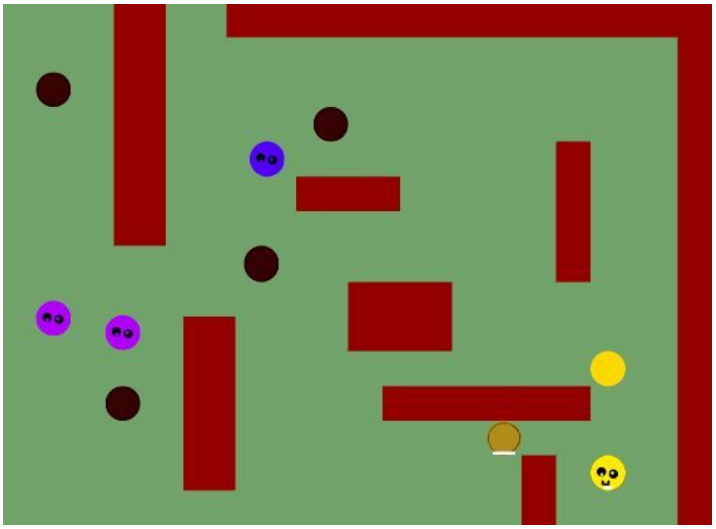
Documentatie KDEV1

Project info:



Projectnaam: Picaman
Maker: Nathan Flier
Student-info: 2de jaars game developer
Jaar: 2017

Uitleg van termen:

De speler: het gele rondje(gezicht mist nog, maar hier zou ik een pacman van willen maken
Domme AI: paars rondje met ogen
Slimme AI: blauw rondje met ogen
Vriend: geel rondje met gezicht
Totem: bruin rondje, kan overgenomen worden door slimme vijanden en door je vrienden
Val: het vage gele rondje met een wit balkje; kun je zelf ergens plaatsen met de spatie-toets; als een vijand hier opkomt en de val is actief, dan wordt deze vijand een vriend
Node: een tile in het grid; dit grid is noodzakelijk om A* pathfinding werkend te krijgen zodat de characters over een grid kunnen bewegen

Controls:

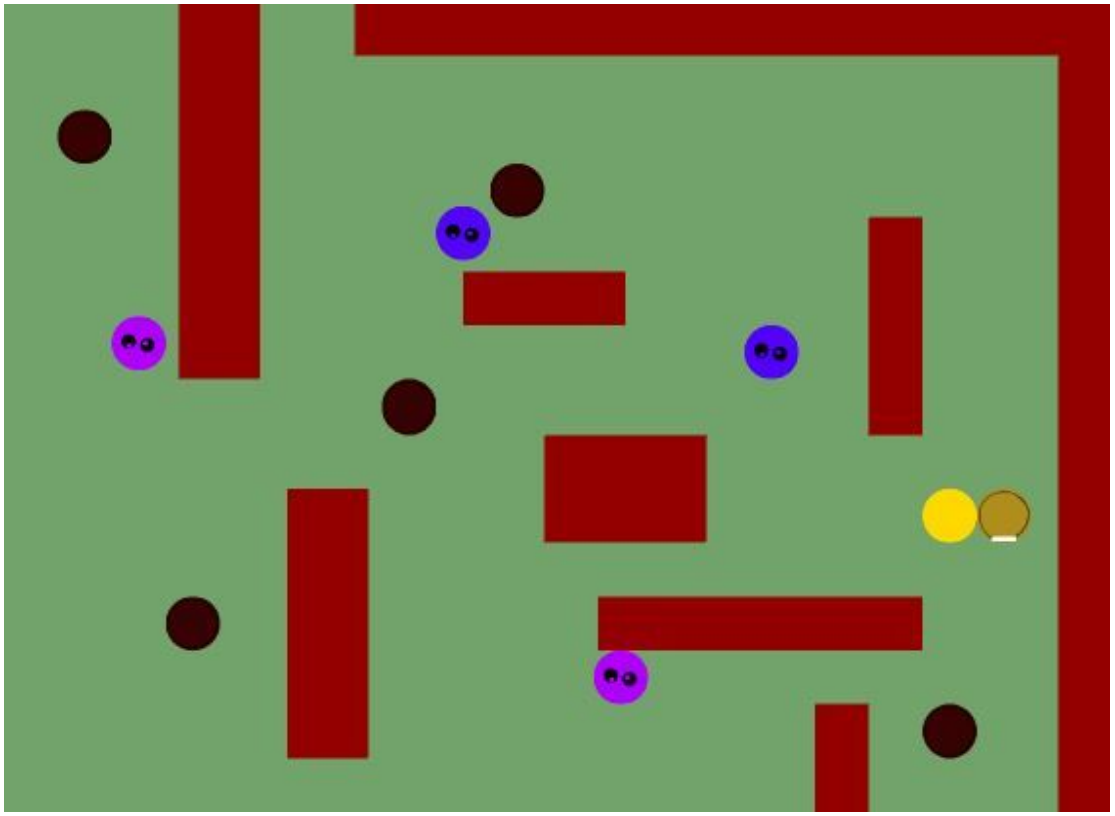
Movement: pijltjestoetsen of WASD
Ontmantelen val: spatie(mits de val actief is)
Plaatsen van val: spatie (mits de val al ontmanteld is)

Gameplay:

Het doel van de speler is om alle totems over te nemen. Dit doe je door er op te staan.
Ook heb je als doel om zoveel mogelijk vijanden tot vrienden te maken, dit bepaald namelijk de score die je haalt. Een vijand tot vriend maken doe je door een val te plaatsen en te zorgen dat een vijand er overheen loopt.
De vijanden zijn geprogrammeerd om tussen totems heen en weer te lopen. Slimme vijanden kunnen totems overnemen als deze eerder al door de speler overgenomen zijn.

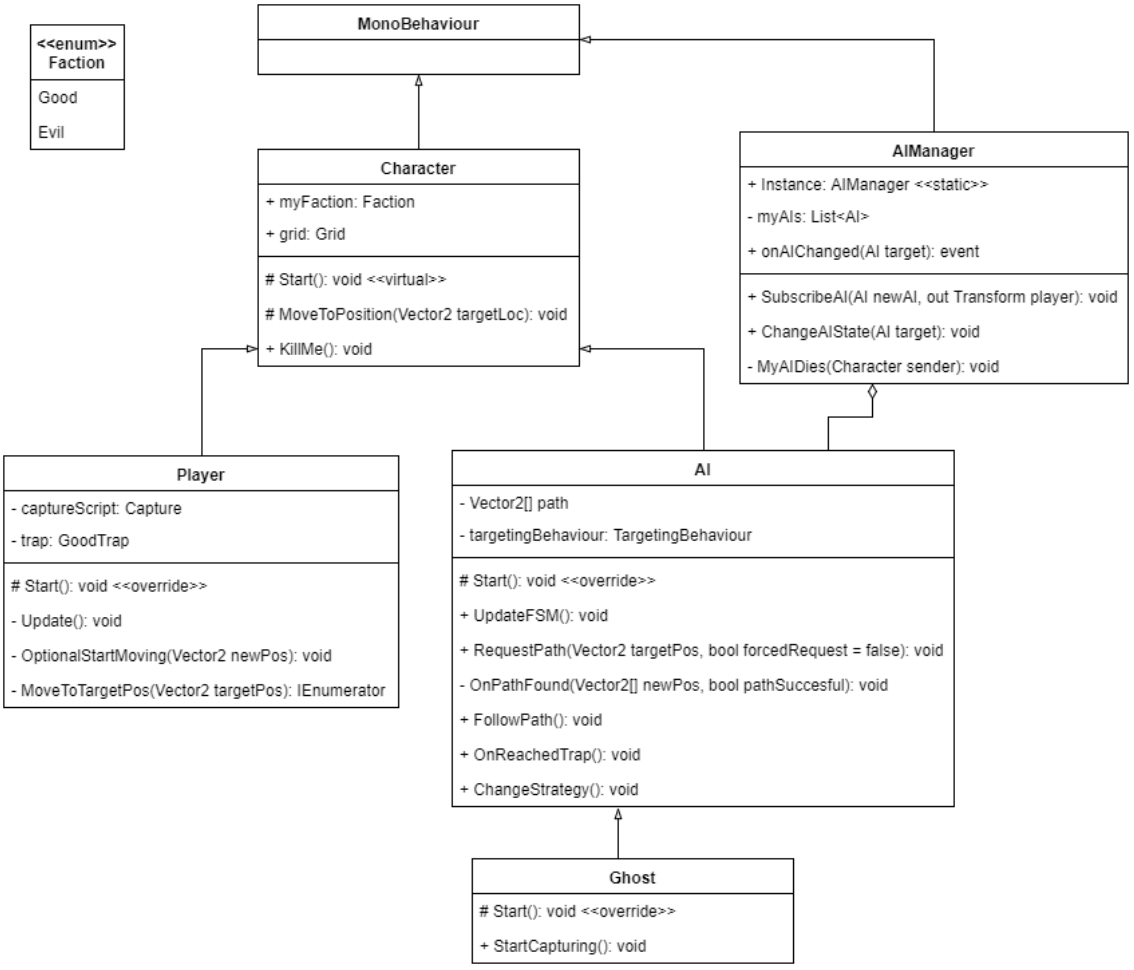
Picaman:

Picaman is gebaseerd op Pacman, maar lijkt er eigenlijk zo weinig op dat ik het haast geen twist meer zou kunnen noemen. Dit is zo gegaan doordat mijn design-ideeën mijzelf enthousiast maakten om ermee verder te gaan, maar wel steeds meer afweken van Pacman. Persoonlijk vind ik dit echter geen probleem omdat ik er wel veel van geleerd heb.



Ik zal de komende a4tjes met name besteden aan uitleggen wat de grootste uitdagingen waren waar ik tegenaan ben gelopen. Wat mij hieraan opvalt is dat het allemaal met de AI te maken heeft. Deze uitdagingen bestaan namelijk uit de Pathfinding, de Finite State Machine en het toepassen van de Strategy Pattern.

Game core UML:



Ik vind persoonlijk dat deze UML de kern van Picaman laat zien doordat het zowel de AI als de Player omvat en de structuur laat zien hoe deze game-characters in code ongeveer zijn opgebouwd.
Wat mist aan de AI is het verband met de Finite State Machine, de pathfinding en met de Strategy Pattern, maar daar zal ik later verder op in wijden.
Ik heb ervoor gekozen om zowel de speler als de AI van dezelfde class te laten inheriten omdat deze verschillende characters overeenkomend gedrag hebben.

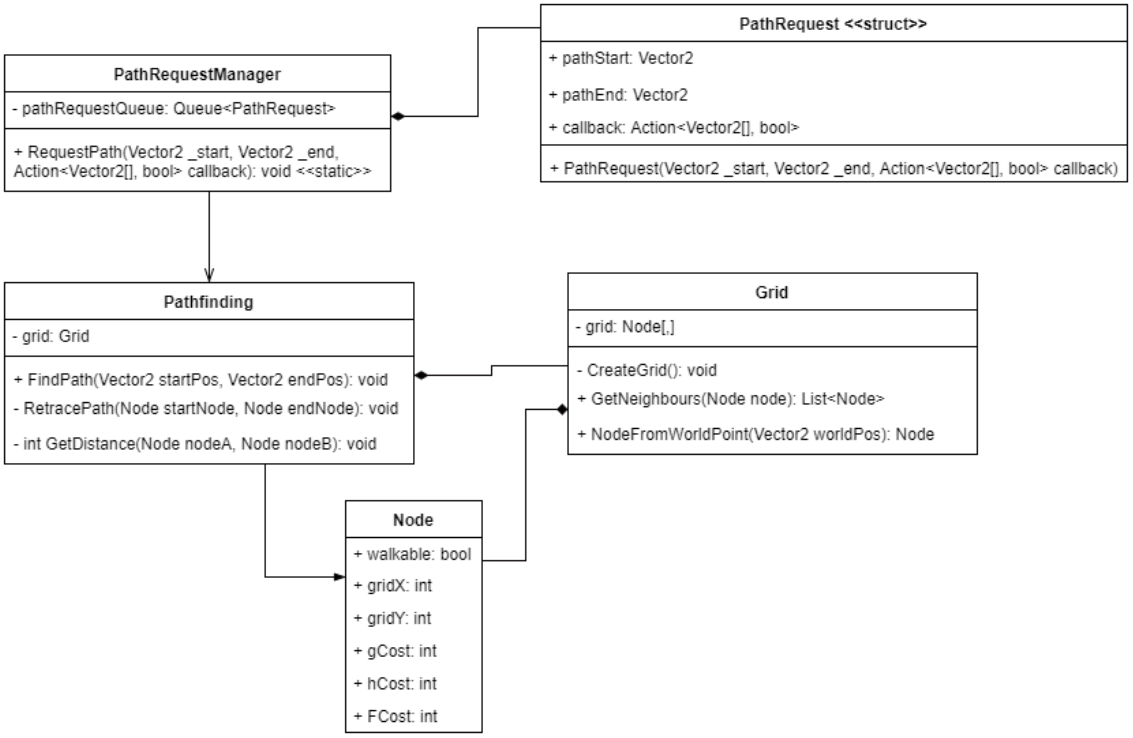
AIManager:

De AIManager is een *Singleton*. Andere scripts hebben dus alleen een reference hiernaar door AIManager.Instance aan te roepen. Dit is handig omdat ik anders voor alle verschillende AI's een referentie zou moeten zoeken naar de AIManager wat betekend dat ik heel vaak bijvoorbeeld 'GameObject.Find()' zou moeten aanroepen.

De AIManager krijgt informatie door van de AI's in het begin van het spel, als een AI een vriend wordt en als een AI dood gaat.
Deze laatste twee 'events' worden aangeroepen mbv een *Observer Pattern.*

Pathfinding:

Voor de pathfinding heb ik gebruik gemaakt van een tutorial-serie[1]. Hiermee heb ik ongeveer de volgende code-structuur gecreëerd:



PathRequestManager:

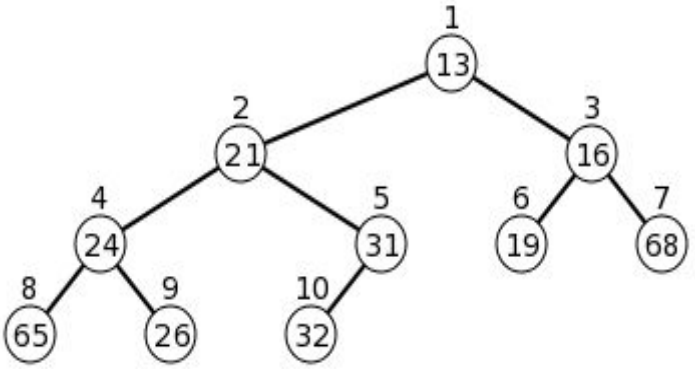
Elke AI maakt hier gebruik van om ofwel een pad te vinden naar de volgende waypoint, of om een pad te vinden naar de speler.
In dit script wordt een pad opgevraagd in Pathfinding.cs door de functie FindPath() aan te roepen. Pathfinding.cs vind vervolgens met het A* algorithme het pad naar de target-positie en geeft deze terug aan PathRequestManager. Als PathRequestManager het pad naar de target-positie gekregen heeft geeft hij deze weer terug aan de AI die dit pad opgevraagd heeft mbv. een Action-callback.
Wat ik zelf vooral fascinerend vind aan dit systeem is het gebruik van een callback: ik heb dit nog nooit eerder gezien en weet zeker dat ik hier later zeker nog gebruik van ga maken!
Ook heb ik van dit systeem geleerd waarom het handig is om in dit geval een static method te gebruiken(RequestPath is een static functie). Op deze manier hoeven de AI's namelijk niet allemaal een referentie naar PathRequestManager op te zoeken aan het begin van het spel. Zo ben ik er eindelijk achter gekomen in welke situatie het gebruik van het static keyword nuttig is.

1 Pathfinding tutorial - Sebastian Lague: https://www.youtube.com/playlist?list=PLFt_AvWsXl0cq5Umv3pMC9SPnKjfp9eGW

Heap:

In deze tutorial-serie is er ook een heap gebruikt om de volgende Node in het pad naar de target-positie te vinden. Ik heb ervoor gekozen om dit niet in het UML diagram te laten zien omdat het niet de werking van het pathfinding systeem veranderd, maar ‘slechts’ een efficiëntere wijze is van de volgende Node in het pad naar de target-positie vinden.
Hierdoor heb ik dus geleerd wat een heap is en hoe dit werkt. Het idee van A* is namelijk dat je continu de waardes van aangrenzende Nodes vergelijkt en steeds de Node met de laagste hCost als volgende Node in het pad evalueert.

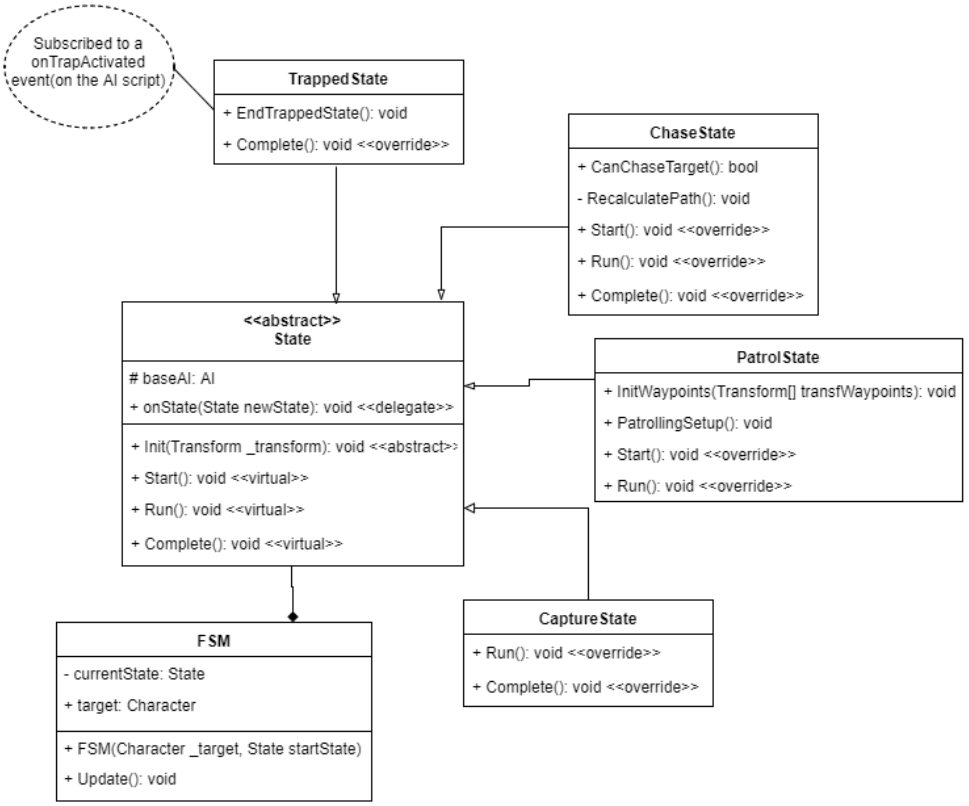
Voorbeeld van een heap:



Finite State Machine

Ik heb ervoor gekozen om voor mijn AI een Finite State Machine te gebruiken.
Voordelen van de *Finite State Machine* in mijn project zijn:

- helpen om de code gescheiden te houden
- ik voorkom ermee dat een AI als het ware in twee states tegelijk kan zijn
- het is een handige manier om gedrag toe te wijzen aan AI's



State toewijzing:

Elke AI houdt een referentie naar de FSM en roept elk frame hier void Update() op aan.
Ik heb de States als volgt ingedeeld:

Basis states:

States die elke AI heeft. Deze states kunnen daarom ook met elkaar communiceren.

- ChaseState: de AI zit achter een target aan (is de speler of een totem)
- PatrolState: de AI patrolled tussen een aantal waypoints

Overige states:

States die toegewezen kunnen worden aan een AI. Van deze states wist ik niet zeker of ik deze mogelijk in latere fases van het ontwikkelproces aan iedere AI wou toewijzen of dat dit per AI zou kunnen verschillen.

- TrappedState: de AI zit vast in een trap van de speler, aan het einde van deze state wordt deze AI een 'vriend'
- CaptureState: de AI is bezig met een totem overnemen

Implementatie:

Ik heb 2 verschillende AI's gemaakt, de domme AI en de slimme AI:
De slimme AI(Ghost compnent, Capture component) heeft alle states.
De domme AI(AI component) heeft geen CaptureState en kan dus niet totems overnemen.

Wat ik hiervan geleerd heb:

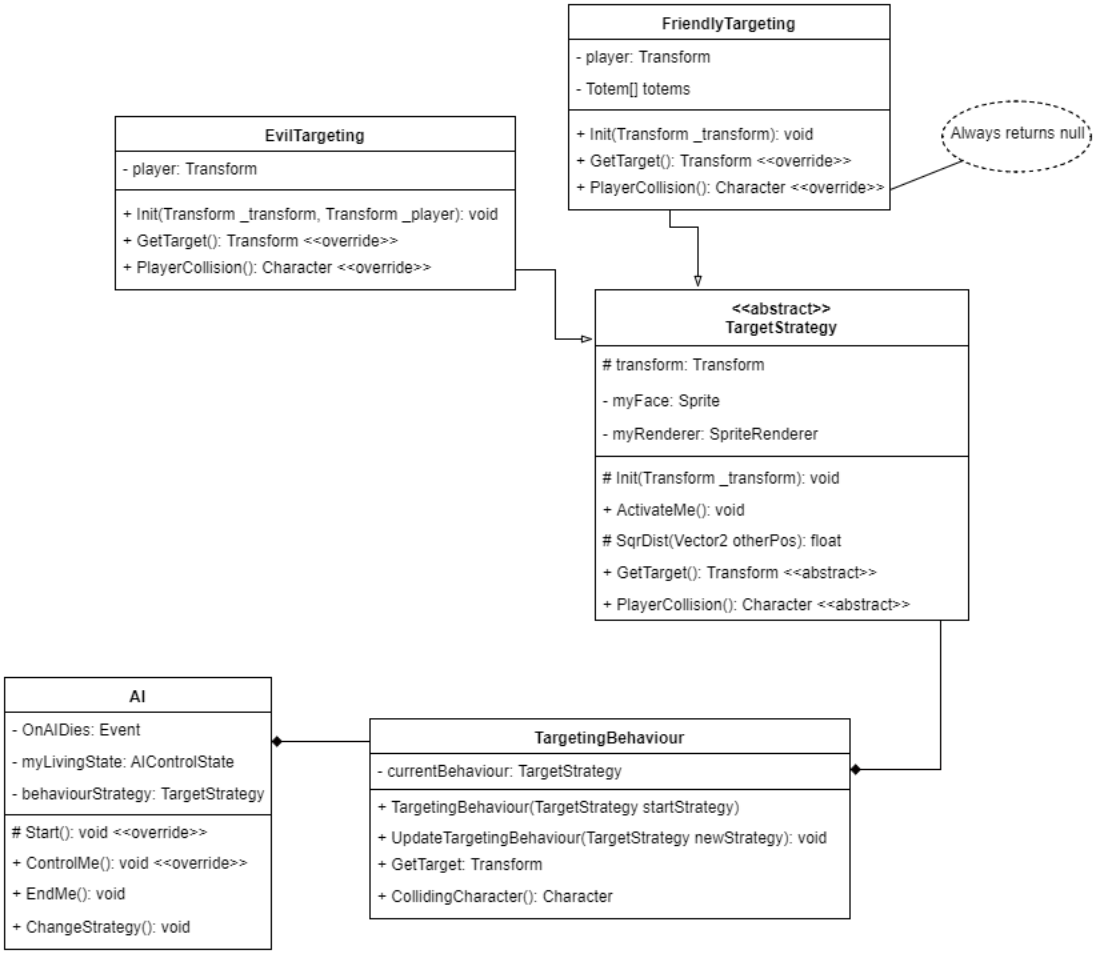
Ik heb hiervoor nog nooit eerder op zo'n abstracte manier met state machines gewerkt. Dit was leerzaam voor mij omdat ik nu beter begrijp waarom de states zou abstract mogelijk zouden moeten zijn en omdat ik nu weet hoe ik dit op een slimme manier op kan zetten.

Uitbreidmogelijkheid:

Ik zou zelf heel graag nog een boss maken. Deze zou dan ook gebruik maken van de Finite State Machine, maar zou geen totems capturen en zou nog een extra state hebben: een SpecialMoveState. Op deze manier zou ik naar mijn mening vrij eenvoudig een AI kunnen toevoegen die de diversiteit enorm vergroot. Hierdoor zou de speler het spel als spannender en zelfs leuker kunnen ervaren.

Strategy Pattern:

Voor mijn AI's wou dat ze zowel vijandig als vriendschappelijk gedrag zouden kunnen vertonen. Voor mij was het logisch dat ik hier een *Strategy Pattern* voor zou gebruiken omdat dit design pattern ervoor zorgt dat gedrag makkelijk vervangen kan worden.



Zoals hierboven te zien is heeft de AI een referentie naar de TargetingBehaviour class. De speler gebruikt deze aan het eind van de TrappedState door de volgende code aan te roepen:

```
public void ChangeStrategy(){
        patrolState.InitWaypoints (friendlyTargeting.patrolPoints);
        targetingBehaviour.UpdateTargetingBehaviour (friendlyTargeting);

        onTrapActivated ();
        trap = null;

        GetComponent<SpriteRenderer> ().color = Color.yellow;

        AIManager.Instance.ChangeAIState (this);
}
```

Door de regel ‘targtingBehaviour.UpdateTargetingBehaviour (friendlyTargeting);’ wordt het gedrag veranderd. Vanaf nu zoekt deze AI niet langer naar de speler als target maar probeert deze AI totems voor de speler te capturen (mits deze AI gebruik maakt van de CaptureState) en kan deze AI de speler niet meer af laten gaan.

Verklaring van afwijking ten opzichte van de geplande UML:

Ten opzichte van wat ik eerst als UML bedacht had heb ik nu ongeveer de UML structuur waar ik oorspronkelijk ook van plan was om op uit te komen. Het grootste verschil zit hem erin dat ik nu niet de Character gebruik laat maken van de FSM, maar de AI wel. Hier heb ik voor gekozen omdat het gedrag van de AI qua code veel gecompliceerder is dan dat van de speler. Bij de speler leek het mij simpelweg niet nuttig. Als voordeel heeft deze aanpak ook dat ik de States iets minder abstract op kon zetten waardoor ik meer gedrag van de AI’s binnen de States kon zetten. Dit heeft mij erbij geholpen om de functionaliteiten meer gescheiden te houden.

Wat ik een volgende keer anders wil aanpakken:

Waar ik tegenaan liep bij het opzetten van de Finite State Machine was dat het naar mijn mening toch voor de AI’s niet abstract genoeg was. Ik heb te veel referenties naar andere scripts moeten gebruiken om dit werkend te krijgen. Daarom hoop ik dat ik dit een volgende keer beter op zou kunnen zetten. Ik hoop dit te doen door op een consistente wijze de States enkel functies binnen één target-script uit te voeren.

Ook zou ik graag me verdiepen in de IEnumerator-versie van de Finite State Machine omdat dit naar mijn mening kan helpen om de code overzichtelijker te houden. Ik heb hier wat van gezien online en het lijkt mij een goed alternatief omdat het op deze manier meer lexicale code is dan de Finite State Machine. Oftewel: de volgorde waarin het getyped staat lijkt meer op de volgorde waarin het uitgevoerd wordt.