Project Hotel

Voor de Hotel Simulatie die voor het eerste blok in jaar 2 van HBO-ICT Zoetermeer gemaakt moet worden, hebben we een aantal keuzes gemaakt die van toepassing op de code zijn. Ook voor verschillende onderdelen van het hotel hebben we een paar keuzes gemaakt die beter gedocumenteerd kunnen worden dan met wat uitge-commente code. Vandaar dat daar een apart document voor is gemaakt.

Contents

[Lift 3](#_Toc527653842)

[Interactie met de klant 3](#_Toc527653843)

[Route calculatie 4](#_Toc527653844)

[Slimme Lift 5](#_Toc527653845)

[Pathfinding 7](#_Toc527653846)

[Graph 7](#_Toc527653847)

[CreateGraph 7](#_Toc527653848)

# Lift

Aan de lift waren een aantal requirements aan gebonden. De lift moest namelijk een slimme lift zijn, oftewel: de lift moest efficiënt klanten ophalen en wegbrengen naar hun verdieping zoals een echte lift dat ook doet. Daarvoor zijn we door een aantal designs gegaan, en hebben we besloten om de lift te laten werken met twee **List<int>** om de verdiepingen waar die naartoe moet bij te houden, een **Enum** voor de directie waar die op gaat en om de **int PositionY** te gebruiken om aan te geven op welke verdieping die zicht bevindt.

## Interactie met de klant

Omdat een klant natuurlijk sommige dingen niet kan weten van een lift (bijvoorbeeld welke verdiepingen die allemaal af moet gaan) hebben we gekozen om sommige elementen **private** te houden (de 2 **List<int>** en de **Enum**). De klant kan alleen informatie van de lift halen wanneer de **GetElevatorInfo()** methode is aangeroepen die in de Elevator class staan. Hierbij wordt gebruik gemaakt van de **System.ValueTuple** NuGet Package, waardoor meerdere waardes in een **Tuple** ge-returned kunnen worden *(zie de code hieronder)*.

public (char, int) GetElevatorInfo()

{

return (Direction, PositionY);

}

*GetElevatorInfo() methode uit de Elevator.cs class*

Deze methode kan aangeroepen worden de klant. De **(char, int)** voor de **GetElevatorInfo()** is de **Tuple** die ge-returned wordt naar de klant wanneer de methode wordt aangeroepen. De lift geeft dan de kant waar die toe gaat terug en de huidige **PositionY** (verdieping) waar die is. Met deze twee gegevens kan de klant uitrekenen welke route die moet nemen (via de trap of met de lift, *zie hoofdstuk* ***Pathfinding*** *voor meer informatie*).

public void RequestElevator(int RequestFloor)

{

if (RequestFloor <= Hotel.Floors.Length - 1 && RequestFloor >= 0)

{

if (RequestFloor > PositionY)

{

Up.Add(RequestFloor);

}

else

{

Down.Add(RequestFloor);

}

UpdateList();

}

}

*RequestElevator(int RequestFloor) methode uit de Elevator.cs class*

Dit is de methode die aangeroepen wordt als de klant met de **Elevator** wilt gaan. Er wordt een **int** meegegeven die wordt toegevoegd aan de **Down-** of **Up** **List<int>**. Allereerst checkt de **if** statement of de **int RequestFloor** eigenlijk wel binnen het **Hotel** zit. Dit doet die door te kijken naar de maximale hoogte van het hotel (**Hotel.Floors.Length**) en door te kijken of het groter of gelijk is aan 0 (**RequestFloor >= 0**). Daarna kijkt de methode of de **RequestFloor int** boven of onder de **Elevator** zijn **PositionY** zit. Als die erboven zit, wordt de **RequestFloor int** toegevoegd aan de **Up List<int>**, anders wordt die toegevoegd aan **Down List<int>**. Als laatste worden de **List**’s ge-update door de **UpdateList()** methode (*zie kopje* ***Route calculatie*** *voor meer informatie* over de **UpdateList()** methode*)*.

## Route calculatie

De lift in het hotel is dus een “slimme” lift, maar wat maakt een lift nou “slim”? Dat is wat we ons eerst af gingen vragen voordat we begonnen met het schrijven van de **Elevator** code. Om uit te zoeken wat een slimme lift nou daadwerkelijk is, hebben we besproken hoe een lift klanten zo efficiënt mogelijk kan ophalen. Hierbij zijn we tot de conclusie gekomen dat de lift hiervoor 2 **Lists** hiervoor kan gebruiken, een **List** die bijhoud welke verdiepingen hij af moet gaan wanneer de lift omhoog gaat en een **List** die behoud welke verdiepingen hij af moet gaan wanneer die naar beneden gaat. We zijn begonnen met dit op te slaan in een **List<int>**, maar zijn daarna op de hoogte gebracht van Tim Cocx *(leraar HBO-ICT Zoetermeer, onze Projectbegeleider voor dit project)* dat een **HashSet<>** een stuk makkelijker is. Voordat we de **HashSet<>** gingen gebruiken, zijn we even gaan kijken of de **HashSet<>** wel een goed idee is om te gebruiken. Al snel kwamen we erachter dat een **HashSet<>** geen Indexing had (waar onze **Elevator** klasse erg veel gebruik van maakt). Het enige probleem dat een **HashSet<>** zou oplossen, is dat bij de **UpdateList()** methode (zie volgende pagina) de **Distinct()** niet meer gebruikt hoefde te worden. We zijn daardoor tot de conclusie gekomen dat de twee **List<>** methode die we gebruikte handiger was, vergeleken met het ombouwen van de **Elevator** klasse om de 2 **HashSet<>** te implementeren.

private void UpdateList()

{

Up = Up.Distinct().OrderBy(x => x).ToList();

Down = Down.Distinct().OrderByDescending(x => x).ToList();

}

*UpdateList() methode uit de Elevator.cs class*

Hierboven is te zien hoe de **List**’s worden ge-update elke keer dat een **ElevatorShaft** werd toegevoegd. Door een **HashSet<>** te gebruiken in plaats van een **List<>** kan de **.Distinct()** query weggehaald worden, want een er kunnen geen duplicate waardes toegevoegd worden aan een **HashSet<>**. We hebben echter gekozen voor de **List<>** in plaats van de **HashSet<>**. Voor de rest wordt de lijst geordend op de **int** waarde in de Up- en Down **List<>**, wat betekend dat de Up **List<>** wordt geordend op een normale wijze (van laag naar hoog), terwijl de Down **List<>** wordt geordend **ByDescending** (van hoog naar laag).

### Slimme Lift

Nu komen we bij het gedeelte aan waar we uitleggen hoe onze slimme lift werkt. Onze slimme lift werkt door de **Enum** die aan de **Directie** gebonden is. We gaan stapsgewijs door de code heen.

if(Direction == ElevatorDirection.IDLE)

{

if(Up.Count > Down.Count)

{

Direction = ElevatorDirection.UP;

}

else

{

Direction = ElevatorDirection.DOWN;

}

}

*Een stuk uit de Move() methode uit de Elevator.cs class*

**Direction** is dus de **Enum** voor de richting van de lift. Hier wordt er even gecheckt of de lift **IDLE** is en of die dus moet bewegen. Dit wordt gedaan door te kijken naar het aantal **int**’s in de **Up**- en de **Down** **List<int>** en die met elkaar te vergelijken. De **List<>** met de grootste prioriteit (het grootste aantal **int**’s) wordt gekozen en de **Direction** wordt dus gezet naar de **List<>** met de grootste prioriteit.

*Een stuk uit de Move() methode uit de Elevator.cs class*

if (Direction == ElevatorDirection.UP)

{

if (Up.Count == 0 && Down.Count != 0)

{

this.Direction = ElevatorDirection.DOWN;

Move();

}

else

{

PositionY++;

for (int i = 0; i < Up.Count; i++)

{

if (Up[i] == PositionY)

{

Up.RemoveAt(i);

break;

}

}

}

}

Dit is waar het bewegen begint. Hier wordt gekeken naar de **Direction**, als de **Direction UP** is, dan wordt er gecheckt naar de **Up List<int>** om te kijken of die leeg is of niet, als dat zo is moet die namelijk switchen naar **DOWN**. Als dit niet zo is, wordt de **PositionY** met 1 verhoogt (om door de lijst van **Up** te gaan). Daarna gaat de **for** loop kijken of er een **int** (verdieping) in de **Down List<int>** staat die hetzelfde is als de huidige **PositionY** van de lift, zodat de **Down List<int>** niet twee keer langs hetzelfde punt gaat. Als de waarde is gevonden, wordt die uit de **Down List<int>** gehaald. Er is al **Distinct()** gekeken naar de **List<>**, dus het heeft geen zin meer om door te gaan zoeken naar duplicates (vandaar dat de **for** loop **break**t). Hetzelfde principe gebeurd bij als de **Direction** **DOWN** is, er wordt dan alleen ge-switched naar **UP** in plaats van **DOWN** als de **List<int>** leeg is en de **PositionY** gaat met 1 omlaag. Zoals een echte slimme lift, zal de **Elevator** dus eerst zijn **List<>** van de kant waar die opgaat leeg maken, daarna gaat die naar de andere verdiepingen. Hierdoor kunnen ook **Elevator** requests toegevoegd blijven worden aan de **List<>** van de kant waar die opgaat, en de **Elevator** zal altijd langs dat punt gaan.

# Pathfinding

In het project is Pathfinding een groot aspect. Wij hebben daarom ook goed nagedacht over hoe we dit probleem gingen oplossen. Bij ons hotel maken we gebruik van een achterliggende **Graph** met **Node**’s. **Node**’s weten welke andere **Node**’s aan hem verbonden zitten en aan welk **IArea** die verbonden is en de **Graph** weet welke **StartingNode** die heeft en een **Node[,]** (2-dimensionale **Array**) voor een overzicht van alle **Node**’s. Perfect voor Pathfinding dus, want we kunnen aan de **Graph** vragen wat de route is van de ene naar de andere **Node**. Hieronder vindt u meer uitleg over de keuzes die zijn gemaakt voor het Pathfinding gedeelte van het project.

## Graph

Na het aanmaken van de **Hotel** klasse wordt er ook een **Graph** klasse gemaakt. Dit is een **static class**, zodat andere classes binnen de **namespace HotelSimulatie** er direct bij kunnen. Deze wordt gevuld door de **CreateGraph()** methode. Deze is te lang om te laten zien, dus verwijs ik hier u om naar de code zelf te kijken en intussen dit document ernaast te houden *(ProjectHotel\HotelSimulatie\HotelSimulatie\Pathfinding)*.

## CreateGraph

De **Graph** wordt gevuld door een dubbele **for** loop die door alle **IArea**’s van **Hotel.Floors** gaat. Aan de hand van de locatie van de **IArea** (wordt gezien met de **i-** en **j int** variable) wordt er een **Node** met een bepaald **NodeType** gecreerd. De **NodeType**’s staan in een **Enum** binnen de **Node** klasse**.** Deze **Node**’s worden in een tijdelijke **Node[,]** opgeslagen om daarna in de **private static HotelNodes** te komen.

Daarna wordt de tijdelijke **Node[,]** (**hotelNodes**) uitgelezen door een nieuwe dubbele **for** loop. Deze keer worden de **Node**’s binnen de **hotelNodes** gevuld met data. Deze