Quebble

Software Design Description

Auteurs: Sjaak Kok (620581) & Patrick Roelofs (584025)

Klas: ITA-OOSE-A-f

Course: OOAD

Docent: Marco Engelbart

Datum: 12-06-2021

Versie 2.0

1 INHOUDSOPGAVE

2 INTRODUCTION

2.1 OVERALL DESCRIPTION

Dit document is het software design description (SDD). Hierin staat de functionaliteit van de software beschreven en hoe de software zich moet gedragen.

De software die gemaakt moet worden is een quiz applicatie genaamd Quebble. Dit is een applicatie waarin spelers korte quizzen van acht vragen kunnen spelen. Het spelen van een quiz kost credits die de speler ook kan kopen.

2.2 PURPOSE OF THIS DOCUMENT

Dit document dient om de technische structuur van de applicatie in kaart te brengen. Dit gebeurt onder andere aan de hand van een design class diagram, een sequence diagram, toelichtingen op de diagrammen en eventueel worden design keuzes toegelicht die niet direct uit de diagrammen af te leiden zijn.

3 DETAILED DESIGN DESCRIPTION

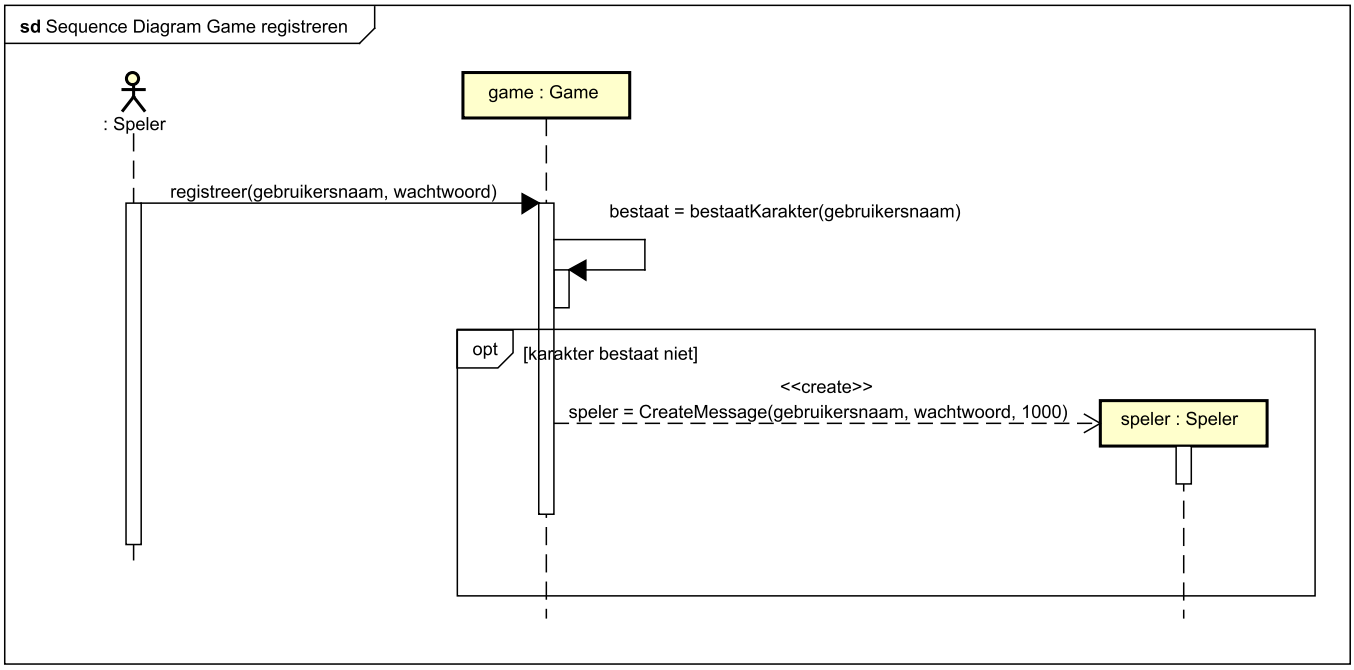
In dit hoofdstuk komen implementatie details aan bod.

3.1 SEQUENCE DIAGRAMS

De sequence diagrammen tonen acties en operaties op volgorde van tijd. In deze diagrammen komen de verantwoordelijkheden van objecten goed naar voren. Aan de hand van deze diagrammen kan er beter een design class diagram worden gemaakt.

3.1.1 Registreren

De sequence diagram hieronder gaat over de use case ‘registreren’.



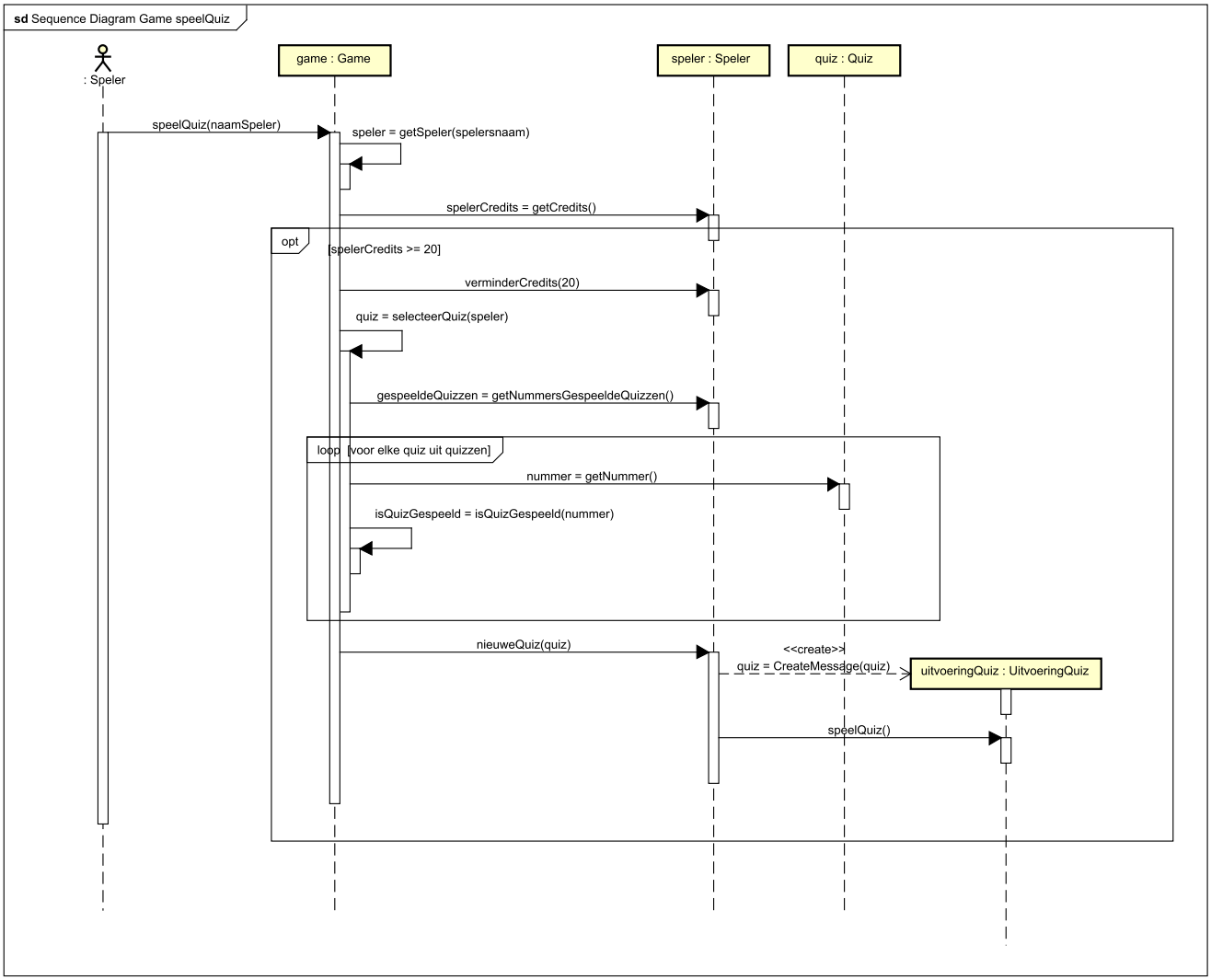
Figuur 1: Sequence Diagram 'Registreren'

Zoals hierboven is weergegeven in figuur 1 initieert de speler de operatie registreren waarin hij zijn beoogde gebruikersnaam en wachtwoord meegeeft. De game controleert vervolgens of er al een speler is met deze gebruikersnaam. Dit doet de game, omdat hij een lijst met alle spelers bijhoudt. Als er niet al een speler is met de beoogde gebruikersnaam creëert de game een nieuwe speler met de opgegeven gebruikersnaam en wachtwoord en het aantal credits dat een speler standaard krijgt bij het registreren.

3.1.2 Spelen van Quebble

De volgende sequence diagrammen gaan over de use case ‘spelen van Quebble’. Voor deze operatie zijn meerdere sequence diagrammen uitgewerkt, omdat dit een vrij grote operatie is en sommige onderdelen moeilijk leesbaar zouden zijn in één diagram.

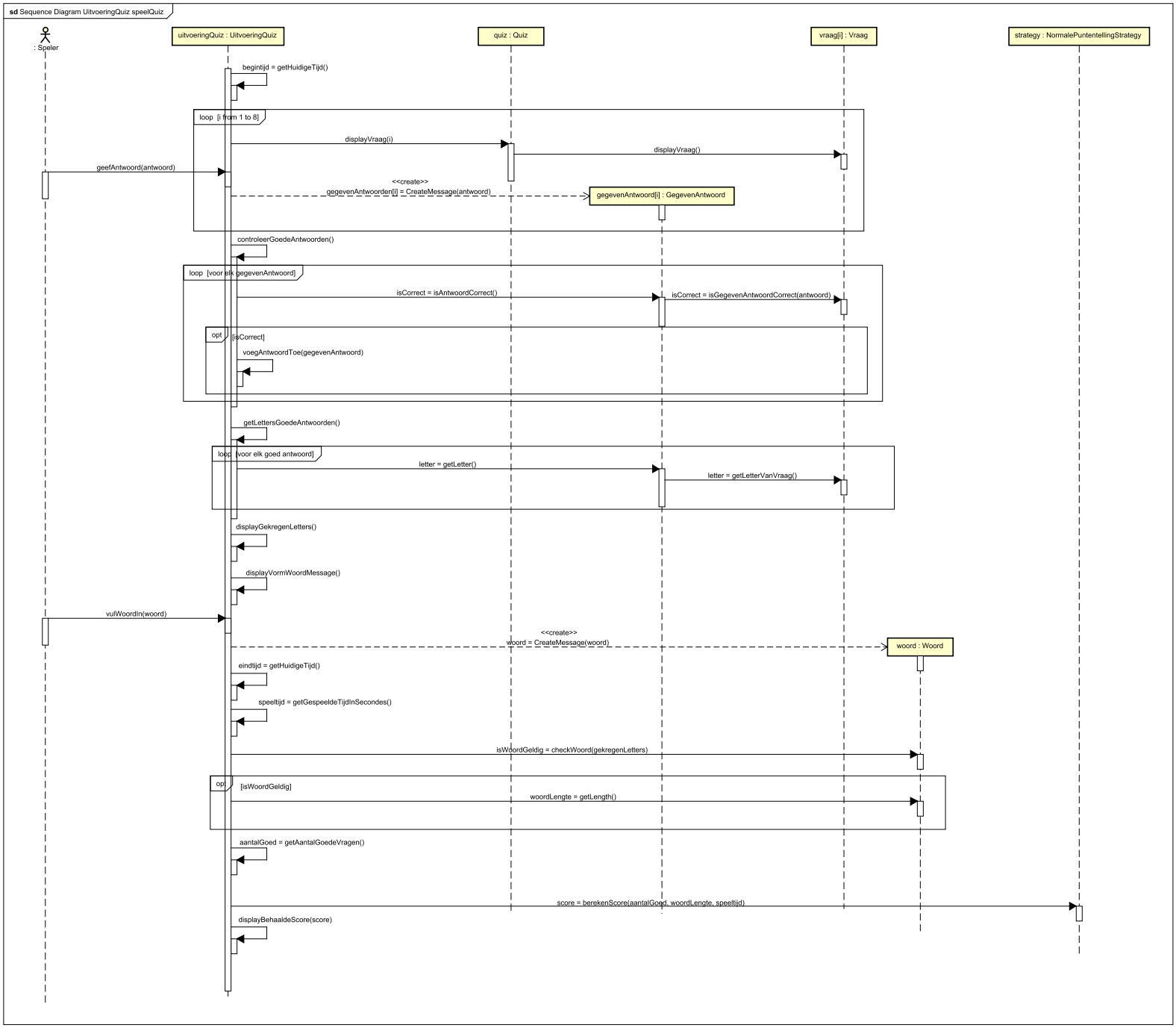
Game – speelQuiz



Figuur 2: Sequence Diagram, Game - speelQuiz

De operatie speelQuiz is de operatie waar het spelen van een quiz mee begint. Vervolgens worden de credits van een speler opgevraagd, omdat het spelen van een quiz twintig credits kost en hij die dus wel moet hebben. Daarna moet de game een quiz selecteren voor de speler. Deze verantwoordelijkheid krijgt de game, omdat hij een lijst bijhoudt van quizzen en spelers. Wanneer de game een quiz heeft geselecteerd wordt deze game meegegeven aan de speler die met deze quiz een instantie van UitvoeringQuiz creëert en daarna speelQuiz van UitvoeringQuiz aanroept. De verantwoordelijkheid om een UitvoeringQuiz te maken met de bijbehorende quiz krijgt speler, omdat Game niks hoeft te weten over de specifieke uitvoering van een quiz van een speler. De operatie speelQuiz van UitvoeringQuiz wordt verder uitgelegd in de volgende diagram.

UitvoeringQuiz – speelQuiz



Figuur 3: Sequence Diagram, UitvoeringQuiz - speelQuiz

In figuur 3 is het sequence diagram te zien voor de operatie speelQuiz van UitvoeringQuiz. Dit sequence diagram is het vervolg op het vorige sequence diagram. Op het eerste gezicht lijkt het alsof UitvoeringQuiz wel erg veel verantwoordelijkheden krijgt, maar UitvoeringQuiz heeft veel klasses aan zich gekoppeld en is daardoor de Creator. Vandaar dat deze klasse veel verantwoordelijkheden krijgt.

De speelQuiz operatie van UitvoeringQuiz begint met het vastleggen van de starttijd waarop een quiz wordt gestart. Daarna wordt er een vraag getoont. De verantwoordelijkheid om een vraag te tonen delegeert UitvoeringQuiz aan Quiz, want die bevat alle vragen. Quiz

delegeert deze verantwoordelijkheid vervolgens weer aan de Vraag zelf, omdat classes zoveel mogelijk dingen moeten doen uit hun eigen domein. Vraag kan dus het beste zichzelf tonen. Vervolgens moet de speler zijn antwoord versturen. De speler stuurt zijn antwoorden direct naar UitvoeringQuiz, omdat deze klasse de gegeven antwoorden van een speler bewaart. Dit had ook via de klasse Game gekund, maar deze klasse zou dan in principe alleen maar een doorgeefluikje zijn. Vandaar dat voor die oplossing niet gekozen is.

Zodra de speler zijn antwoord verstuurt naar UitvoeringQuiz maakt UitvoeringQuiz een nieuwe instantie van de klasse GegevenAntwoord. UitvoeringQuiz houdt alle gegeven antwoorden bij. Daarom is het logisch dat vanuit daar een GegevenAntwoord gecreëerd wordt. Ook wordt aan GegevenAntwoord de vraag waar hij bij hoort meegegeven. Dit proces van een vraag laten zien en een antwoord sturen gebeurt acht keer.

Als de speler al zijn antwoorden heeft gegeven worden zijn antwoorden gecontroleerd. UitvoeringQuiz vraagt dan aan elk GegevenAntwoord of hij correct is. Als dat het geval is wordt het antwoord toegevoegd aan een nieuwe lijst waarin alle goede antwoorden staan. GegevenAntwoord delegeert het controleren of het antwoord correct is aan Vraag, omdat Vraag dit weer kan delegeren aan het betreffende Antwoord. Het Antwoord vergelijkt zichzelf met het gegevenAntwoord, omdat dit in lijn is met het Single Responsibility principe.

Om alle gekregen letters te krijgen waarmee de speler een woord moet vormen vraagt UitvoeringQuiz van elk goede antwoord de bijbehorende letter op. Het opvragen van de letter delegeert GegevenAntwoord aan Vraag, omdat Vraag de letters bevat.

Als UitvoeringQuiz alle letters heeft worden deze getoond voor de speler. De speler moet nu een woord vormen met de gekregen letters. Als een speler een woord invoert wordt er een instantie van de klasse Woord gemaakt. Opnieuw is dit een verantwoordelijkheid voor UitvoeringQuiz, omdat het woord bij een specifieke quiz hoort.

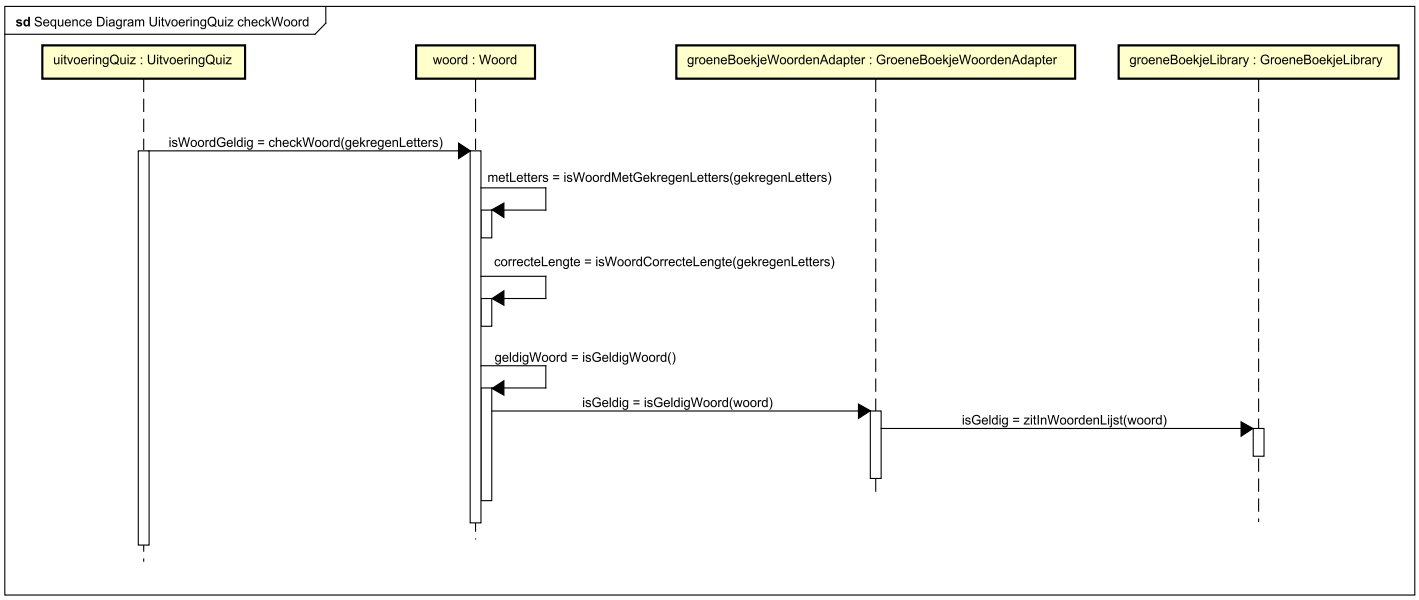
Na het invullen van het woord is de quiz klaar. Daarom wordt nu ook de eindtijd van de quiz vastgelegd. Ook wordt alvast de speeltijd berekent, zodat straks de score berekent kan worden.

Vervolgens moet gecontroleerd worden of het ingevoerde woord een geldig woord is. Het controleren van de geldigheid van een woord gebeurt door Woord, omdat deze klasse alles weet van woorden. Ook past dit bij het Single Responsibility principe, Information Hiding en Information Expert. Op deze manier hoeft UitvoeringQuiz niks te weten van hoe een Woord de geldigheid controleert. Als een woord een geldig woord is wordt de lengte van dat woord opgevraagd en anders blijft de lengte gelijk aan nul.

Omdat eerder al alle goede antwoorden zijn opgeslagen is het nu makkelijk om het aantal goede antwoorden te krijgen door het aantal goede vragen te tellen.

Vervolgens moet de score berekent worden. Dit delegeert UitvoeringQuiz aan een klasse met de gewenste strategie. Omdat er meerdere strategieën kunnen zijn moet er uiteindelijk in de Design Class Diagram een interface komen met verschillende strategieën.

UitvoeringQuiz – checkWoord



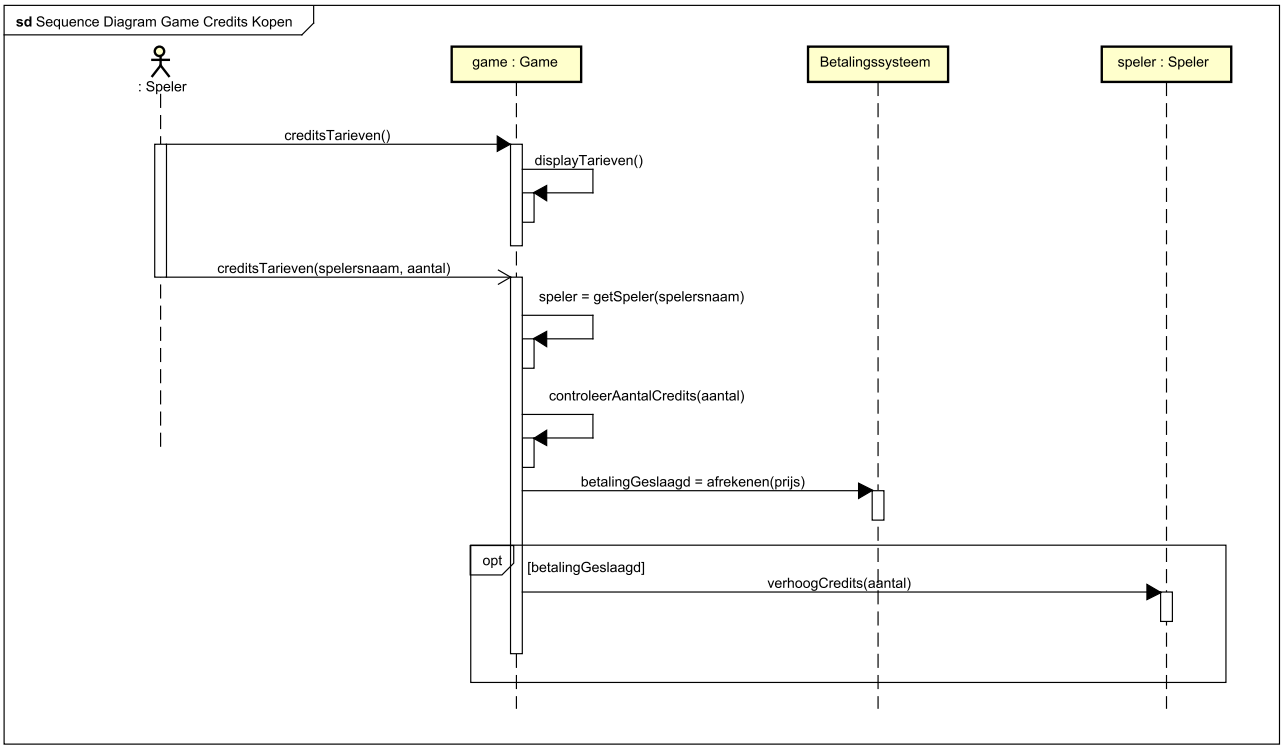
Figuur 4: Sequence Diagram, UitvoeringQuiz - checkWoord

Om de operatie om het woord te controleren te verduidelijkheden hebben we hiervoor nog een extra sequence diagram gemaakt. We hebben ervoor gekozen om hiervoor nog een apart sequence diagram te maken, omdat het sequence diagram op de vorige pagina wel erg groot werd en het anders niet meer leesbaar zou zijn.

De operatie begint met het aanroepen van checkWoord met de letters die de speler heeft gekregen met het juist beantwoorden van vragen. Vervolgens moeten er drie dingen worden gecontroleerd. Het eerste dat gecontroleerd wordt is of het ingevoerde woord ook daadwerkelijk is gemaakt met de letters die de speler heeft gekregen. Het tweede dat gecontroleerd moet worden is of het ingevoerde woord evenveel letters bevat als het aantal letters dat de speler heeft gekregen voor de juiste antwoorden. Dit moet gebeuren in verband met de mogelijkheid dat een speler een letter dubbel gebruikt en daardoor een langer woord vormt dan is toegestaan. De derde en tevens laatste controle is of het ingevoerde woord ook een daadwerkelijk woord is. Dit wordt gedaan door een externe library. Het controleren wordt gedelegeerd naar een adapter. De adapter roept vervolgens de library aan. Dit is gedaan om alleen de adapter afhankelijk te maken van de implementatie van de library.

3.1.3 Credits Kopen

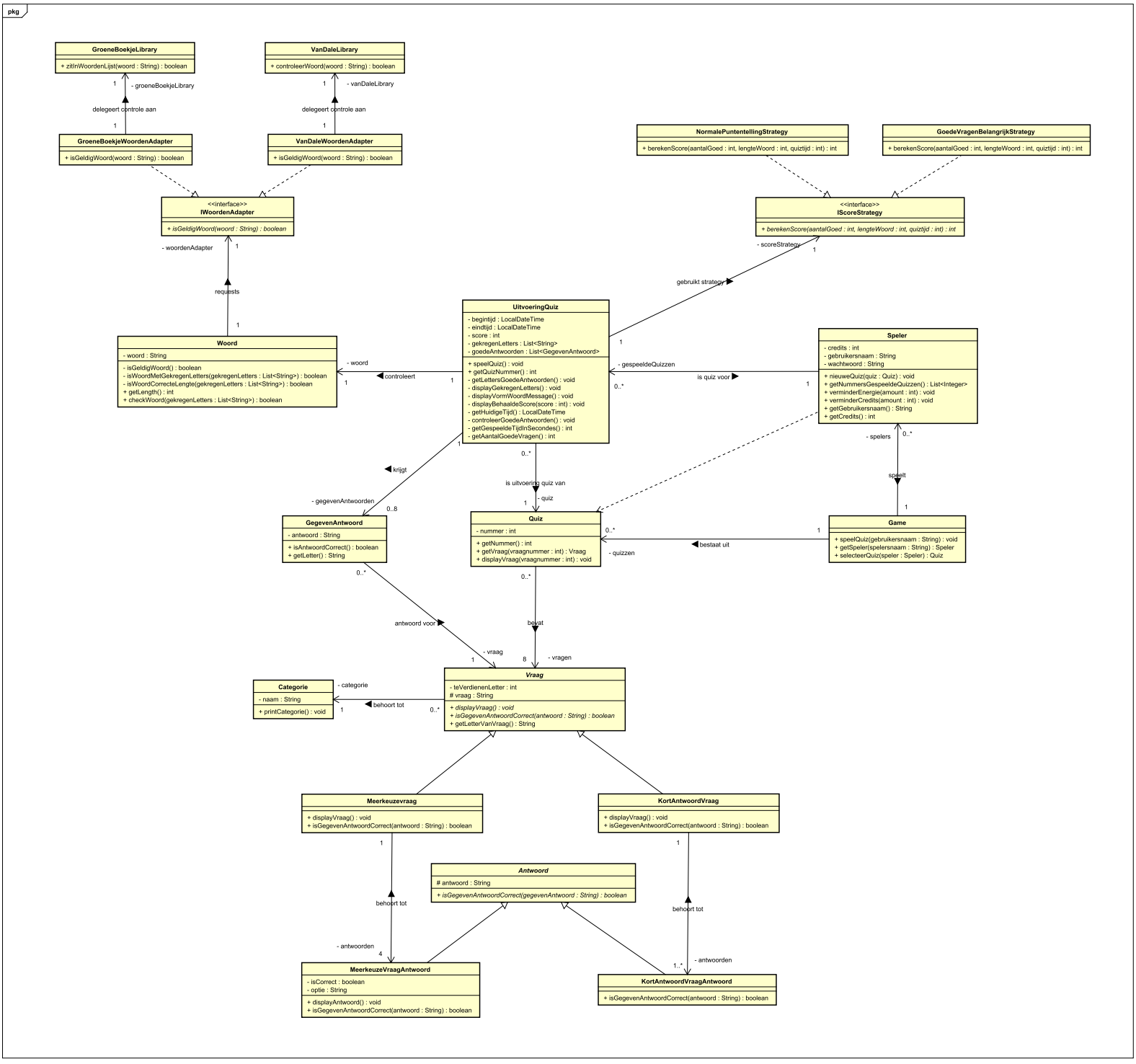
De sequence diagram hieronder in figuur 5 gaat over de use case ‘credits kopen’.



Figuur 5: Sequence Diagram 'Credits Kopen'

De speler begint met het opvragen van de tarieven. Dit hoeft de speler niet per se te doen als hij de tarieven uit zijn hoofd weet. De operatie koopCredits verzorgt het daadwerkelijke kopen van credits. De speler geeft aan deze methode zijn naam mee en het aantal credits dat hij wil kopen. Vervolgens wordt het aantal credits gecontroleerd. Dit moet gecontroleerd worden, omdat het aantal wel daadwerkelijk een aantal moet zijn dat op de tarievenlijst staat. Vervolgens wordt de speler naar het betalingssysteem gestuurd met de prijs die behoort bij het gekozen aantal credits. Als de betaling is gelukt wordt het aantal credits van de speler opgehoogt met het gekozen aantal.

3.2 DESIGN CLASS DIAGRAM



Figuur 6: Design Class Diagram

Hierboven in figuur 5 staat het volledige Design Class Diagram voor Quebble. Het diagram komt grotendeels overeen met het domeinmodel in het SRS document. De enige afwijkingen met het domeinmodel zitten hem in het toepassen van design patterns die worden toegelicht in het volgende hoofdstuk.

3.3 DESIGN DECISIONS

Om een zo goed mogelijke applicatie te ontwikkelen voor Quebble is er gebruik gemaakt van verschillende design patterns en principes. Design keuzes die eerder nog niet zijn besproken staan hier toegelicht.

Ten eerste is er gebruik gemaakt van het Strategy Pattern voor het berekenen van de score. Dit is goed te zien in figuur 5. Het Strategy is hier toegepast, omdat er voor het berekenen voor de score meerdere opties moeten zijn. Voor elke strategie is er een aparte klasse aangemaakt. Die de verantwoordelijkheid krijgt voor een berekening. Deze implementatie houdt ook nauw verband met het Open/Closed principle waarbij classes open moeten zijn voor uitbreiding maar gesloten voor verandering. Voor het berekenen van de score is er nu makkelijk een extra manier toe te voegen zonder veel code te hoeven aan te passen.

Ten tweede is er gebruik gemaakt van het Adapter Pattern. Dit is goed te zien linksboven in figuur 5. Er is voor dit pattern gekozen, omdat de klasse Woord een interface voor een bepaalde methode verwacht, maar de libraries die de het controleren van het woord kunnen doen gebruiken allebei een andere interface. Door gebruik te maken van het Adapter Pattern wordt de implementatie van de library omgezet in een implementatie die de klasse Woord verwacht. Ook wordt door gebruik te maken van dit pattern het Dependency Inversion Principle in stand gehouden waarbij high-level modules niet afhankelijk zijn van low-level modules en in plaats daarvan afhankelijk zijn van abstracties. In dit geval is het Woord niet direct verbonden met de library.

3.4 NA HET IMPLEMENTEREN

Over het algemeen is het goed gelukt om de applicatie te implementeren in Java na de applicatie te hebben uitgedacht in dit document.

Een verbeterpunt voor de applicatie zou zijn dat alle user input aan de voorkant van de applicatie gebeurt. Op dit moment zit alle user input in de klasse UitvoeringQuiz. Op deze manier is het lastiger om er bijvoorbeeld een front-end applicatie aan te koppelen.