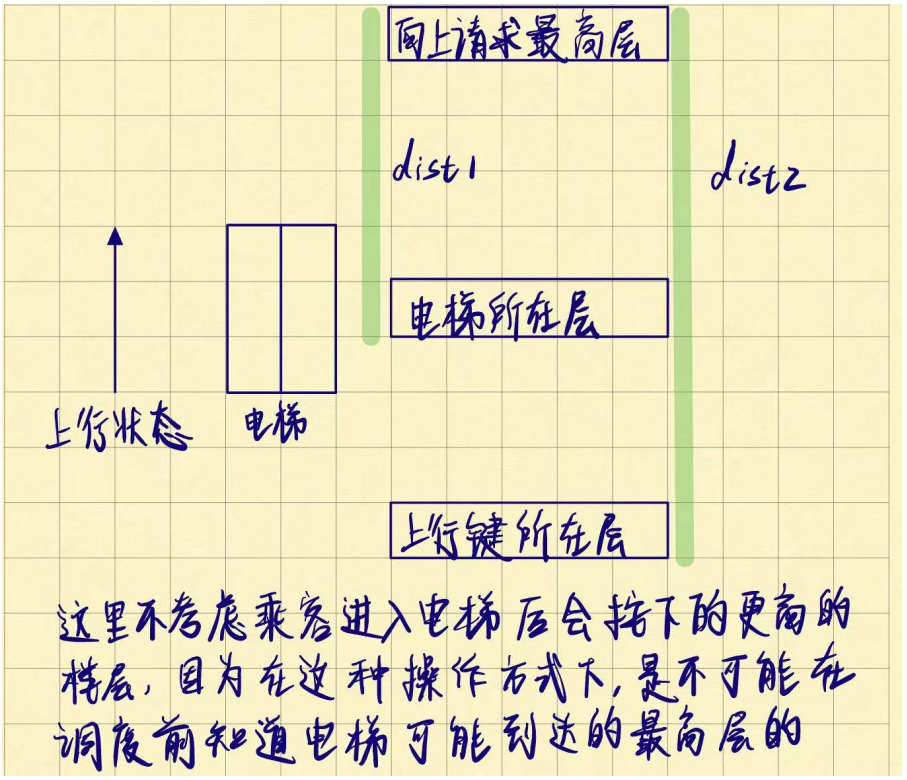
- 在Form1中定义Passenger类型的列表passengers，实现对乘客的管理；
- 在ElevatorController中定义Elevator类型的数组elevators，实现对五部电梯的管理；

调度算法

- 在ElevatorController中定义四个二维数组，记录需要处理的上行和下行请求，以及等待处理的上行和下行请求：

```
public int[,] upRequest = new int[ElevatorsCount, FloorsCount];
public int[,] downRequest = new int[ElevatorsCount, FloorsCount];
//wait数组里储存的是电梯需要经过状态变换才能处理的请求
public int[,] waitUpRequest = new int[ElevatorsCount,FloorsCount];
public int[,] waitDownRequest = new int[ElevatorsCount, FloorsCount];
```

- wait数组里储存的时电梯需要经过状态变换才能处理请求；
- 当上行键被按下时，调用public int handleUpRequest(int floor)，处理上行请求；当下行键被按下时，调用public int handleDownRequest(int floor)，处理下行请求；两个函数的返回值都是被分配到的电梯编号；具体来说，调度算法如下（以处理上行请求为例）：
    - 根据每部电梯的状态和电梯所在楼层计算与被按下的上行键所在楼层的距离：
        - 如果电梯处于静止状态，这个距离等于电梯所在楼层减去上行键所在楼层的绝对值；
        - 如果电梯处于上行状态，那么分为两种情况：
            - 电梯当前所在楼层大于上行键所在楼层，这个距离等于该电梯目前向上请求中最高的层减去电梯所在楼层的绝对值，加上电梯目前向上请求中最高的层减去上行键所在楼层的绝对值；



        - 电梯当前所在楼层小于上行键所在楼层，这个距离等于电梯当前所在楼层减去上行键所在楼层的绝对值；
    - 如果电梯处于下行状态，那么这个距离等于该电梯目前向下请求中的最低的层减去电梯所在楼层的绝对值，加上电梯目前向下请求中最低的层减去上行键所在楼层的绝对值；
- 选出与上行键所在楼层距离最近的电梯，并在其上行请求列表中添加上行键所在的楼层：
    - 如果该电梯目前处于静止状态，那么有两种情况：
        - 上行键所在楼层大于电梯所在楼层，把该请求添加进`upRequest[elevatorIndex,floor]`中，因为电梯即将改变成上行状态，该请求能立即得到处理；
        - 上行键所在楼层小于电梯所在楼层，把该请求添加进`waitUpRequest[elevatorIndex,floor]`中，因为电梯需要先改变成下行状态，再改变成上行状态，才能处理该请求；
    - 如果电梯目前处于上行状态，那么有两种情况：
        - 上行键所在楼层大于电梯所在楼层，把该请求添加进`upRequest[elevatorIndex,floor]`中，因为电梯不需要改变状态就可以响应该请求；
        - 上行键所在楼层小于电梯所在楼层，把该请求添加进`waitUpRequest[elevatorIndex,floor]`中，因为该电梯需要改变状态才能响应该请求；
    - 如果电梯处于下行状态，那么直接把请求添加进`waitUpRequest[elevator,Index]`中，因为该电梯需要改变状态才能响应该请求。
- 当楼层按钮被按下时，直接对upRequest和downRequest进行修改，当在上行电梯中按下比电梯当前所在楼层小的楼层，将不会被响应，当在下行电梯中按下比电梯当前所在楼层小的楼层，将不会被响应：

```
1  private void FloorButton_Click(object sender, EventArgs e, int
elevatorIndex, int floor)
2  {
3
4      Button clickButton = sender as Button;
```

```
5
6       // 处理楼层按钮点击事件
7       //MessageBox.Show($"电梯 {elevatorIndex + 1} 的第 {floor + 1} 层按钮被
点击");
8       if (manage.elevators[elevatorIndex].getStatus()==status.up)
9       {
10          if (floor >= manage.elevators[elevatorIndex].getCurrentFloor())
11          {
12              manage.upRequest[elevatorIndex, floor] = 1;
13              if (clickButton != null)
14              {
15                  clickButton.BackColor = Color.Green;
16              }
17
18          }
19      }
20      else if (manage.elevators[elevatorIndex].getStatus()==status.down)
21      {
22          if (floor <= manage.elevators[elevatorIndex].getCurrentFloor())
23          {
24              manage.downRequest[elevatorIndex, floor] = 1;
25              if (clickButton != null)
26              {
27                  clickButton.BackColor = Color.Green;
28              }
29
30          }
31      }
32      else
33      {
34          if (floor >= manage.elevators[elevatorIndex].getCurrentFloor())
35          {
36              manage.upRequest[elevatorIndex, floor] = 1;
37              if (clickButton != null)
38              {
39                  clickButton.BackColor = Color.Green;
40              }
41
42          }
43          else
44          {
45              manage.downRequest[elevatorIndex,floor] = 1;
46              if (clickButton != null)
47              {
48                  clickButton.BackColor = Color.Green;
49              }
50
51          }
52      }
53  }
```

电梯运行逻辑

- 为每部电梯启动一个独立的线程，并设置线程为后台线程，线程执行`private void ElevatorRoutine(int elevatorIndex)`方法：

```
private void StartElevatorThreads()
{
    for(int i=0;i<ElevatorsCount;i++)
    {
        int index = i;
        Thread elevatorThread = new Thread(() => ElevatorRoutine(index))
        {
            IsBackground = true
        };
        elevatorThread.Start();
    }
}

private void ElevatorRoutine(int elevatorIndex)
{
    while (true)
    {

        Thread.Sleep(500);
        pullUpward(elevatorIndex);
        Thread.Sleep(500);
        prepareForDown(elevatorIndex);
        Thread.Sleep(500);
        pushDownward(elevatorIndex);
        Thread.Sleep(500);
        prepareForUp(elevatorIndex);
    }
}
```

- 在`void pullUpward(int elevatorIndex)`里处理电梯的上行请求，在`void pushDownward(int elevatorIndex)`里处理电梯的下行请求；
- 对`waitUpRequest`和`waitDownRequest`的处理分别在`prepareForUp`和`prepareForDown`中进行；
- 在`ElevatorController`的初始化函数中启动线程：

```
public ElevatorController()
{
    for(int i=0;i<elevators.Length; i++)
    {
        elevators[i] = new Elevator();
    }

    for(int i=0;i < ElevatorsCount;i++)
    {
        for(int j=0;j < FloorsCount;j++)
        {
            upRequest[i, j] = 0;
            downRequest[i, j] = 0;
            waitUpRequest[i, j] = 0;
            waitDownRequest[i, j] = 0;
        }
    }
```

```
    StartElevatorThreads();
}
```

事件定义

- 在ElevatorController中定义了五个事件：

```
    public event Action<int, int> ElevatorMoved;
    public event Action<int, int> ElevatorRequestProcessed;
    public event Action<int, int> ElevatorExternalRequestProcessed;
    public event Action<int> OpenDoorPressed;
    public event Action<int> CloseDoorPressed;
```

  分别用于处理电梯移动，电梯楼层请求被处理和电梯外部请求被处理，开门按钮被按下和关门按钮被按下；

- 在Form1中订阅这些事件：

```
  manage.ElevatorMoved += Manage_ElevatorMoved;
  manage.ElevatorRequestProcessed += Manage_ElevatorRequestProcessed;
  manage.ElevatorExternalRequestProcessed += Manage_ExternalRequestProcessed;
  manage.OpenDoorPressed += Manage_OpenDoorPressed;
  manage.CloseDoorPressed += Manage_CloseDoorPressed;
```

- 事件订阅是一个在软件设计中常见的模式，它属于更广泛的发布-订阅（pub-sub）模式。在这种模式中，组件可以宣布（发布）某些事件的发生，而其他组件可以表达对这些事件的兴趣（订阅），当事件发生时，这些订阅了事件的组件会自动接收通知并作出相应的响应。
- 例如，在ElevatorController的pullUpward方法中，执行下面这个片段，程序会检验电梯当前所在楼层是否存在请求，如果有，则请求被响应，事件被触发，接着订阅了事件的方法将会被调用，楼层按钮和上行按钮的高亮会消失，电梯也能得到位置的更新：

```
 while (elevators[elevatorIndex].getCurrentFloor() != elevators[elevatorIndex].getTargetFloor())
 {
     elevators[elevatorIndex].move();
     if (upRequest[elevatorIndex, elevators[elevatorIndex].getCurrentFloor()] == 1)
     {
         upRequest[elevatorIndex, elevators[elevatorIndex].getCurrentFloor()] = 0;
         ElevatorRequestProcessed.Invoke(elevatorIndex, elevators[elevatorIndex].getCurrentFloor());

         ElevatorExternalRequestProcessed.Invoke(elevators[elevatorIndex].getCurrentFloor(), 1);
     }
     ElevatorMoved.Invoke(elevatorIndex, elevators[elevatorIndex].getCurrentFloor());
 }
 if (elevators[elevatorIndex].getCurrentFloor() == i)
 {
     upRequest[elevatorIndex, i] = 0;
     ElevatorRequestProcessed.Invoke(elevatorIndex, i);
     ElevatorExternalRequestProcessed.Invoke(i, 1);
 }
```

定义定时器

- 定时器（Timer）在编程中用于在设定的时间间隔内重复执行某些任务，或者在一定的延迟之后执行任务。`checkElevatorTimer.Elapsed += CheckElevatorStatus;`这行代码表示将`CheckElevatorStatus`方法绑定到定时器的`Elapsed`事件。这意味着每当定时器的计时周期结束时，`CheckElevatorStatus`方法将被自动调用。`AutoReset = true`设置定时器在触发`Elapsed`事件后自动重新开始计时。这使得定时器成为一个重复触发的周期性定时器。
- 项目中一共使用了三个定时器，分别用于播放动画，检查电梯状态，以及更新乘客列表：

```
 private void InitializeTimer()
 {
     animationTimer.Interval = 200; // 设置动画帧切换的时间间隔为200毫秒
     animationTimer.Tick += new EventHandler(AnimationTimer_Tick);
     updateTimer = new System.Windows.Forms.Timer();
     updateTimer.Interval = 1000;  // 更新间隔设置为1000毫秒（1秒）
     updateTimer.Tick += new EventHandler(UpdateTimer_Tick);
     updateTimer.Start();
     // 初始化定时器
     checkElevatorTimer = new System.Timers.Timer(1000); // 设置时间间隔为1
秒
     checkElevatorTimer.Elapsed += CheckElevatorStatus;
     checkElevatorTimer.AutoReset = true;
     checkElevatorTimer.Enabled = true; // 初始状态为禁用
 }
```

- CheckElevatorStatus定时检查是否有电梯到达了乘客的目标楼层，并更新电梯的状态标签：

```
private async void CheckElevatorStatus(Object source,
System.Timers.ElapsedEventArgs e)
{
    // 在Task.Run内执行需要在UI线程上完成的操作
    await Task.Run(() =>
    {
        Invoke((MethodInvoker)delegate
        {
            foreach (var passenger in passengers)
            {
                if (!passenger.isHandled &&
manage.elevators[passenger.assignedElevator].getCurrentFloor() ==
passenger.currentFloor)
                {
                    passenger.isHandled = true;
                    messageBox.AppendText($"Please press the open door
button for Elevator {passenger.assignedElevator + 1} to let the passenger
in.{Environment.NewLine}");


                    var openTcs = new TaskCompletionSource<bool>();
                    openDoorTcs[passenger.assignedElevator] = openTcs;
                    openTcs.Task.ContinueWith(async t =>
                    {
                        // 等待两秒后，执行后续逻辑
                        await Task.Delay(2000);

 openDoorButtons[passenger.assignedElevator].BackColor = Color.White;
                        var closeTcs = new TaskCompletionSource<bool>();
```

```
                        closeDoorTcs[passenger.assignedElevator] = closeTcs;
                        await closeTcs.Task;

                        // 关门后延迟两秒
                        await Task.Delay(2000);

 closeDoorButtons[passenger.assignedElevator].BackColor = Color.White;
                        Invoke((MethodInvoker)(() => toTargetFloorOfPassenger(passenger.id, passenger.assignedElevator, passenger.targetFloor)));
                    }, TaskScheduler.FromCurrentSynchronizationContext());
                }
            }
        });
    });


    // 其他状态更新也必须在UI线程上执行
    Invoke((MethodInvoker)delegate
    {
        for (int i = 0; i < ElevatorsCount; i++)
        {
            Elevator elevator = manage.elevators[i];
            status currentStatus = elevator.getStatus();
            int capacity = elevator.getCapacity();
            elevatorCapacityLabels[i].Text = $"Capacity: {capacity}";

            // 更新状态标签颜色及文字
            UpdateElevatorStatusLabels(currentStatus, i);
        }
    });
}

```