Functioneel Ontwerp Sugar Wars

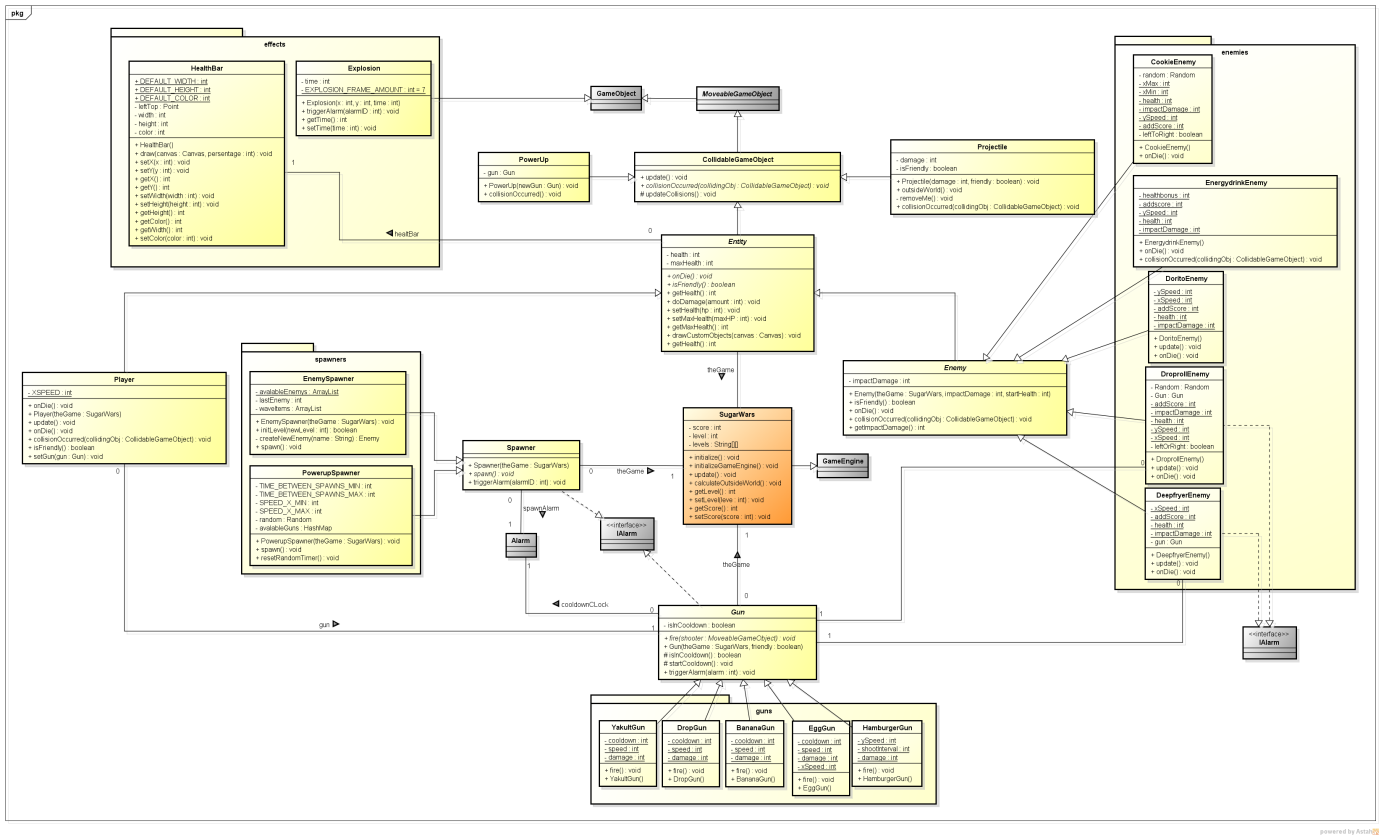
Door Stefan Mudde en Sijmen Huizenga  
16-03-2015

# Voorwoord

Dit is het technisch ontwerp voor de game SugarWars. In dit technische ontwerp worden de verschillende klasse besproken. Deze klasse worden in een klasse diagram weergegeven. Daarna worden de belangrijkste klasse apart besproken.

# Klasse Diagram

Dit is het klasse diagram van ons project:



De High Quality versie van dit klasse diagram is te vinden in “class diagram.png”.

# Veranderingen Game Engine

### Buttons Verwijdert

Als eerste zijn een stel buttons uit de onscreenbuttons.xml verwijdert. Alleen de button a, linker pad en rechter pad zijn over gebleven. Daarna is de code binnen OnScreenButtons.java ook aangepast zodat alle verwijzingen naar de verwijderde buttons ook niet meer bestaat.

Deze verandering was nodig omdat wij de OnScreenButtons.disableButton() niet aan de praat kregen. Het verwijderen van de buttons in de game engine leek ons de makkelijkste oplossing.

### Protected CalculateOutsideWorld

De GameEngine bevat de methode CalculateOutsideWorld. Deze methode wordt gebruikt om te controleren of een GameObject buiten de wereld bestaat. Als een object buiten de wereld valt, dan wordt de methode outsideWorld() aangeroepen. De definitie van ‘buiten de wereld’ volgens de originele implantatie is: ’1 of meer pixels zijn buiten de border van de wereld’. In ons geval wilde we deze definitie aanpassen zodat de functie outsideWorld() pas wordt aangeroepen als het volledige object buiten de wereld is.

De GameEngine staat het niet toe deze methode te overschrijven. Door de visibility van deze methode te veranderen naar protected kan SugarWars deze methode toch overschrijven.

### getOnScreenButtonsView()

Binnen de klasse OnScreenButtons is de statische methode getOnScreenButtonsView toegevoegd. Deze methode geeft het view van onScreenButtons.xml terug. Dit view is normaal gesproken niet toegankelijk voor game makers, maar in ons geval wilde we wel speciale toegang tot dit View.

Ons dashboard (level en score) hebben we toegevoegd aan de onscreenbuttons.xml. Op deze manier kan alle lay-out code in het xml bestand staan. Dit zorgt weer dat je live in de editor het resultaat van de lay-out kan zien. Hierdoor konden we het dashboard heel mooi positioneren.

Op de zelfde manier is het stukje tekst ‘GameOver’ toegevoegd. Er is een TextView in het midden van onscreenbuttons.xml toegevoegd. Dit TextView is helemaal gestileerd binnen het xml bestand en daarna op ‘invisable’ gezet. Op het moment dat de speler dood gaat, hoeft alleen dat textView op ‘visable’ gezet te worden.

Om het dashbaord te updaten en het gameover textview zichtbaar te maken, moet er toegang zijn tot het View waarin deze objecten bestaan. Dit view is te vinden in OnScreenButtons. Aan de hand van deze getter kan ook op andere plekken toegang worden verkregen tot dit view.

# Klasse

## SugarWars

Dit is de game klasse van het spel. Deze is een extensie op GameEngine en is de baas over de game. Hier wordt alles aangestuurd. Hier worden onder andere de Spawners aangemaakt.

## CollidableGameObject.

In het spel zijn er veel acties die moeten gebeuren als twee MoveableGameObjects elkaar raken. MoveableGameObject heeft een methode die hierbij kan helpen: ‘getCollidedObjects’. Deze geeft een lijst met alle botsende GameObjects. De code zou in het algemeen de volgende acties uitvoeren:

1. Is de lijst met botstende GameObjects null en dus leeg?
   1. Ja: doe nis
   2. Nee: ga naar stap 2
2. Ga langs elk botstend GameObject en
   1. Test of het GameObject een instantie is van de gezochte klasse
      1. Ja: doe de actie
      2. Nee: ga door met stap 2

### implementatie

Bovenstaande programmeer ontwerp kan je op meerdere manieren implementeren.

### Zelf uitzoeken

Als eerste optie is om de klasse het zelf uit te laten zoeken. Dit betekent dat dat elke klasse (Enemy, Player, Projectile, Upgrade) zelf een manier bedenkt om de bovenstaande code uit te werken. Dit zorgt voor herhalende code.

### Functie

Een andere optie zou zijn om een functie te maken:

public static ArrayList**<**GameObject**>** searchCollision**(**MoveableGameObject center**,** Class**<?** **extends** GameObject**>** toSearchClass**){**

ArrayList**<**GameObject**>** found **=** **new** ArrayList**<>();**

ArrayList**<**GameObject**>** collidingObjects **=** center**.**getCollidedObjects**();**

**if(**collidingObjects **==** **null)**

**return** **null;**

**for(**GameObject obj **:** collidingObjects**){**

**if(**obj**.**getClass**().**equals**(**toSearchClass**)){**

found**.**add**(**obj**);**

**}**

**}**

**return** found**;**

**}**

Voordeel hiervan is dat er maar één methode nodig is. Je hoeft geen klasse of interface aan te maken. Nadeel hiervan is dat je “Class<? extends GameObject>” gebruikt, wat niet is behandeld in de lessen.

### Interface

De volgende optie bestaat uit een interface met een generic type. Deze zou er als volgt uitzien.

public interface ICollision**<**K **extends** GameObject**>** **{**

public void collidesWith**(**GameObject collidingObj**);**

**}**

In de main update binnen GameEngine zou dan langs alle GameObjects geloopt kunnen worden, en dan kunnen kijken welke ICollision implementeren. Daarna door middel van GameObject.getCollidedObjects kijken welke botsen. Deze worden gefilterd door het Generic Argument . Het gebotste object wordt doorgegeven aan het ICollisoin object. Dit ziet er als volt uit:

public void checkCollisions**(){**

**for(**GameObject toCheck **:** items**){**

**if(!** **(**toCheck **instanceof** MoveableGameObject**))**

**return;**

MoveableGameObject toCheckMovable **=** **(**MoveableGameObject**)** toCheck**;**

**if(**toCheckMovable **instanceof** CollidableGameObject**.**ICollision**){**

CollidableGameObject**.**ICollision**<?** **extends** GameObject**>** originalObj **=** **(**CollidableGameObject**.**ICollision**<?** **extends** GameObject**>)** toCheck**;**

Type type **=** **((**ParameterizedType**)** originalObj**.**getClass**().**getGenericSuperclass**()).**getActualTypeArguments**()[**0**];**

toCheckMovable**.**getCollidingObject**();**

ArrayList**<**GameObject**>** collidingObjects **=** toCheckMovable**.**getCollidedObjects**();**

**if(**collidingObjects **==** **null)**

**return;**

**for(**GameObject obj **:** collidingObjects**){**

**if(!(**obj **instanceof** MoveableGameObject**))**

**continue;**

**if(**obj**.**getClass**().**equals**(**type**)){**

originalObj**.**collidesWith**(**obj**);**

**}**

**}**

**}**

**}**

**}**

Je kan zien dat deze code erg ingewikkeld is. Er wordt gebruik gemaakt van reflection en type arguments. Een ander nadeel is dat alle objecten langs wordt gegaan om te checken. Dit is overbodig.

### klasse

Als laatste is er nog een optie om een extentie te maken op MoveableGameObject genaamd CollidableGameObject. Eerst even de code:

public abstract class CollidableGameObject **extends** MoveableGameObject **{**

@Override

public void update**()** **{**

**super.**update**();**

ArrayList**<**GameObject**>** objects **=** getCollidedObjects**();**

**if(**objects **==** **null)**

**return;**

**for(**GameObject obj **:**objects**){**

**if(**obj **instanceof** CollidableGameObject**){**

CollidableGameObject enemy **=** **(**CollidableGameObject**)** obj**;**

enemy**.**collisionOccurred**(this);**

**}**

**}**

**}**

public abstract void collisionOccurred**(**CollidableGameObject collidingObj**);**

**}**

Het mooie van deze oplossing is dat je als gebruiker de klasse CollidableGameObject kan uitbreiden. Op het moment dat jou object botst met een ander CollidableGameObject dan wordt de functie collisionOccurred aangeroepen.

Ook hier zitten nadelen aan. Als een andere klasse CollidableGameObject extend, dan is het niet mogelijk om een aanpassing te maken aan de bots-zoekcode, én ook de update van MoveableGameObject te callen. Daarom is er uiteindelijk gekozen om een combinatie te maken tussen Functie en Klasse. Dan kan de update en bots-code apart overwitten worden. In de source code is te zien hoe dit precies is opgelost.

## Entity

De Entity is een extensie op CollidableGameObject. Entity heeft levelspunten (health) en kan kwaadaardig of niet kwaadaardig zijn (friendly). Je zou kunnen zeggen dat Entity iets levends voorstelt. Entity observeerd de levenspunten en roept op het goede moment ‘onDie()’ aan. Ook heeft elke Entity een HealthBar. Binnen Entity wordt deze healthbar up-to-date gehouden.

Elke Entity heeft een SugarWars object. Dit is een voorziening voor de klasse die Entity uitbreiden.

## Player

De Player stelt de speler voor. In ons geval is dat het winkelwagentje. De speler is een extensie op Entity omdat de speler leeft. Verder houd de speler zijn levens bij, en heeft hij zelf methoden die aangeven wat er moet gebeuren als de speler dood gaat.

Binnen de speler wordt ook de input geregeld. Hier wordt gekeken of de knoppen zijn ingedrukt en wordt de juiste beweging en actie uitgevoerd.

## Enemy

De enemy stelt een slecht stuk fruit voor. Het is een abstracte klasse omdat enemy niet zelf kan bestaan. Enemy wordt extend door veel verschillende enemys. Deze zijn te vinden in het package enemies. Enemy is een extensie op Entity zodat hij kan reageren op Projectiles.

Verder kan de enemy zijn eigen functionaliteit implementeren in de update() functie en oDie() functie. De speciale eigenschappen van de enemys zijn beschreven in het ontwerp rapport.

## Projectile

Players en Enemys kunnen projectielen schieten. Een projectiel wordt weergegeven in een Projectile. Een projectiel zijn een extensie van CollidableGameObject omdat ze met andere game objecten kunnen bosten.

Elk Projectile heeft een damage en een friendly waarde. De friendly geeft aan wie er geraakt kan worden. De damage geeft aan hoeveel health van de geraakte Entity af gaat. Projectiles zullen alleen botsen met Entitys die niet de zelfde friendly waarde hebben.

## Gun

Een projectiel wordt geschoten door een Gun. Een Gun is een abstracte klasse. Een gun bevat een cooldown alarm. Een uitbreiding op een gun kan door middel van de functie startCooldown() de cooldown starten, en door middel van isInCooldown() checken of de cooldown afgelopen is. Binnen Gun wordt door middel van de boolean variable ‘isInCooldown’ bijgehouden of de Gun in cooldown is.

Verder is er een abstracte void functie die moet worden geimplementeerd door de verschillende guns. Deze is ‘fire’ en bevat wat er moet gebeuren als de gun wordt afgeschoten. Hier is de unikiteit van de guns gebruikt: Elke gun zal op een unieke manier schieten.

Guns hoeven niet per-se projectielen te schieten. Ze kunnen in theorie ook health aan de player geven. Dit is helemaal aan de gun implementatie.

## PowerUp

Een powerup is een extensie op CollidableGameObject die een Gun bevat. Op het moment dat de PowerUp met een Player bots, wordt de gun binnen PowerUp aan de Player gegeven.

## Explosion

Een explosie is een GameObject met een animatie van een explosie. Deze animatie wordt afgespeeld in de tijd die wordt meegegeven aan de constructor. Een Explosie beweegt niet, en wordt verwijdert zichzelf automatisch uit de game nadat de explosie animatie over is.

## HealthBar

De levensbar is een rood vierkant. Dit kan aan de hand van de methode ‘draw’ op een Canvas getekend worden. Met deze methode ‘draw’ wordt het percentage dat de balk gevuld moet worden meegegeven.

Verder heeft de HealthBar een leftTop, width, height en color. Voor elk van deze variable is een getter en setter aanwezig.

## Spawner

Er zijn twee soorten spawners: EnemySpawner en PowerupSpawner. Deze twee spawners hebben een paar overeenkomsten. Deze overeenkomsten zijn te vinden binnen de Spawner klasse.

De eerste overeenkomst is dat beide spawners een referentie naar de game hebben (SugarWars). Verder hebbe beide een alarm met een bepaalde tijd. Als dit alarm afgelopen is, moet het alarm opnieuw gezet worden en moet iets gespawnd worden. Dit wordt allemaal geregeld binnen Spawner.

## PowerupSpawner

Deze klasse beheert het neerzetten van powerups. Deze powerups worden random neergezet met een tussepauze tussen TIME\_BETWEEN\_SPAWN\_MIN en TIME\_BETWEEN\_SPAWN\_MIN. Verder wordt de snelheid afgewisseld tussen SPEED\_X\_MIN en SPEED\_X\_MAX.

Welke gun als powerup wordt neergezet wordt gekozen uit een lijst met gun klasse. Er wordt random een gun-klasse gekozen. Hier wordt een nieuwe instantie van gemaakt. Uit een HashMap wordt de goede afbeelding gevonden. Daarna wordt de powerup neergezet.

## EnemySpawner

EnemySpawner spawnt Enemys’s! Dit doet hij aan de hand van het ‘levels’ array binnen SugarWars. Dit array wordt level voor level uitgevoerd. Elke keer dat er een nieuw level geladen wordt, wordt het goede array uit het levels array overgeladen naar het waveItems arrayList. Deze arraylist wordt van voren naar achter uitgelezen. Dit uitlzen gebeurt in de functie spawn. Stap voor Stap gebeuren de volgende acties:

* Er wordt gechecked of het level afgelopen is, zo ja, dan moet er gewacht worden totdat de laatst gespawned enemy dood is.
* De eerstvolgende actie wordt geladen.
* Deze actie wordt uitgevoerd. Dit gebeurt door te kijken wat voor een type actie dit is:
  + Wacht Actie (De timer moet naar een nieuwe tijd gezet worden)
  + Spawn Actie (Nieuwe enemy maken en neerzetten)
* Alarm Resetten