Quebble

Software Requirements Specification

v1.0
1 april 2021

ITA-OOSE-A-f 2020
Rowan Paul Flynn 637588
Alex Cheng 634967

OOSE-OOAD
Docent: Marco Engelbart

Inhoudsopgave

Inleiding

Het bedrijf Solid Games wil een quiz-applicatie Quebble ontwikkelen. Quebble bevat een grote verzameling quizzen die elk uit acht vragen bestaan. Elke quiz is een mix van meerkeuzevragen (met steeds vier alternatieven) en kort-antwoord-vragen. Als alle vragen zijn beantwoord dan worden de vragen gecontroleerd en voor elk correct antwoord wordt er een letter gegeven. Met deze letters maakt de speler een woord, waarvoor punten correct worden toegekend.

In dit document staat beschreven wat de software gaat doen en hoe verwacht wordt dat het zich gedraagt met een domeinmodel, omschrijven van use cases en overige requirements.

1) Introductie

De implementatie van het design is te vinden in bijlage A Code Quebble. In dit hoofdstuk zijn gebruikte OO-principes te vinden.

Om belangrijke klassen niet toegankelijk te maken voor de buitenwereld wordt er gebruikt gemaakt van zogenaamde Information Hiding. Dit wordt mogelijk gemaakt door variabelen private te maken en alleen verkrijgbaar zijn door zogenaamde Getters. Daarnaast zijn er ook Setters die alleen toegang hebben tot deze private variabelen om ze aan te passen.

Voorbeeld:
```
private int id;

public int getId() {
   return id;
}

public void setId(int id) {
   this.id = id;
}
```

Deze Getters en Setters zijn dus erg essentieel voor een applicatie om een scheiding te kunnen maken tussen verschillende klassen.

Naast Information Hiding maakt de Quebble applicatie ook gebruik van High Cohesion. Ook dit is een erg veel voorkomend concept tijdens het programmeren in Java. High Cohesion houdt namelijk in dat hoe taken in de klasse of module bij elkaar horen en hoe goed de code in elkaar zit. Hierbij wordt vooral opgelet hoe je die methodes die bij elkaar horen ook daadwerkelijk bij elkaar stopt in de klasse.

Voorbeeld:
```
public class Quiz {
   private int id;
   private ArrayList<Vraag> vragen;

   public Quiz(int id, ArrayList<Vraag> vragen) {
       this.id = id;
       this.vragen = vragen;
   }

   public int getId() {
       return id;
   }

   public void setId(int id) {
       this.id = id;
   }

   public ArrayList<Vraag> getVragen() {
       return vragen;
   }
```

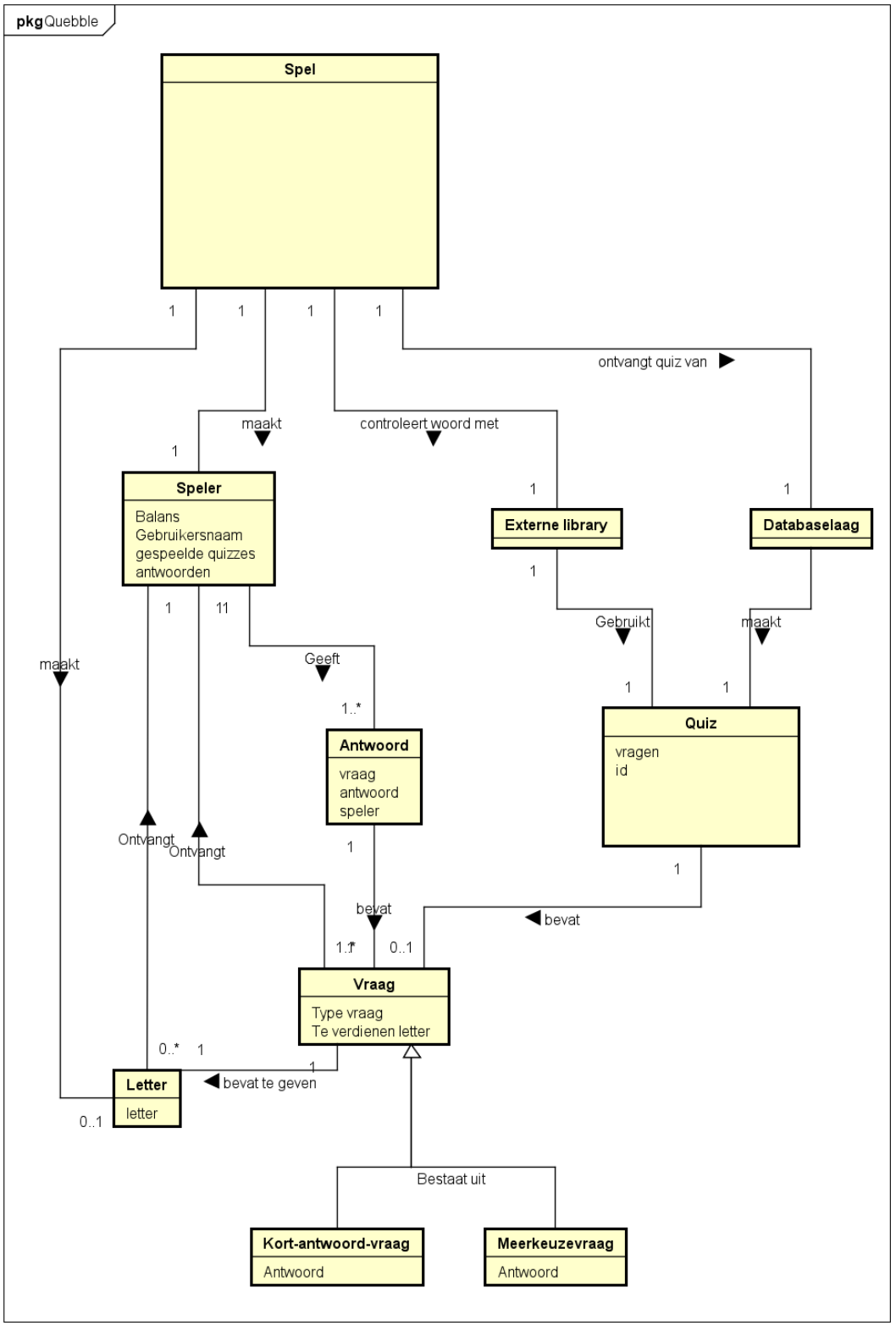
```
public void setVragen(ArrayList<Vraag> vragen) {
    this.vragen = vragen;
}
```

Dit is de Quiz klasse, deze klasse focussed zich op een specifieke taak en bevat alleen de functionaliteiten rondom de Quiz, daarom is dit een helder voorbeeld van High Cohesion.

Ten slotte wordt er in de applicatie ook gebruik gemaakt van de Single Responsibility Principle. Dit betekent dat elke klasse maar verantwoordelijk is voor een gedeelte van de applicatie. Ook hiervan is een voorbeeld hiervan de Quiz klasse, die bevat een quiz. Deze klasse is dus alleen verantwoordelijk voor de quiz.

2) Domein model

Hieronder staat het domein model die alle domeinen beschrijft voor de applicatie. De applicatie heeft een spel klasse die een speler bevat en de quiz ophaalt uit de database laag. Elke quiz bevat 8 vragen en een id. De 8 vragen kunnen zowel meerkeuze als kort antwoord vragen zijn.



3) Use Cases

De Quebble applicatie bestaat uit 4 verschillende use cases, waarvan de 3 belangrijkste verder zijn uitgewerkt::

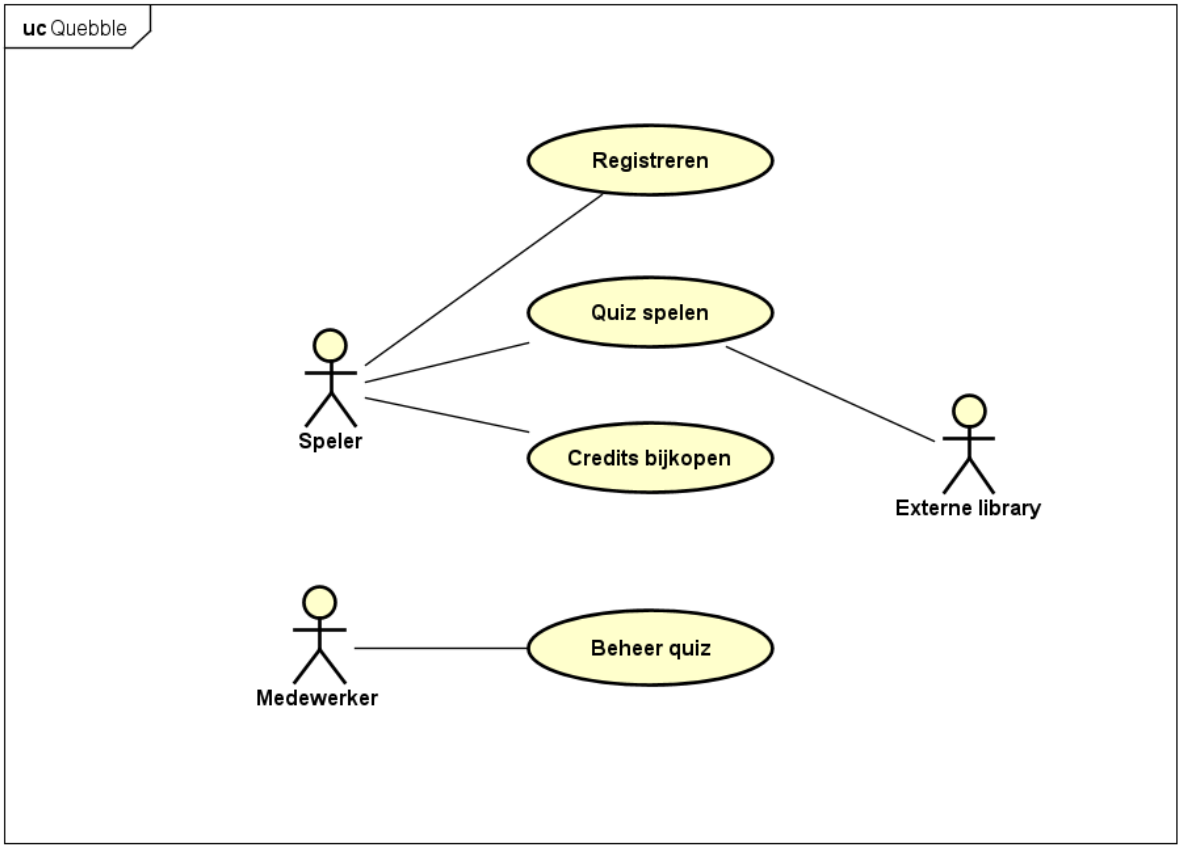
- Quiz spelen
- Registreren
- Credits bijkopen
- Beheer quiz

De belangrijkste use case is quiz spelen, want die gaat over het spelen van de quiz en bevat daardoor alle belangrijke benodigdheden. Daarna is het registreren het belangrijkst zodat een speler aangemaakt kan worden. Zonder speler zou iedereen dezelfde resultaten (en balans) delen.

Als derde is credits bijkopen, want als je je registreert dan krijg je al 1000 credits en daar kan je behoorlijk veel spellen mee spelen. Plus zonder spel en speler is er geen nut voor credits.

De minst belangrijkste use case is beheer quiz, want die gaat over het beheren van de quizzes, in het begin kunnen quizzes ook gewoon hardcoded in de applicatie komen te staan.

3.1 Use case model



Zoals hierboven te zien is, zijn er vier use cases voor de Quebble applicatie. Drie voor de speler en één voor de medewerker. Hieronder zijn de use cases in beschreven in brief description voor meer informatie.

3.2 Fully dressed use cases

Hieronder zijn de fully dressed use cases voor de drie belangrijkste use cases, zoals hierboven beschreven. Voor de laatste use case is alleen de brief use case description uitgewerkt.

3.2.1 Quiz spelen

<table>
<tr><td colspan="2">Primary actor: Speler</td></tr>
<tr><td colspan="2">Stakeholders and Interests: Solid Games</td></tr>
<tr><td colspan="2">Brief description: Wanneer een nieuwe quiz gestart wordt worden er 40 credits van de balans van de speler afgehaald en begint er een nieuw spel. De speler krijgt, voor zover mogelijk, een quiz die hij/zij nog nooit eerder heeft gehad.

Wanneer de quiz gestart is krijg de speler een vraag getoond die hij/zij moet beantwoorden. Als alle vragen zijn beantwoord dan ontvangt de speler voor elke correct beantwoorde vraag een letter. Deze letters worden gebruikt om een woord te vormen. Indien het ingevulde woord correct is wordt aan de speler, gebaseerd op een puntentelling, punten toegekend.</td></tr>
<tr><td colspan="2">Preconditions: Speler moet geregistreerd zijn.
Er moet een quiz bestaan met voldoende vragen.
Speler heeft minimaal 40 credits in zijn/haar balans.</td></tr>
<tr><td colspan="2">Postconditions (Success Guarantee): Kent punten toe.</td></tr>
<tr><td colspan="2">Main Success Scenario (Basic Flow):</td></tr>
<tr><td>Actor Action</td><td>System Responsibility</td></tr>
<tr><td>[1] Gebruiker start quiz

[7] Beantwoord vraag
[9] Herhaalt [6] & [7] totdat 8 vragen beantwoord zijn

[11] Maakt woord met gegeven letters</td><td>
[2] Checkt balans speler
[3] Haalt 40 credits van balans speler af
[4] Kijkt naar quiz die speler nog niet gedaan heeft
[5] Voegt speler toe aan nog niet gespeelde quiz
[6] Toont quizvraag

[10] Controleert ingevulde vragen
[12] Geeft letters terug op aantal correcte antwoorden

[13] Controleert of woord geldig is
[14] Kent punten toe gebaseerd op puntentelling</td></tr>
<tr><td colspan="2">Extensions (Alternative Flow):</td></tr>
</table>

[12C] Speler kan geen woord maken	**Alternative Flow A: Gebruiker heeft niet genoeg balans** [3A] Stuurt een foutmelding naar de gebruiker dat de saldo niet goed is **Alternative Flow B: Gebruiker heeft alle quizes al gespeeld** [6B] Voegt speler toe aan quiz die speler al gehad heeft **Alternative Flow C: Gebruiker beantwoord geen een vraag correct** [10C] Controleert ingevulde vragen [11C] Geeft geen enkele letter terug **Alternative Flow D: Gebruiker geeft geen geldig woord** [13D] Woord is niet geldig [14D] Worden geen punten toegekend aan speler

3.2.2 Registreren

Primary actor: Gebruiker	
Stakeholders and Interests: Solid Games	
Brief description: Als een gebruiker een quiz wil spelen dan registreert hij/zij zich met een gebruikersnaam en wachtwoord zodat zijn/haar gegevens worden opgeslagen. Nadat de gebruiker succesvol is geregistreerd krijgt hij/zij 1000 credits.	
Preconditions: Gebruiker bevat een e-mail adres	
Postconditions (Success Guarantee): Gebruiker kan nu inloggen	
Main Success Scenario (Basic Flow):	
Actor Action	**System Responsibility**
[1] Gebruiker vult e-mail en wachtwoord in	[2] Controleert of e-mail nog niet in gebruik is [3] Creëert account voor gebruiker [4] Speler ontvangt 1000 credits als startsaldo
Extensions (Alternative Flow):	
	Alternative Flow A: E-mail al in gebruik [2A] E-mail niet geldig [3A] Annuleert registreren proces

3.2.3 Credits bijkopen:

Primary actor: Speler
Stakeholders and Interests: Solid Games

Brief description: Een medewerker moet vragen kunnen beheren, bijvoorbeeld door nieuwe vragen kunnen maken en oude vragen op inactief zetten. Ook kunnen ze een nieuwe quiz samenstellen met reeds bestaande vragen of een al bestaande quiz bewerken.	
Preconditions: Speler is ingelogd	
Postconditions (Success Guarantee): Speler heeft credits gekocht	
Main Success Scenario (Basic Flow):	
Actor Action	**System Responsibility**
[1] Gebruiker geeft betaalmethode en hoeveelheid credits op die hij/zij wil kopen	
	[2] Systeem stuurt gebruiker door naar betaalsysteem
[3] Gebruiker rond aankoop af in betaalsysteem	
	[4] Systeem voegt credits toe aan account
Extensions (Alternative Flow):	
[3B] Betaling faalt	**Alternative flow A: Betaling faalt:** [4B] Geeft foutmelding aan gebruiker

3.2.4 Beheer quiz:

Een medewerker moet vragen kunnen beheren, bijvoorbeeld door nieuwe vragen kunnen maken en oude vragen op inactief zetten. Ook kunnen ze een nieuwe quiz samenstellen met reeds bestaande vragen of een al bestaande quiz bewerken.

4) Overige requirements

Hier zijn de functionele requirements besproken die niet voorkwamen in de use cases.

Legenda:
[F] - Functionality
[U] - Usability
[R] - Reliability
[P] - Performance
[S] - Supportability
[+] - Design constraints, implementation, interface & physical requirements

4.1 Functionele eisen:

[F] Bij kort-antwoord-vragen kunnen een of meer antwoorden worden goedgerekend
[F] Elke vraag behoort tot een categorie
[F] Als een vraag beantwoord hebt, kan je hem niet meer aanpassen

4. 2 Niet-functionele eisen:

[S] Er moet gemakkelijk nieuwe talen aan de applicatie kunnen worden toegevoegd
[S] De applicatie werkt op desktop en mobile
[+] De applicatie moet geschreven worden in de Java language
[P] De quiz mag geen vertragingen die langer dan 5 seconden duren in de applicatie zitten
[S] Woorden moeten kunnen worden gecontroleerd door middel van een externe library die gemakkelijk te vervangen is

Conclusie

In dit document is beschreven wat de software gaat doen en hoe verwacht wordt dat het zich gedraagt met behulp van verschillende diagrammen en requirements (al dan niet verwerkt in use cases). Het implementeren van de Quebble applicatie is een stuk makkelijker met behulp van dit document en het software design description document.