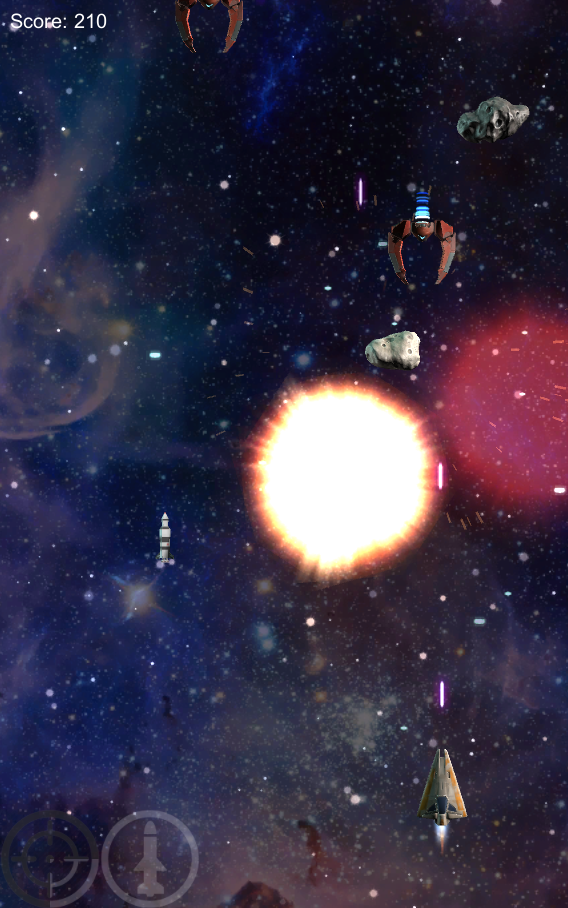
**Technisch** **Uitwerkingsplan  
& Documentatie**



Quincy Smulders

Hogeschool voor de Kunsten Utrecht  
Game Development

26 / 10 / 2018

Inhoud

[Plan van Aanpak 3](#_Toc528285138)

[Engine 3](#_Toc528285139)

[Risico-analyse 3](#_Toc528285140)

[Wat zijn de risico’s? 3](#_Toc528285141)

[Wat zijn de maatregelen hiertegen? 3](#_Toc528285142)

[Wat zijn de prioriteiten? 3](#_Toc528285143)

[Design Patterns 3](#_Toc528285144)

[Geïmplementeerde patterns 4](#_Toc528285145)

[Object Pool 4](#_Toc528285146)

[Geplande patterns 4](#_Toc528285147)

[State Machine 4](#_Toc528285148)

[Decorator 4](#_Toc528285149)

[Unified Modeling Language 5](#_Toc528285150)

[Activity Diagram 5](#_Toc528285151)

[Class Diagram 6](#_Toc528285152)

[Uitleg 7](#_Toc528285153)

[Activity Diagram 7](#_Toc528285154)

[Class Diagram 7](#_Toc528285155)

[Review 7](#_Toc528285156)

# Plan van Aanpak

## Engine

Als Engine ben ik van plan om Unity te gebruiken. De reden hiervoor is dat ik mij in dit programma wil verdiepen en de bijbehorende script taal, namelijk C#. Ook sluit het goed aan op de lessen die tevens Unity als voorbeeld gebruiken.

## Risico-analyse

### Wat zijn de risico’s?

Omdat ik last-minute heb moeten kiezen of ik Design of Development wou doen liep ik al meer dan drie weken achter op de rest van de klas. Ook heb ik al lang geen code meer geschreven dus zal ik dit flink moeten bijspijkeren om niet nog meer achter de feiten aan te gaan lopen. De voornaamste risico’s zijn voor mij een spel dat geen gameplay heeft en een spel dat niet kan runnen door bugs of errors.

### Wat zijn de maatregelen hiertegen?

Ik ga dit blok vooral kleine en makkelijke dingen implementeren door middel van het volgen van verschillende tutorials van Unity. Mijn beredenering hiervoor is dat je beter iets kleins kan hebben dat werkt, dan allemaal grootste plannen schetsen die je uiteindelijk niet kan uitvoeren. Zodra ik iets een spel heb dat werkt kan ik hier langzaam op uitbouwen en itereren.

### Wat zijn de prioriteiten?

Mijn prioriteit is vooral een werkend spel op te leveren. De prioriteiten zullen hier dan ook vooral liggen bij de core-features van het spel. De top drie, in geen specifieke volgorde is als volgt:

1. Een werkende player controller.
2. Een doel in het spel; in dit geval punten scoren.
3. Het kunnen behalen van punten. Dit wordt bereikt door middel van wapens en enemies.

Verder wil ik minstens één van de design patronen toepassen, maar het liefst meerdere indien mogelijk.

# Design Patterns

Bij het re-design van een retro-game komen verschillende dingen kijken. Ten eerste is de uitdaging van de opdracht een retro-game maken met een twist. Als spel heb ik gekozen voor Galaga, de simpele Top-Down Shooter. De twist die ik bedacht had was een speler met verschillende *states* door middel van toenemende en sterker wordende *abilities*. Vervolgens ben ik verschillende *design patterns* gaan onderzoeken om te kijken welke het beste bij deze eigenschappen pasten.

## Geïmplementeerde patterns

### Object Pool

#### Wat is Object Pooling?

Wanneer je bepaalde objecten, waarvan je weet dat je ze nodig gaat hebben, al laat instantiëren spreekt men van *object pooling*. In plaats van nieuwe objecten te creëren en oude te vernietigen tijdens het spelverloop, hergebruikt het spel deze objecten uit een *pool*.

#### Waarom Object Pooling?

*Object Pooling* kan zorgen voor een flinke performance boost. Wanneer bepaalde *classes*, zoals bijvoorbeeld een *minigun*, veel objecten moeten aanmaken om vervolgens weer te slopen is het qua performance slimmer om de objecten, in dit geval kogels, te recyclen. Unity maakt gebruik van Garbage Collection om geheugen weg te zetten dat niet meer gebruikt wordt. De Destroy() functie brengt dit op gang en zorgt ervoor dat de CPU vertraagd en dit kan *lag-spikes* veroorzaken. Het is dus beter om dit te vermijden.

## Geplande patterns

### State Machine

#### Wat is een State Machine?

Een State Machine is een abstract model voor het gedrag van een systeem. Hier is sprake van wanneer een model bestaat uit verschillende toestanden en / of acties en de overgangen hiertussen.

#### Waarom State Machines?

Dit design pattern zorgt ervoor dat een object zijn gedrag kan veranderen wanneer zijn interne state veranderd. Zo lijkt het object zijn *class* te veranderen. Dit wil ik gaan gebruiken om een evoluerende *enemy spawner* te maken, terwijl het spel al draait, door aan de verschillende *states* extra en / of nieuwe *enemies* mee te geven.

### Decorator

#### Wat is een Decorator?

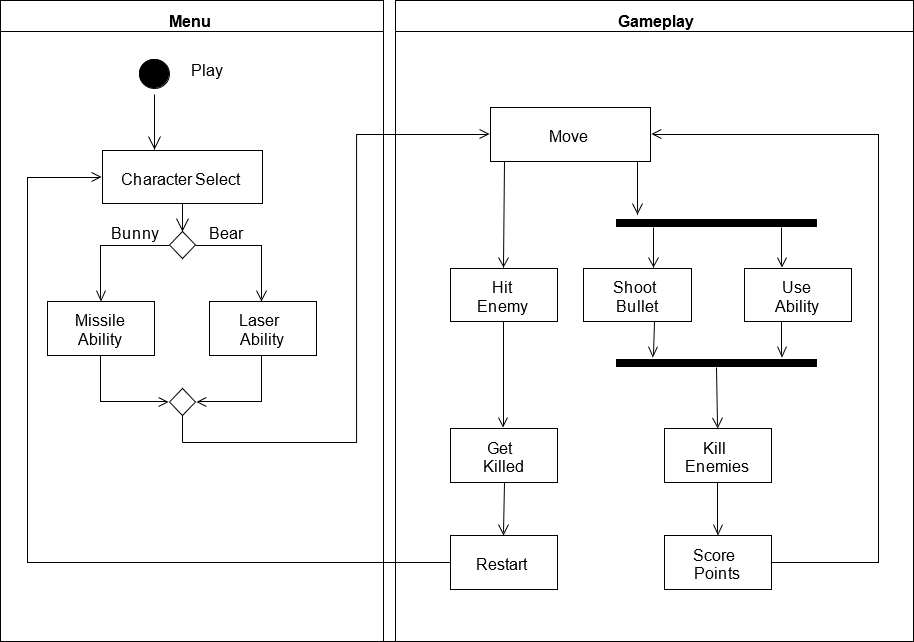
Met een decorator kan je extra of aanvullende verantwoordelijkheden dynamisch aan een object vastmaken. Dit zorgt voor een flexibel alternatief voor het uitbreiden van de functionaliteit van *subclasses*.

#### Waarom een Decorator?

Om de speler uiteindelijk meerdere eigenschappen te geven terwijl het spel al draait. Bijvoorbeeld door middel van power-ups die in een willekeurige volgorde voorkomen.

# Unified Modeling Language

## Activity Diagram



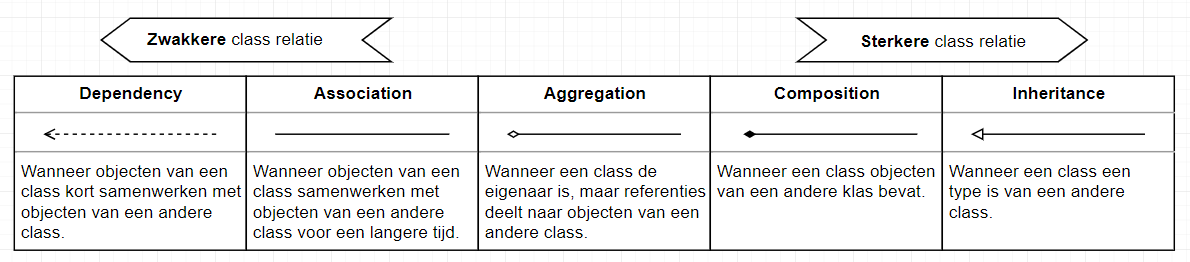
## Class Diagram

## Uitleg

### Activity Diagram

Dit UML-diagram volgt de acties die een speler kan maken. Zo begint het spel wanneer de speler een character kiest. Dit wordt aangeduid doormiddel van een ruit en een splitsend pad dat vervolgens weer bij elkaar komt. Terwijl de speler beweegt ontstaat er een loop. Deze begint met het parallelle gebruik van schieten en het gebruik van de secondary fire. Hiermee kan de gebruiker enemies verslaan, indien hij/zij raak schiet, en punten verdienen. De loop wordt echter verbroken wanneer de speler een vijand raakt. Dit resulteert in een death en beëindigd het spel. Vervolgens kan het spel herstart worden en begint het weer van voor af aan.

### Class Diagram

[[1]](#footnote-1)

Van onder naar boven, Laser Ability, Projectile Ability en Bullet Ability zijn typen van Ability. Dit is een *generalization* wat ook wel bekend staat als *inheritance*. Een Ability bestaat uit een Ability Cooldown. Als de Ability verwijderd zou worden heeft dit gevolgen voor de overeenkomstige delen. Omdat de Ability Cooldown hierdoor niet meer zou kunnen bestaan spreken we hier van een *composition*. Een stapje terug in de *class relationship line* is de *aggregation*. Zo bezit de speler verschillende abilities. Toch zou de Player Controller nog steeds kunnen werken zonder de Ability class.

Bij de rest van de pijlen spreken we van een *dependency*. Dit houdt in dat de Player Controller iets moet weten van het Destroy By Contact script, maar niet altijd. Het is de zwakste directe relatie tussen classes en het impliceert dat de twee samen *kunnen* werken. Ik twijfelde hier tussen *dependency* en *association*, maar ben uiteindelijk toch voor de eerste optie gegaan. De reden hiervoor is dat de classes niet constant met elkaar samenwerken, maar alleen wanneer ze uit de boundary gaan of wanneer ze elkaar aanraken.

### Conclusie

In het UML-diagram komt het niet erg naar voren, maar een probleem waar ik al redelijk vroeg op stuitte is het feit dat in het Destroy By Contact script de relaties tussen zowel enemies, kogels en spelers beschreven moest worden. De volgende keer zou ik deze functionaliteit liever bij de kogels leggen. Hierdoor zouden er slechts twee referenties nodig zijn, namelijk tussen player bullets en enemies, en tussen enemy bullets en de player.

1. R. Miles & K. Hamilton (2008) *Learning UML 2.0*. O'Reilly Media. [↑](#footnote-ref-1)