# Template voor Architectuur en Ontwerpdocument

## 1 Systeemarchitectuur

De architectuur waarop het systeem is gemaakt werd aangeleverd met de GitHub Repository. Het maakt gebruik van verschillende packages en met elkaar samenwerken om het systeem te bouwen. In het compleet zijn kan het indelen in de volgende groepen:

* Kaart
  + Hierin staat alles geschreven dat de kaart bouwt waarop de algoritmes worden berekend.
  + De **Kaart,** bestaat uit **Coördinaten.** Op de **Coördinaten** staan **Terrein** van een TerreinType. Op sommige van deze **Coördinaten** staan **Steden**. Tussen deze **Coördinaten** wordt een **Pad** berekend.
* Algoritme
  + In dit Package staan de Interfaces van de te berekenen algoritmes voor de verschillende opdrachten.
* Markt en Markt.Actie
  + Deze twee package bezitten samen alles wat nodig is voor opdracht 4, het berekenen van een optimaal handelsplan.
  + Een **Handelsplan** is een lijst van **Acties.** Deze **Acties** worden bepaald vanuit een **Handelspositie.** Er wordt gehandeld op de **Markt** in Handelswaar.
* Lader en Model
  + Deze twee packages maken samen wat nodig is om de, door ons geschreven code, te mengen met de Systeemarchitectuur.
  + De **WereldLader** zorgt ervoor dat er de gemaakte **Wereld** door de **Handelaar** onderzocht kan worden.
* UI, Overlay, Util en Debug
  + Zijn alle packages met Klassen over het bekijken en debuggen van de berekende methodes.

## 2 Opdracht 2: Ontwerp (A\*-Algoritme – Geschreven door Michiel Maas)

Voor opdracht 2 van het project moest een SnelstePadAlgoritme geschreven worden. Voor twee coördinaten op de Kaart moest die algoritme het Snelste pad berekenen. Dit pad moest rekeningen houden met de terrein kosten van de verschillende Coördinaten.

Ik vond dit een hele leuke uitdaging, Een lange berekeningen die met verschillende dingen samenwerkt en zelf bedenkt wat het beste is, dit is echt programmeren!

Omdat Robin en ik allebei vorig jaar al gewerkt hebben aan dit project moesten we de Herkansing Opdracht maken. De eis aan herkansers was dat niet hetzelfde algoritme werd gekozen dat vorig jaar is gebruikt. Robin had vorig jaar gebruik gemaakt van Ant-Race, en ik zelf ben nooit zo ver gekomen om een goede uitwerking te maken van een van de Algoritmes. De keuze was dus makkelijk.

Voordat ik was begonnen met het schrijven van de code heb ik eerst veel onderzoek gedaan. De duidelijkste bronnen van deze:

* [Coding Train: A\* Pathfinder Algorithm](https://www.youtube.com/watch?v=aKYlikFAV4k)
  + In deze driedelige tutorial wordt in JavaScript een uitwerking gemaakt van het A\* Algoritme. Veel van mijn geschreven code is gebaseerd op deze tutorial.
  + [Hier is de bijbehorende GitHub Repository te vinden.](https://github.com/CodingTrain/AStar)
* [A\* (A Star) Search Algorithm – Computerphile](https://www.youtube.com/watch?v=ySN5Wnu88nE)
  + Dit filmpje gaf een duidelijke uitleg van de werking van A\*. Ik heb dit filmpje meerdere keren bekeken om het algoritme goed te kunnen begrijpen.
* [A\* Pathfinding – Sebastian Lague](https://www.youtube.com/watch?v=-L-WgKMFuhE)
  + Dit filmpje geeft een duidelijk uitleg van het algoritme, met een mooi en overzichtelijke implementatie.

Na al dit onderzoek ben ik begonnen met programmeren.

## Hoe werkt A\*?

In het kort werkt A\* als Dijkstra, maar met een extra factor: die afstand naar het eind punt.   
Voor ieder coördinaat dat het pad tegenkomt worden drie waardes berekend:

* De G-Score: De Som van de Terreinkosten van de coördinaten de bezocht worden om op dat punt aan te komen.
* De H-Score: De hemelsbrede afstand tussen het huidige coördinaat en het eind coördinaat
* De F-Score: De som van de G-Score en de H-Score.

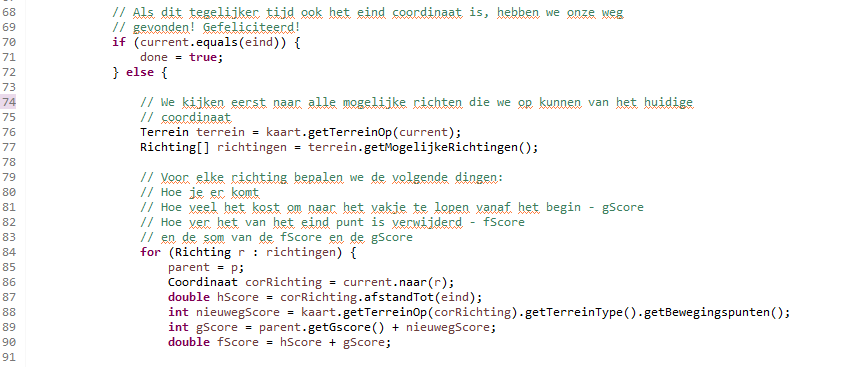
Voor elke volgende stap wordt er gekeken naar de coördinaat met de laagste F-Score. Dit coördinaat wordt onderzocht en van de coördinaten naast die coördinaten worden de verschillende scores berekend. Dit gebeurd totdat het gevonden coördinaat het Eind-coördinaat is (in welk geval je het pad hebt gevonden), of als er geen coördinaten meer zijn om te onderzoeken (in welk geval er geen mogelijke weg is te vinden tussen de coördinaten).

Om mijn code goed te laten werken heb ik gebruik gemaakt van CoordinaatHelper Klasse. Dit is een werkt als een Capsule voor Coördinaten, waar bij ik ook de verschillende Scores en het voorgaande Coördinaat bij kan opslaan. Op deze manier heb ik alle gegevens makkelijk opgeslagen.

## Eisen van opdracht 2

### Het Pad moet een begaanbare route bevatten van de Start- naar Eind coördinaten

Deze eis is erg belangrijk. Wanneer er een onbegaanbaar coördinaat op het Pad ligt, kan het eigenlijk niet worden begaan. Deze eis wordt vervuld in het volgende stuk code (regel 77 uit SnelstePadAlgoritmeImpl):

De methode *terrein.getMogelijkeRichten()* komt uit de gegeven library.



Met deze methode wordt er een lijst van Richting objecten teruggegeven, maar alleen Richting objecten die ook bezocht kunnen worden. Regel 52 vraagt: *if(t.getTerreinType().isToegangelijk()).* Als dit het geval is wordt het toegevoegd aan de lijst results, die daarna gereturned wordt. Het algoritme maakt alleen gebruik van de Richten die met deze Methode worden opgehaald, dus ook alleen met Coördinaten die bezochten kunnen worden.

Dit wordt getest in de volgende Test:



Dit is een positieve test op een kleine schaal, maar laat zien dat alle coördinaten toegankelijk zijn.

### Er moet een begaanbaar Pad berekend kunnen worden voor alle Steden combinaties

Deze Eis is moeilijk positief te testen, maar ik kon deze wel negatief testen. Wat als je geen Pad kan bereken tussen twee Steden? Voor deze eis moeten we weer kijken naar de code uit de SnelstePadAlgoritmeImpl Klasse.



Voor de loop die het pad bereken zijn er twee dingen die het draaiend houden. Als eerst is dat de controle of het bezochte coördinaat het Eind Coördinaat is. Dit gebeurt in regel 50. Zolang het Coördinaat dat onderzocht wordt niet het Eind Coördinaat is, gaat de loop door.

Wanneer het Eind Coördinaat is gevonden (regel 66), wordt de *done* variabele aangepast, en wordt de voorwaarde voor de loop beëindigt.

Het algoritme blijft zoeken zolang er mogelijkheden zijn. Als alle bezoekbare coördinaten onderzocht zijn wordt er een stop gebracht aan het algoritme met een IllegalArgumentException (regel 56).

Deze twee opties worden grondig getest door de volgende twee tests.

In de eerste test wordt voor alle mogelijke Steden-Combinaties een Pad berekend. Zodra een Pad niet kan wordt dat aangegeven.

Bij de tweede test wordt en gebruik gemaakt van een kaart waarop geen pad te maken is, er loopt een rivier tussen de twee steden. In dit geval wordt er een IllegalArgumentException gegeven, zoals verwacht.



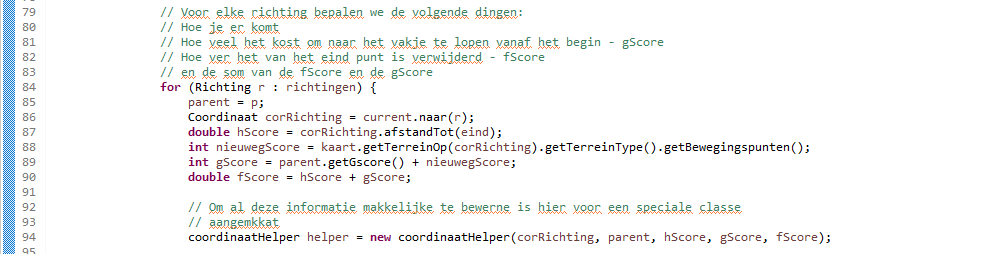
### Het pad moet het Snelste Pad zijn, niet het Kortste.

Dit was een van de grootste uitdagingen. Het gekozen Pad moet het Snelste Pad zijn, niet het Kortste. Hiermee wordt bedoeld dat het Pad moet uitgaan van de Terrein Kosten, en niet het aantal bezochte coördinaten. Het pad moet dus om een groot Bos of Bergen rug heen, als dat langer zou duren in terreinkosten.

Bij deze eis wordt er gebruikt gemaakt van de drie waardes die het A\*-Algoritme voorschrijft:

* De H-Score: De Hemelsbrede afstand van het huidige Coördinaat tot het Eind Coördinaat
* De G-Score: Som van de TerreinKosten van de Coördinaten bezocht om bij dit Coördinaat uit te komen
* De F-Score: De Som van de H-Score en de G-Score.

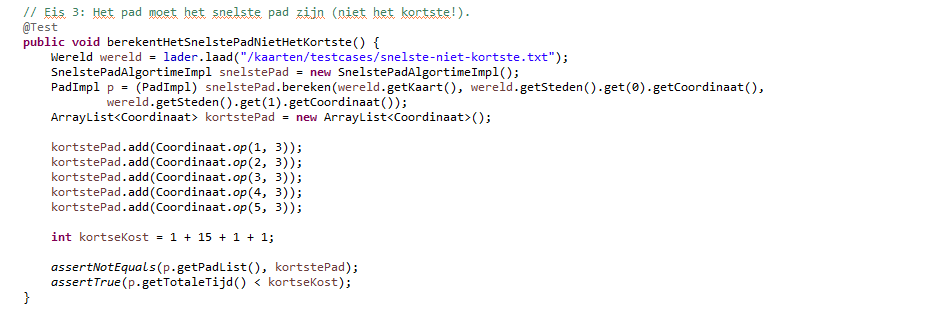
Met deze drie waardes worden beide belangen onderzocht. Zo snel mogelijk de goede kant op, maar ook niet te lang of te moeilijk. De implementatie van dit principe is te zien in de volgende code van de SnelstePadAlgoritmeImpl klasse.



Zoals je kan zien wordt er gebruik gemaakt van een coordinaatHelper. Dit is een object dat bestaat uit:

* Het Huidige Coördinaat
* De Parent, het coördinaat waar het vandaag komt (in coordinaatHelper vorm)
* De H-Score
* De -Gscore
* De F-Score

Deze eis wordt getest in de volgende testen:

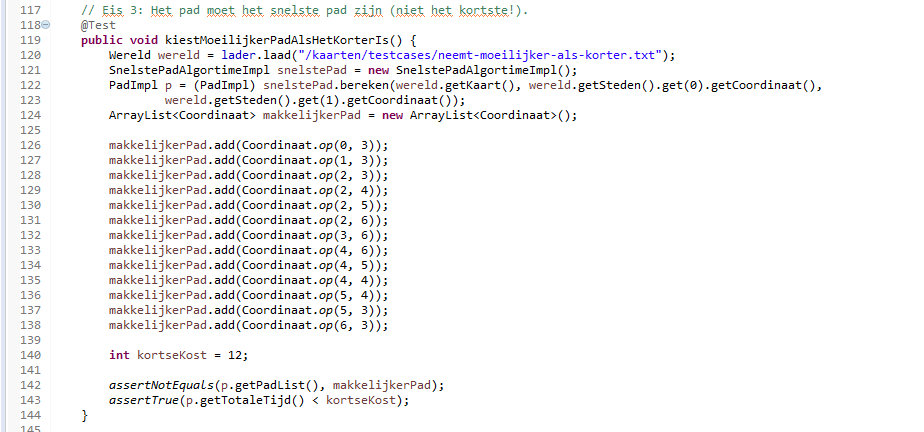


In deze test wordt er gebruik gemaakt van een Kaart met twee steden, en daar tussenin een lijn van bossen, tot aan de op een na laatste coördinaat. Voor dit pad zijn er twee mogelijkheden:

1. Het vermijden van het Bos, en de omweg nemen onderlangs
2. Recht door het Bos heen gaan en de vertraging meenemen.

In dit geval kost het Pad om het Bos heen langer, dan erdoorheen te gaan. Het Algoritme weegt beide opties aan elkaar af, en beslist uit eindelijk te gaan voor het Pad door het Bos heen, omdat deze inderdaad korter is.

De andere test toont het tegenovergestelde aan:



De Kaart gebruikt voor deze test is identiek aan die van de vorige test, maar zijn de Bossen vervangen met Bergen.

In deze situatie worden opnieuw de beide wegen opties aan elkaar afgewogen (de omweg, of over de bergen heen). Maar de bergen zijn een te grote vertraging, dus kiest het pad ervoor om de omweg te nemen, die uiteindelijk Sneller blijkt, terwijl hij dus niet Korter is.

Met deze twee testen is aan te duidelijk dat het Algoritme de verschillende opties van wegen aan elkaar afweegt, en uiteindelijk kiest voor het Snelste Pad, niet het Kortste Pad.

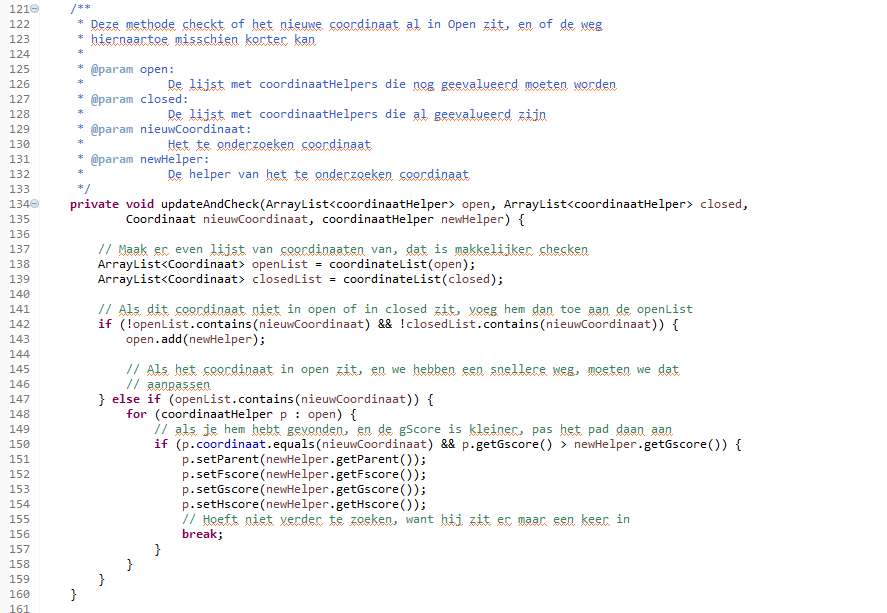
### Het Pad moet kwalitatief goed zijn

Dit is een erg globale Eis. “Het Pad moet kwalitief goed zijn” betekend veel dingen:

* Dat het Algoritme geen fouten maakt in zijn uitvoering
* Dat er in het uiteindelijke Pad geen fouten zitten
* Het correcte Pad is gevonden tussen de twee steden.

Voor deze Eis zijn meer testen geschreven dan de vorige Eis, en met een goede reden. Er kan veel fout gaan.

Ik zal de makkelijke fouten niet handelen in dit stuk, maar ik zal een specifieke situatie duidelijk toelichten: “Wat als het Algoritme een Snellere weg vindt naar een Coördinaat?”

Voor deze situatie is een speciale clausule gemaakt in het Algoritme. Zodra het pad een Snellere manier heeft gevonden naar een Coördinaat, dat hij al een keer heeft onderzocht, zal het alle gegevens aanpassen naar het Snellere antwoord. Kijk hieronder naar de code van de updateAndCheck() methode. Deze methode wordt aangeroepen voor elk coördinaat dat grenst aan het coördinaat dat wordt onderzocht. 

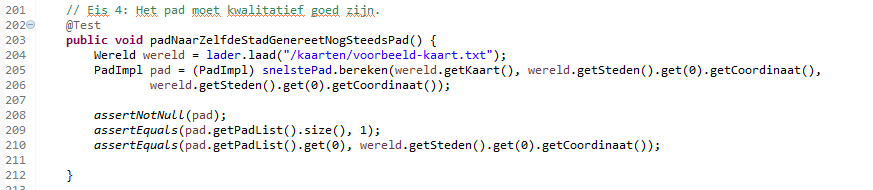
Op Regel 150 wordt met een grote If-statement gekeken of het Coördinaat dat onderzocht wordt:

1. Al in de Open List staat. Dat het coördinaat al een keer is gevonden, maar nog niet is onderzocht
2. Of de bekende weg naar dit Coördinaat langer is dan de nieuw gevonden weg

Als dit zo is, dan is er een short-cut gevonden naar dit punt. Het Algoritme kan op een snellere manier naar dit punt toe komen dan het al dacht, en moet dat dus aanpassen in de OpenList. Als het uiteindelijke Pad over dit coördinaat zal gaan, heeft het nu een snellere manier om daar te komen.

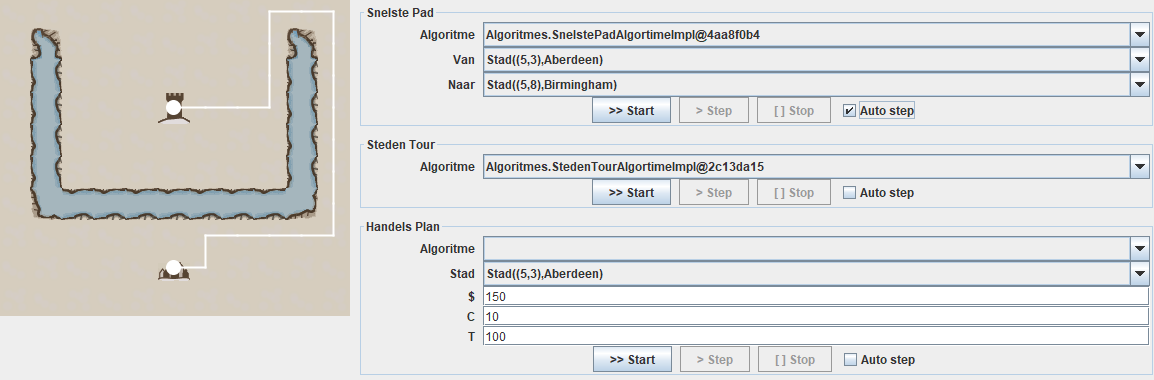
Voor alle verschillende aspecten van deze eis zijn de volgende Testen geschreven:

### Kwaliteitscontrole

Dit zijn twee eenvoudige Test die aantonen dat het Start- en Eind Coördinaten daadwerkelijk ook de gegeven Start- en Eind Coördinaten zijn.

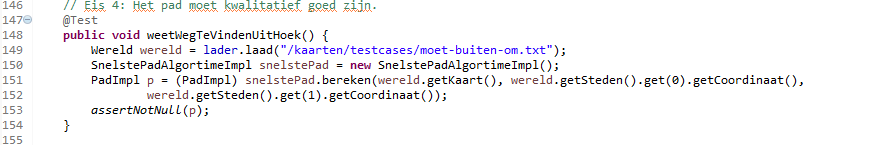
Deze Test controleert of, wanneer je een Pad berekend van een Stad naar dezelfde Stad, je wel een correct antwoord krijgt. Het kan zijn dat je per ongelijk op een knopje drukt, en dat deze berekening wordt gemaakt. Als het systeem zou crashen hierdoor, zou zonde zijn. Deze Test toont aan dat deze berekening wel degelijk goed gaat.

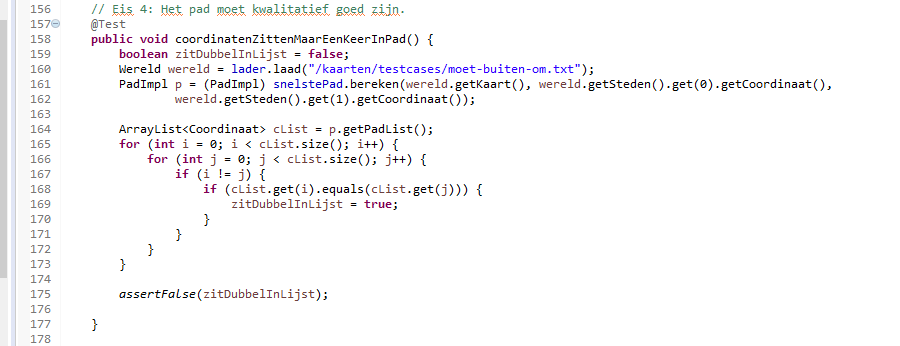
### Moeilijke Situatie

Deze Test geeft een voorbeeld van een moeilijke situatie. Op deze Kaart staan twee Steden, gescheiden door een stuk water. Zie onder:

Wanneer je een Pad probeert te berekenen tussen Aberdeen en Birmingham krijgt het Algoritme een grote uitdaging. Van nature uit wil het pad direct naar het Zuiden gaan, zo direct mogelijk naar Birmingham. Onderweg komt het dan en groot stuk, ontoegankelijk, Water tegen. Het pad probeert eindeloos mogelijkheden aan het Zuiden van Aberdeen totdat die er niet meer zijn. Uiteindelijk zal de Algoritme de overige opties nemen, en zal hij het pad vinden dat eerst naar het Noorden gaat zodat het om het grote stuk water heen kan. Dit Pad is erg onnatuurlijk, maar het algoritme weet het te vinden en gaat uiteindelijk de goede kant op.

Het feit dat het Algoritme dit pad weet te vinden, en dan ook nog de snelst mogelijke manier heeft zonder fouten, is een teken van de kwaliteit van het Pad. In de onderstaande Test wordt gekeken of dit Pad gemaakt kan worden.



De volgende Test maakt gebruik van dezelfde kaart:

Deze Test toont aan dat alle Coördinaten van het Pad maar één keer in de lijst zitten. Dit is belangrijk, anders worden Terrein’en dubbel bezocht en is de uiteindelijke som van de TerreinKosten incorrect. Om te laten zien dat dit echt niet fout gaat, wordt gebruikt gemaakt van dezelfde kaart van de vorige Test. Dit scenario heeft de grootste kans dat dit fout gaat.

### Kortere Weg

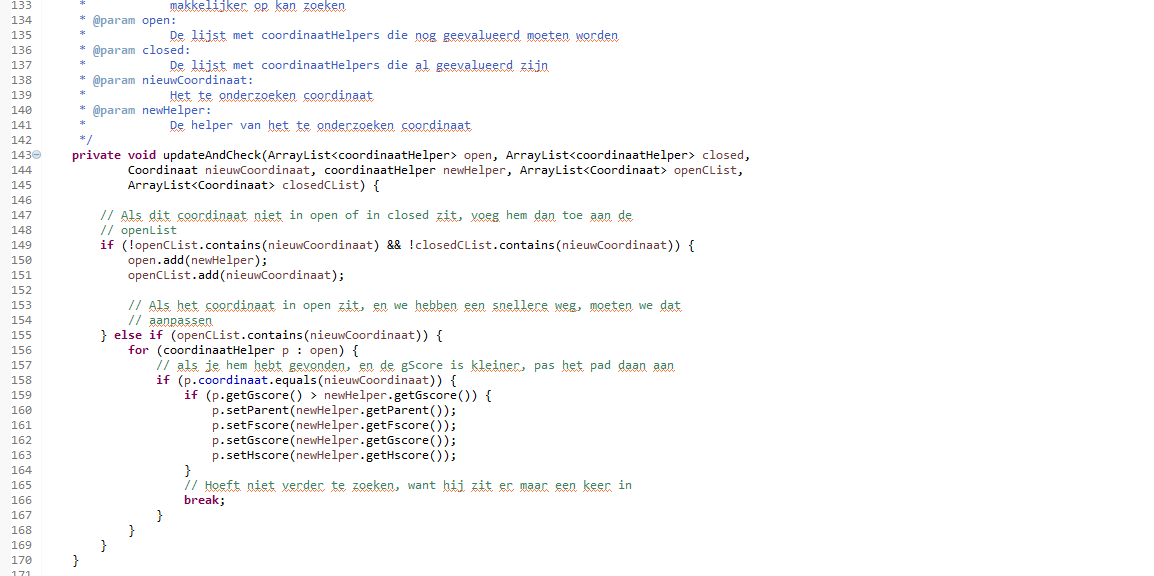
Het kan zijn dat het Algoritme op een gegeven moment een short-cut weet te vinden. Een manier om bij een Coördinaat te komen, dat minder TerreinKosten kost dan dat het eerder had uit gevonden. Als dit zo is, moet het Pad zich aanpassen en die weg volgen, het is nu eenmaal sneller. Om dit goed te laten gebeuren moet dit coordinaatHelper in de OpenList aanpassen naar de nieuwe gegevens.

Ik heb hard geprobeerd om dit test baar te krijgen, maar dit was erg moeilijk. Doordat de TerreinKosten oneven zijn, en de je alleen de Kardinale Richtingen kan gebruiken, is het (haast) ongemogelijk om een situatie te creeëren waar dit werkt.

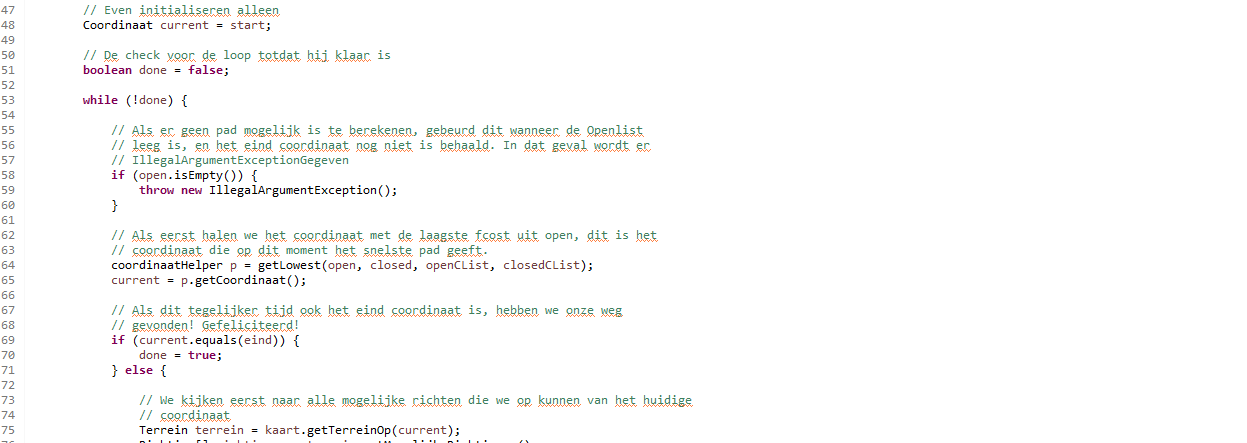
### Het Pad moet efficiënt berekend worden

Efficiëntie is heel erg belangrijk in een Algoritme. Er worden veel (dezelfde) berekeningen gedaan. Om rekenkracht te besparen wil je ervoor zorgen dat het proces zo efficiënt mogelijk gebeurd.

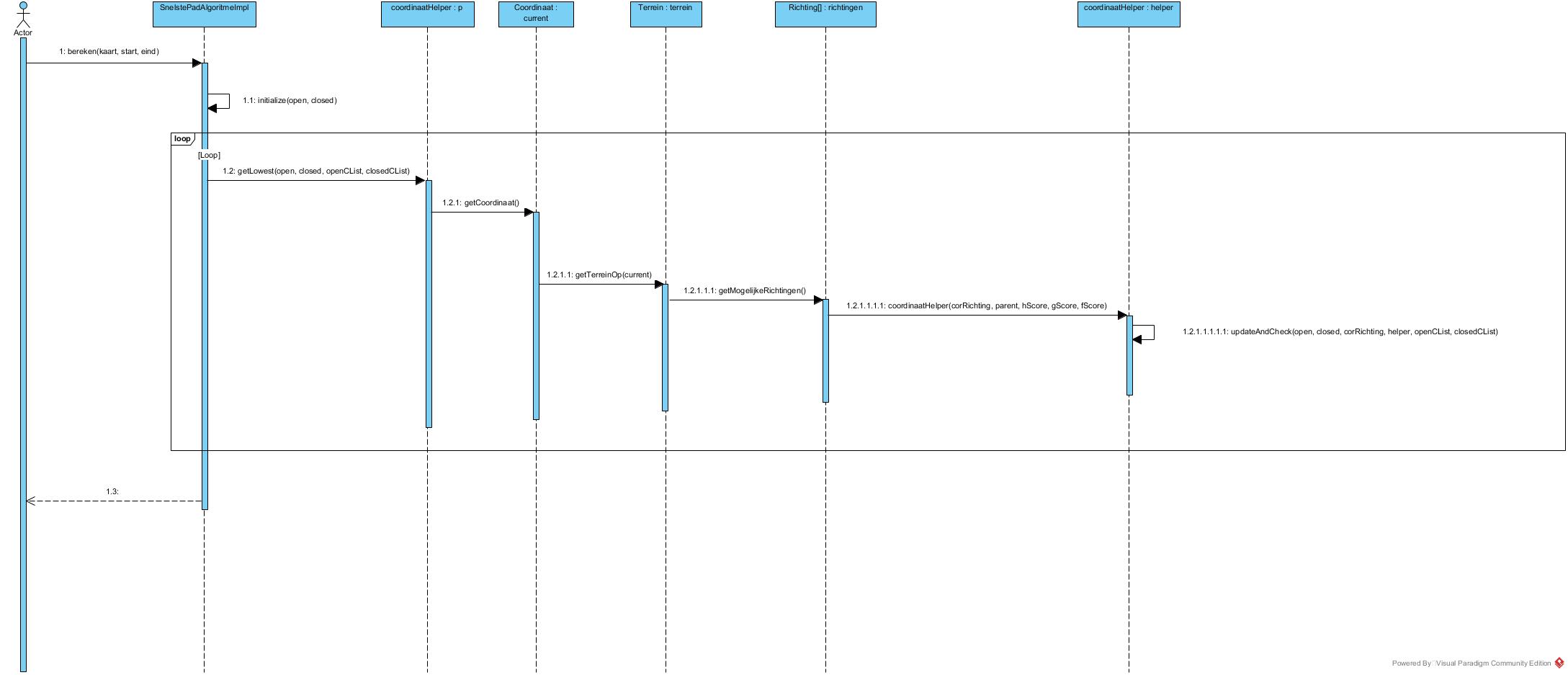
Deze is moeilijk te Testen, daarom heb ik er ook geen voor geschreven. In plaats daarvan zal ik een aantal voorbeelden laten zien van manieren die ik heb gebruikt om het proces efficiënt te laten doorlopen:

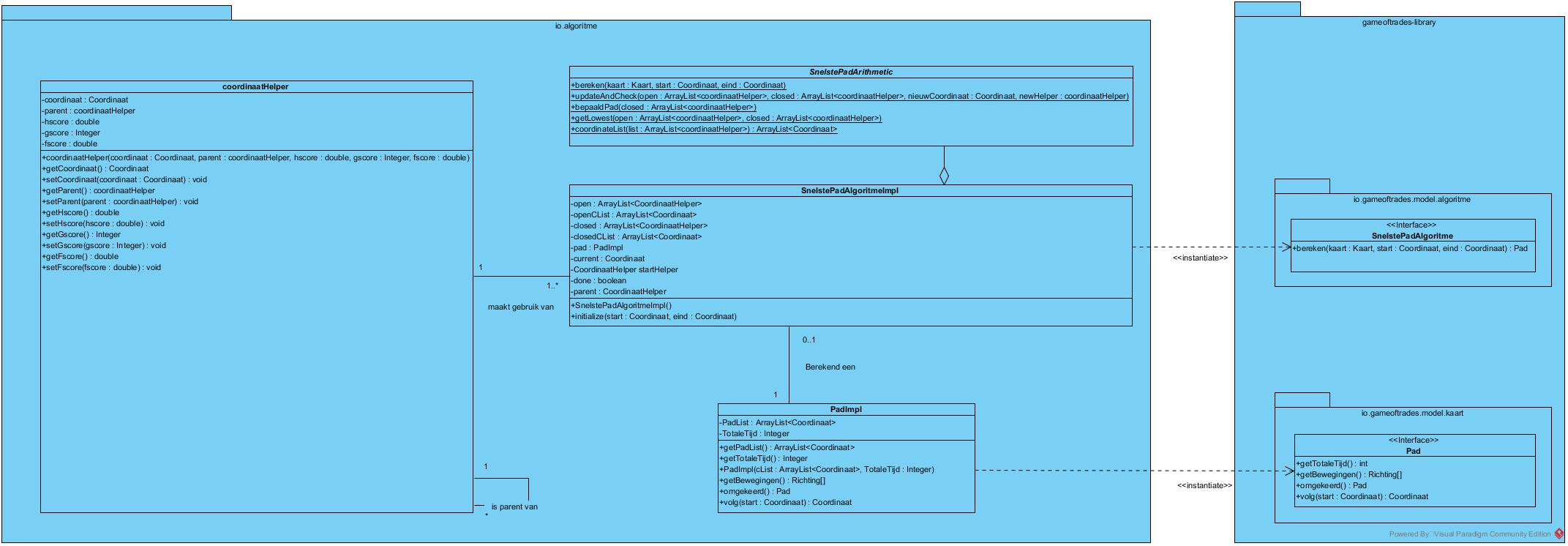


Aan dit stuk code zijn drie dingen op te merken:

1. Naast twee Array List<coordinaatHelper>, zijn er ook twee ArrayList<Coordinaat>
   1. Origineel had ik een methode die de ArrayList<coordinaatHelper> omzette naar ArrayList<Coordinaat>. Om dit iedere keer te doen koste veel rekenkracht. Ik heb er voor gekozen om naast de ArrayList<coordinaatHelper> van de *Open* en *Closed* lists, ook een ArrayList<Coordinaat> naast te hebben met dezelfde coördinaten. Dit is redelijk omslachtig, maar scheelt wel veel rekenkracht. Nu hoef ik niet telkens de ArrayLists om te zetten om er goed in te kunnen zoeken.
2. Op regel 166 staat een *break;*
   1. In dit stukje wordt de Open list doorzocht naar een coördinaat. Zodra de bijbehorende coordinaatHelper is gevonden hoef je niet meer verder te zoeken. Op deze manier doorbreek je de loop, en hoeft de rest van de lijst niet meer onderzocht te worden. Dit scheelt veel rekenkracht.
3. Er wordt gelijk al gewerkt met coordinaatHelper
   1. Voordat er wordt gekeken of de coördinaten in de *Open-* of *Closed* Lists staan, wordt er al een coordinaatHelper aangemaakt. Als blijkt dat de coordinaatHelper niet toegevoegd hoeft te worden, was de creatie van de coordinaatHelper overbodig. Ik heb geprobeerd de code aan te passen zodat deze overbodige handeling niet zou hoeven plaats te vinden, maar dit was geen succes. Dit zorgde er namelijk voor dat de code heel rommelige werd, en ik heel erg veel variabelen moest mee geven aan de updateAndCheck() methode. Dit zorgt er voor dus dat de code minder efficiënt loopt, maar de dat de code overzichtelijker is.

Een ander voorbeeld is te vinden op regel 70. Zodra bekend is welk Coordinaat onderzocht gaat worden, wordt gekeken of dat Coordinaat het Eind Coordinaat is. Als dit zo is, dan heeft het Algoritme zijn best gedaan, en het Pad gevonden. De variabele wordt aangepast en de *while loop* wordt doorbroken. Zo doet het Algoritme geen werk dat het niet hoeft te doen.

**

**