|  |  |
| --- | --- |
| **Software Design Description: Quebble** | **Academie IT en Mediadesign**    Course: OOAD  **Jaap Maaskant, 638742**  **Boris Otte, 640749**  Versie 1.0  Docent:  Herman Telman  AIM 2e leerjaar  1 april 2021  Nijmegen |

Inhoud

[1. Inleiding 3](#_Toc68182249)

[2. Design Class Diagram 4](#_Toc68182250)

[3. Sequence Diagrammen 8](#_Toc68182266)

[3.1 Quiz spelen 9](#_Toc68182267)

[3.1.1 System sequence diagram 9](#_Toc68182268)

[3.1.2 speelQuebble 10](#_Toc68182269)

[3.1.3 getVolgendeActie 11](#_Toc68182270)

[3.1.4 getLetters 12](#_Toc68182271)

[3.1.5 verwerkReactie 13](#_Toc68182272)

[3.1.6 verwerkAntwoord 14](#_Toc68182273)

[3.1.7 verwerkGemaaktWoord 15](#_Toc68182274)

[3.1.8 geefEindscore 16](#_Toc68182275)

[3.2 Quiz beheren 17](#_Toc68182276)

[3.2.1 System sequence diagram – quiz samenstellen 17](#_Toc68182277)

[3.2.2 stelQuizSamen 18](#_Toc68182278)

[3.3 Credits bijkopen 19](#_Toc68182279)

[3.3.1 System sequence diagram 19](#_Toc68182280)

[4. Ontwerpkeuzes 20](#_Toc68182281)

[4.1 IPrintableToConsole 20](#_Toc68182282)

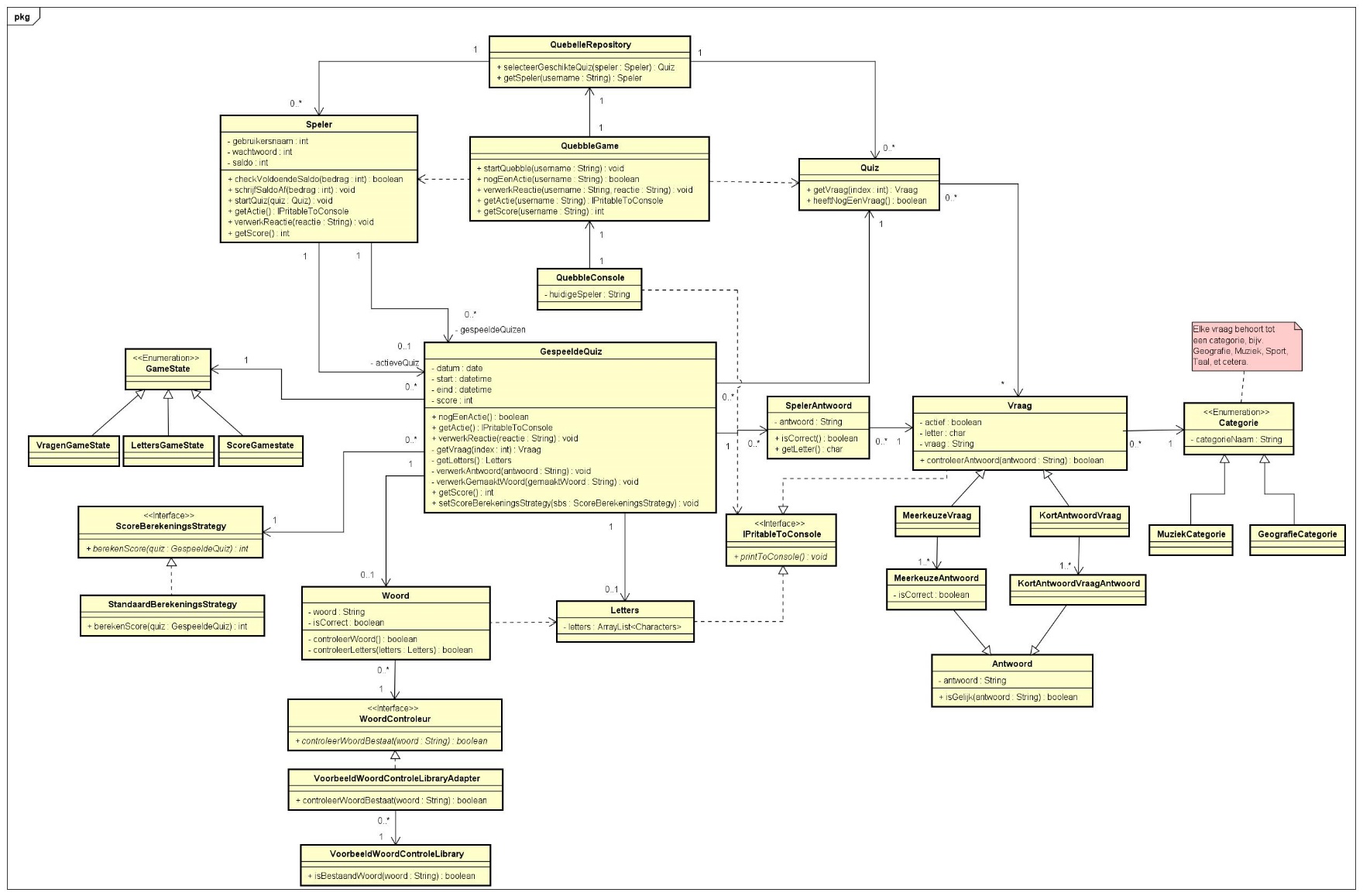
[4.1 State Pattern 20](#_Toc68182283)

# Inleiding

In dit document is het ontwerp van de quiz-applicatie Quebble te vinden. Het ontwerp wordt beschreven aan de hand van een design class diagram en meerdere sequence diagrammen. Dit document bouwt voort op de bijbehorende software requirements specification.

# Design Class Diagram

In dit hoofdstuk wordt het design class diagram behandelt. Aan de hand van dit diagram zal het gebruik van OO ontwerpprincipes en design patterns worden toegelicht.



Figuur 1 - Design Class Diagram

### QuebbleConsole

Deze class is puur verantwoordelijk voor de interactie met de gebruiker. Beslissingen over wat de volgende actie is en hoe een reactie verwerkt moet worden liggen buiten deze class. Het enige waar deze class kennis over heeft is hoe acties overgebracht kunnen worden naar de gebruiker en hoe reacties opgevangen kunnen worden. Het verwerken hiervan wordt gedelegeerd naar de class QuebbleGame. Tijdens de realisatie is ervoor gekozen QuebbleConsole de gebruikersnaam bij te laten houden zodat de gebruiker niet bij iedere systeemoperaties zijn gebruikersnaam mee hoeft te geven. Hierdoor wordt QuebbleConsole een soort client die kennis heeft van de gebruiker terwijl de rest van de applicatie stateless blijft.

Dit is een class die niet voorkomt in het domeinmodel. Tijdens het opstellen van het domeinmodel wordt niet nagedacht over zaken als interactie met de gebruiker. Tijdens de verder technische uitwerking van het model is dit uiteraard wel belangrijk, vandaar dat deze taak nu wel meegenomen is in het ontwerp.

### IPrintableToConsole

Om QuebbleConsole onafhankelijk te houden van concrete implementaties van spelfases en -objecten is QuebbleConsole gebonden aan deze interface. Zodoende hoeft QuebbleConsole geen kennis te hebben van het specifieke object dat het afbeeld voor de gebruiker, het weet alleen zeker dat het object zelf de functionaliteit bevat om dit te doen. Op het moment implementeren alleen Vraag en Letters deze interface maar door dit ontwerp kunnen dit in de toekomst eenvoudig spelfases en -objecten toegevoegd worden zonder dat QuebbleConsole aangepast hoeft te worden.

### QuebbleGame

Deze class linkt een aantal belangrijke classes maar verwerkt zelf weinig logica. Het identificeren van een speler op basis van de username wordt gedelegeerd naar QuebbleRepository . Ook wordt het selecteren van een geschikte quiz voor een speler gedelegeerd naar de QuebbleRepository. Alle systeemoperaties die via QuebbleConsole binnenkomen worden verder gedelegeerd naar de juiste objecten die zijn opgehaald uit QuebbleRepository.

QuebbleGame is de toegang tot alle functionaliteit van het domein. Deze class bestond nog niet in het domeinmodel omdat daar juist het domein in kaart werd gebracht. Nu er nagedacht wordt over de technische implementatie is het van belang dat er een class bestaat die deze functie op zich neemt. Zodoende komt deze class nu wel terug in het ontwerp.

### QuebbleRepository

Deze class is in de eerste plaats verantwoordelijk voor het opslaan en beschikbaar stellen van data. Daarnaast heeft hij ook de verantwoordelijkheid om een geschikte quiz te selecteren voor een speler. Aangezien hij al beschikt over alle informatie met betrekking tot quizzen, spelers en spelgeschiedenis is hij de aangewezen kandidaat om deze beslissing te nemen.

Tijdens het opstellen van het domeinmodel is nog niet nagedacht over het opslaan en uitlezen van data. Nu er nagedacht wordt over de technische implementatie is dit concept wel van belang en is er een class ontworpen dat deze taak op zich neemt.

### Speler

Deze class representeert de speler. Hij houdt state bij voor een specifieke speler en bevat alle methodes die een speler uit kan voeren op de quiz die hij op dat moment aan het spelen is. Het daadwerkelijk uitvoeren van deze acties wordt opnieuw gedelegeerd. Dit maal naar naar de actieve quiz, wat een instantie is van GespeeldSpel. De enige logica die wel echt verwerkt wordt binnen deze class is het controleren of er voldoende saldo is en het afschrijven ervan. De informatie die hier voor nodig is ligt ook binnen deze class dus deze is hier de aangewezen kandidaat voor.

### Quiz

Deze class is in de eerste plaats een verzameling vragen die samen een quiz vormen. Om deze informatie bruikbaar te maken voor andere classes kunnen vragen opgevraagd worden op basis van een index. Daarnaast is er een methode beschikbaar waarmee andere classes kunnen achterhalen of er nog een vraag te beantwoorden is of dat de quiz volbracht is.

### GespeeldeQuiz

Deze class linkt Speler en Quiz. Waar Quiz een algemene samenstelling van een quiz bevat, is GespeeldeQuiz een daadwerkelijke actualisatie van een gespeelde quiz. Zodoende komen er in deze class een aantal aspecten samen en wordt er daadwerkelijk logica verwerkt. Daarnaast bevat de class uiteraard basis gegevens over de gespeelde quiz zoals de speeldatum en het start- en stop moment zodat speelduur achterhaald kan worden.

Er wordt een GameState bijgehouden zodat GespeeldeQuiz kan achterhalen hoe een reactie verwerkt moet worden en welke volgende actie uitgegeven moet worden, indien beschikbaar.

Er wordt een index bijgehouden waarmee achterhaald kan worden op welke vraag een antwoord binnenkomt en welke vraag als volgende uitgegeven moet worden, indien er nog een beschikbaar is. De daadwerkelijke vragen worden opgehaald uit Quiz op basis van deze index.

Wanneer een quiz uitgespeeld is kan de score opgevraagd worden. Het berekenen hiervan wordt gedelegeerd naar een ScoreBerekeningsStrategy, zodat hier in de toekomst gemakkelijk andere algoritmes voor ingezet kunnen worden.

### Gamestate

Gamestate is een enum die gebruikt wordt om bij te houden in welke spelfase een GespeeldeQuiz zich bevindt. Op het moment zijn dit enkele Vragen, Letters en Score.

### Letters

Deze class bestaat voornamelijk om de verdiende letters naar de client te verzenden en te kunnen weergeven. Vervolgens zijn ze nodig om een woord mee te kunnen controleren. Deze class heeft geen verdere verantwoordelijkheden.

Het concept letter kwam al wel terug tijdens het opstellen van het domeinmodel maar wordt niet alleen als collectie opgenomen in het classdiagram. Dit is voornamelijk gedaan omdat er alleen met letters als collectie acties worden ondernomen en de opslag van letters die verdient kunnen worden in vragen prima in de vorm van een Char gedaan kan worden.

### Woord

Deze class bevat een door een speler ingevoerd woord. Het is aan deze class om dit woord te controleren op overeenkomst met de meegegeven letters. Daarnaast dient het te controleren of het woord een bestaand Nederlands woord is. Daarvoor gebruikt het een externe library. Om deze library uitwisselbaar te maken, is hier het adapter pattern toegepast in de vorm van de interface WoordControleur en bijbehorende adapter.

Woord kwam al wel terug in het domeinmodel en er was ook al wel enige kennis dat deze gecontroleerd moesten worden om mee te nemen in de score, maar de technische implementatie was nog niet van belang. Nu deze wel wordt uitgewerkt en de requirement van het gebruik van uitwisselbare externe libraries wordt meegenomen

### ScoreBerekeningsStrategy

Het berekenen van de score wordt door GespeeldeQuiz naar deze class gedelegeerd. De voornaamste reden om dit zo te modeleren is omdat er is aangegeven dat de klant het berekenen van scores flexibel wil houden om hier in de toekomst eventueel andere algoritmes voor te gebruiken. Op het moment is er maar één strategy maar dit ontwerp biedt wel de mogelijkheid dit eenvoudig uit te breiden.

Tijdens het opstellen van het domeinmodel is alleen vastgesteld dat een GespeeldeQuiz een behaalde score moet hebben maar niet hoe deze berekend dient te worden. Nu er nagedacht wordt over de technische implementatie is dit echter wel van belang en wordt ook belangrijk dat dit op een flexibele manier moet gebeuren. Zodoende komt deze nieuwe class hier nu aanbod.

### Vraag

Deze class bevat een vraag, één of meerdere Antwoorden voor die vraag, een Letter en een Categorie. De Vraag en de Letter zijn door andere classes op te halen. Vraag is verantwoordelijk voor het controleren van SpelerAntwoorden. Vraag implementeert IPrintableToConsole, en print in het geval van een Kort-Antwoord-Vraag alleen zijn vraag, en print in het geval van een MeerkeuzeVraag ook de MeerkeuzeAntwoorden.

### Categorie

Categorie is een enum die aangeeft wat de categorie van een Vraag is. In de huidige implementatie heeft deze class verder geen functie.

### Antwoord

Deze class bevat een antwoord op een vraag. Deze class kan zichzelf vergelijken met een inkomend antwoord om aan te geven of ze gelijk zijn of niet. In het geval van een MeerkeuzeVraagAntwoord houdt het antwoord ook bij of het correct is of niet.

### SpelerAntwoord

Deze class bevat een door de speler gegeven antwoord op een Vraag en wordt aangemaakt door GespeeldeQuiz. Het SpelerAntwoord kan zichzelf doorspelen naar de Vraag om zich te laten controleren, en kan van die Vraag de bijbehorende Letter opvragen.

# Sequence Diagrammen

Om de functionaliteit van het systeem verder uit te werken zijn er, voor de belangrijkste systeemoperaties sequence diagrammen opgesteld. In dit hoofdstuk worden deze diagrammen behandelt en waar van toepassing worden ontwerpkeuzes toegelicht. In het vorige hoofdstuk zijn al een groot aantal keuzes behandelt, om niet in herhaling te vervallen zal er in dit hoofdstuk alleen maar genoemd worden hoe de eerder toegelichte keuzes ook hier te herkennen zijn. Sommige ontwerpkeuzes komen in meerdere of zelfs alle diagrammen terug. Deze zullen alleen de eerste keer benoemd worden en daarna als toegelicht geschouwd worden. Ook de verantwoording voor de toegevoegde classes ten opzichte van het domeinmodel zijn al in het voorgaande hoofdstuk toegelicht en zullen hier niet herhaalt worden.

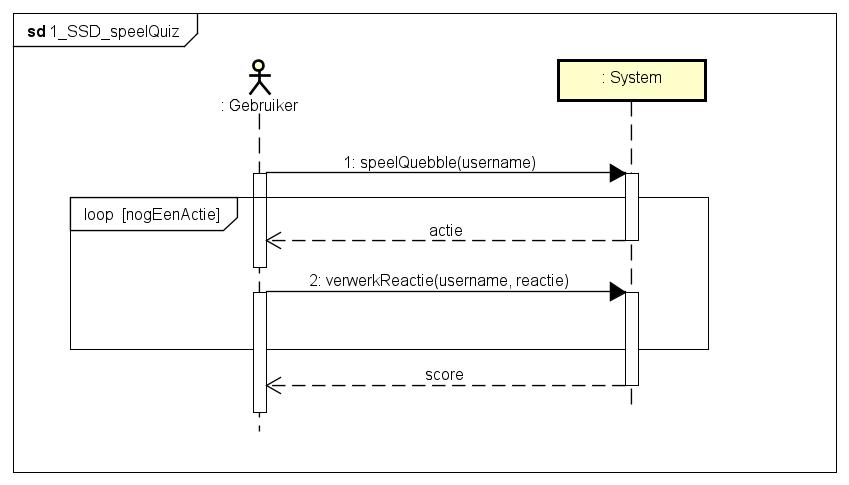
Bij erg grote sequence diagrammen is ervoor gekozen deze op te splitsen in meerdere diagrammen. Dit om het overzichtelijk te houden en in één diagram te focussen op één stap binnen de systeemoperatie. Dit heeft helaas wel tot gevolg dat niet iedere sequence diagram met een reply message naar de gebruiker kan eindigen wanneer dat wel het geval moet zijn. De gebruikte tool laat het niet toe om een reply message te sturen naar een actor die niet de eerste methode-aanroep deed, in ons geval valt deze buiten de diagram en is dus onbereikbaar.

Tot slot moet gezegd worden dat, zoals toegelicht in het voorgaande hoofdstuk, er in de implementatie voor gekozen is een specifieke class de verantwoordelijkheid te geven de username van de huidige speler te laten opslaan. Zodoende is het niet nodig bij iedere systeemoperatie de username mee te geven. Er is voor gekozen dit in System Sequence Diagrammen en Sequence Diagrammen wel zo te laten om het stateless principe van de applicatie te benadrukken. Dit is het enige deel waar de diagrammen niet overeenkomen met de code.

## 3.1 Quiz spelen

### 3.1.1 System sequence diagram

De speler geeft aan een quiz te willen starten, vervolgens toont het systeem telkens acties die de speler moet ondernemen. De speler reageert gedurende het spel via de console. Wanneer er geen spelacties meer zijn toont het systeem de score die de speler behaald heeft voor de quiz.

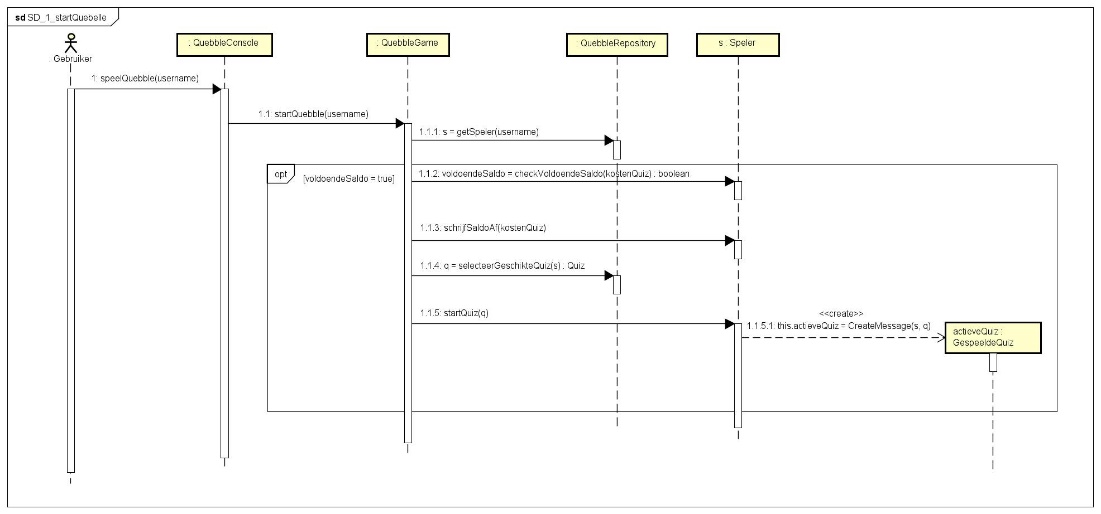


Figuur 3 - Sequence Diagram speelQuebble

### 3.1.2 speelQuebble

De speler geeft aan een quiz te willen starten, de kosten van de quiz worden afgeschreven van het saldo, een geschikte quiz wordt gezocht en gestart.

In dit diagram komen een aantal ontwerpkeuzes terug. Zo doet QuebbleConsole niks anders dan de input van de gebruiker vertalen naar een actie die op QuebbleGame aangeroepen wordt. QuebbleGame doet op zijn beurt niks anders dan QuebbleRepository vragen de juiste Speler bij de username te vinden en vervolgens methodes op deze Speler aan te roepen. Het daadwerkelijk afhandelen van de logica wordt allemaal binnen de Speler gedaan.

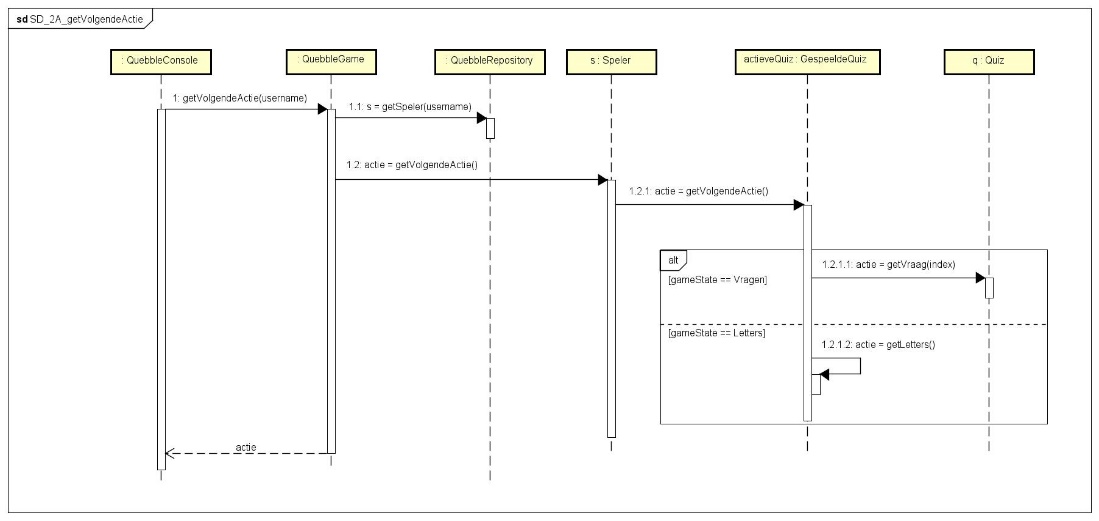


Figuur 4- Sequence Diagram startQuebble

### 3.1.3 getVolgendeActie

Een volgende actie wordt opgehaald om aan de speler te tonen.

In dit diagram komt vooral erg duidelijk terug hoe vrijwel alles classes geen kennis hebben van wat de actie precies inhoudt, i.e. of er om een vraag of om letters wordt gevraagd. Alleen het GespeeldeQuiz object heeft kennis van de state van de quiz die op dat moment gespeeld wordt en kiest op basis daarvan welke actie ondernemen moet worden.

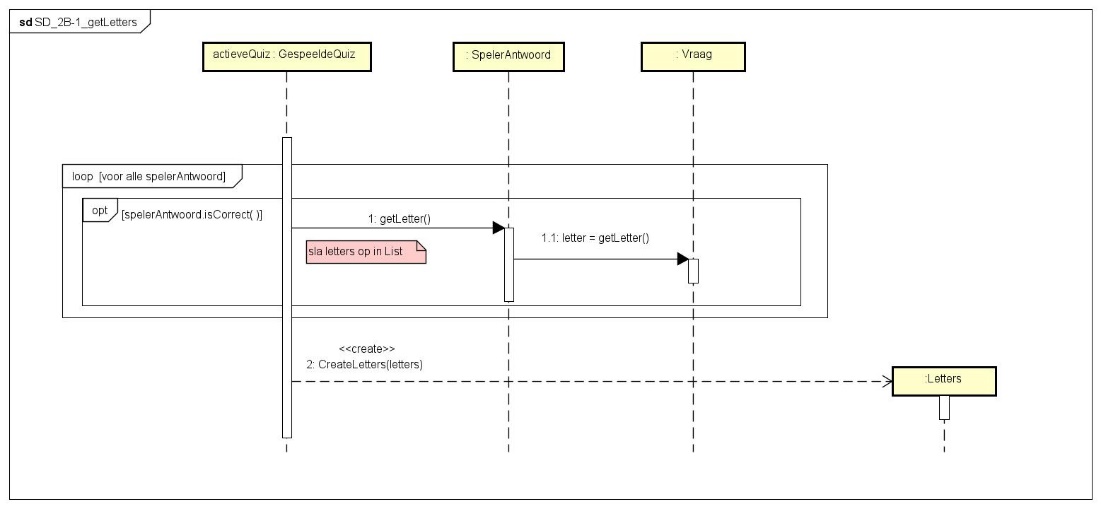


Figuur 6 - Sequence Diagram getVolgendeActie

### 3.1.4 getLetters

Na afloop van de quiz worden de letters behorende bij correcte SpelerAntwoorden opgehaald en verwerkt tot Letters.

In dit diagram zijn geen significante ontwerpkeuzes te vinden. Het is één van de mogelijke implementaties van deze functionaliteit, anderen zouden ook prima geweest zijn. Hooguit kan genoemd worden dat de verantwoordelijkheden netjes gedelegeerd worden waar van toepassing.

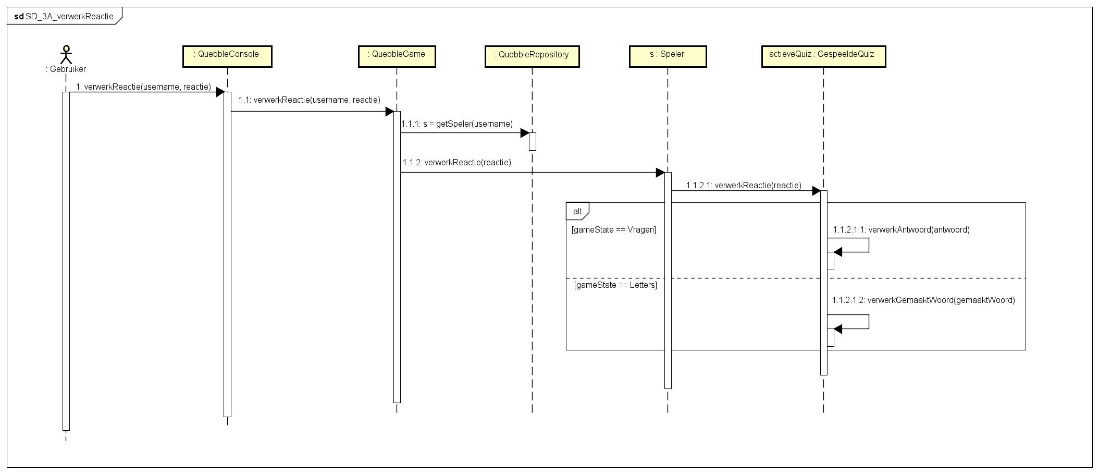


Figuur 7 - Sequence Diagram getLetters

### 3.1.5 verwerkReactie

Speler input wordt verwerkt door het systeem.

In dit diagram is net als bij de operatie getActie te zien hoe alleen GespeeldeQuiz kennis heeft van de daadwerkelijke implementatie van de acties en reacties. Dit punt wordt op meerdere plaatsen in het verslag toegelicht en zal hier niet verder behandelt worden.

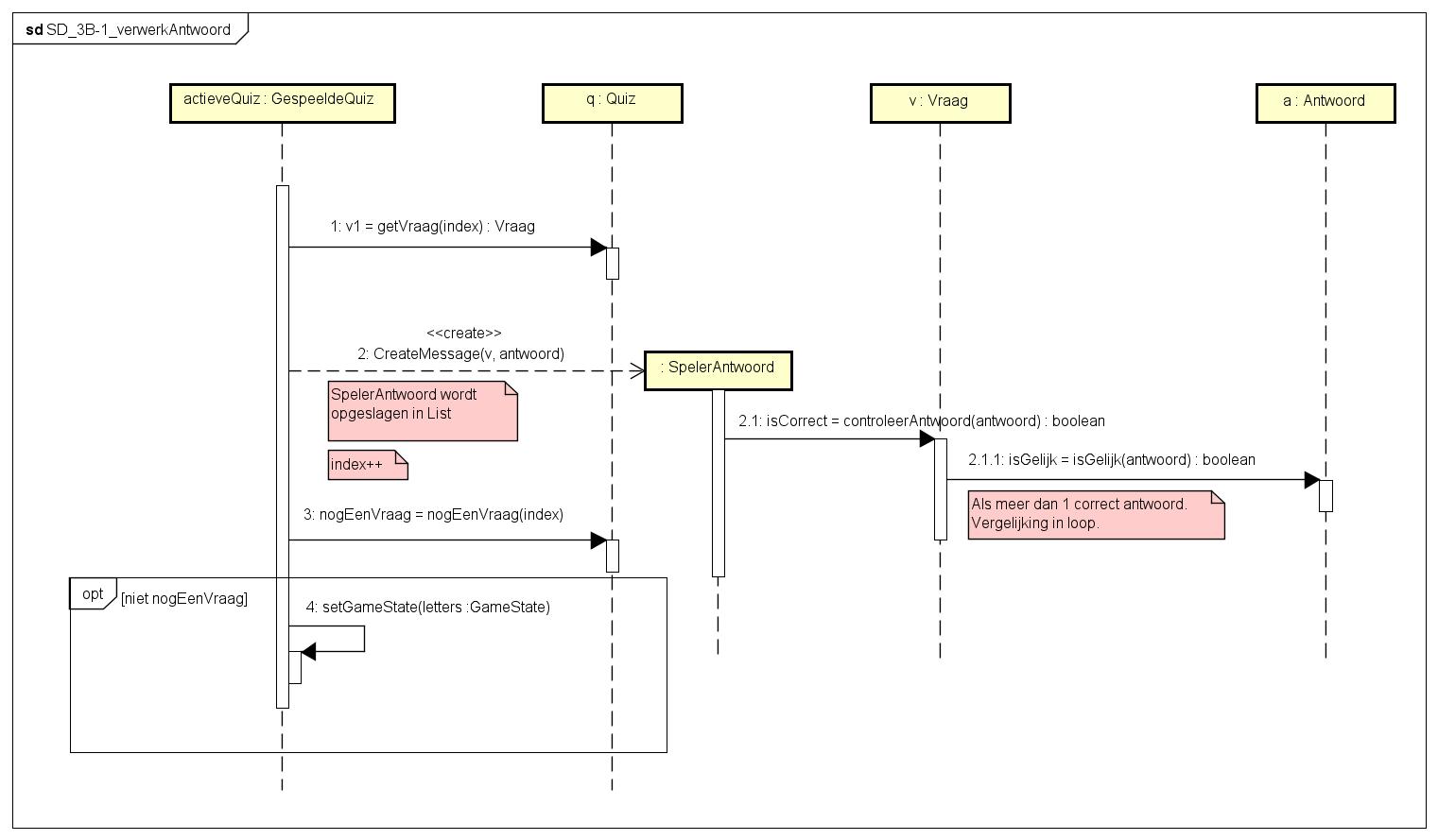


Figuur 8 - Sequence Diagram verwerkReactie

### 3.1.6 verwerkAntwoord

De speler input wordt verwerkt tot een SpelerAntwoord. Dit SpelerAntwoord controleert direct of het correct is. Wanneer er geen vragen meer zijn in de quiz wordt de gameState op letters gezet.

In dit diagram zijn geen significante ontwerpkeuzes te vinden. Het is één van de mogelijke implementaties van deze functionaliteit, anderen zouden ook prima geweest zijn. Hooguit kan genoemd worden dat de verantwoordelijkheden netjes gedelegeerd worden waar van toepassing.

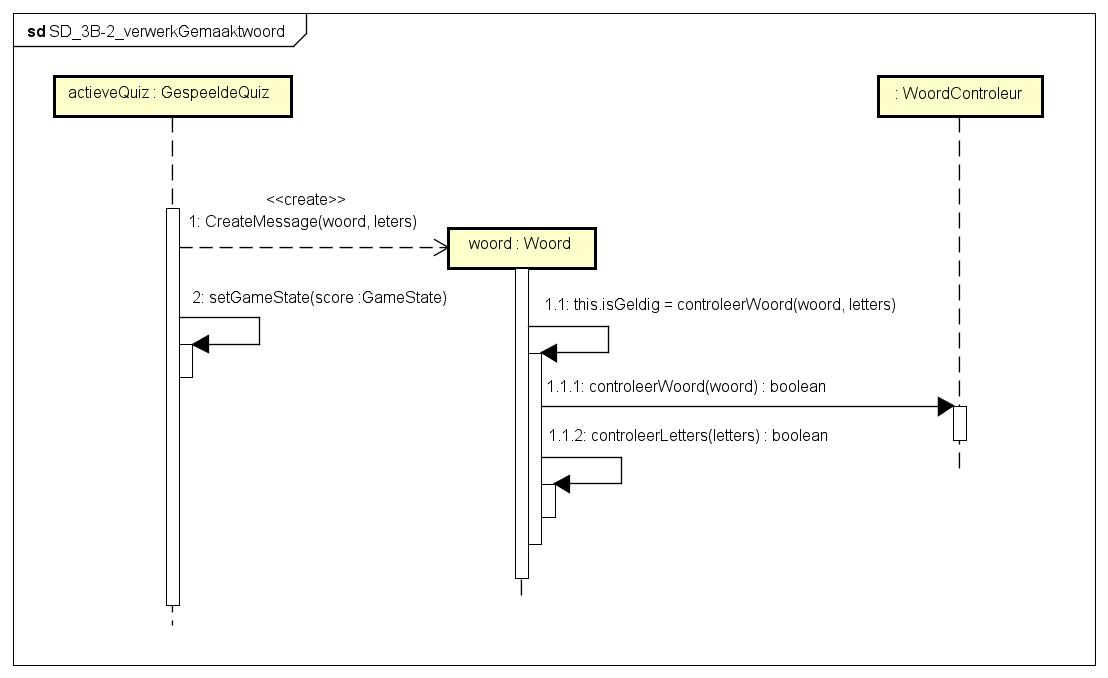


Figuur 9 - Sequence Diagram verwerkAntwoord

### 3.1.7 verwerkGemaaktWoord

De speler input wordt verwerkt tot een Woord. Dit woord controleert zichzelf op overeenkomst met de gegeven letters en laat zich door de WoordControleur controleren om te checken of het woord een bestaand Nederlands woord is. De gameState wordt op score gezet.

In dit diagram is een deel van het geïmplementeerde Adapter Pattern te zien. In het vorige hoofdstuk is het gebruik van dit pattern al toegelicht en hier zal er dus alleen naar worden verwezen. Wel is goed om op te merken dat vanuit het perspectief van Woord deze alleen weet dat het controleren van het woord wordt gedelegeerd naar een WoordControleur. Of deze WoordControleur eigenlijk een Adapter is, is niet van belang en wordt dus niet afgebeeld.

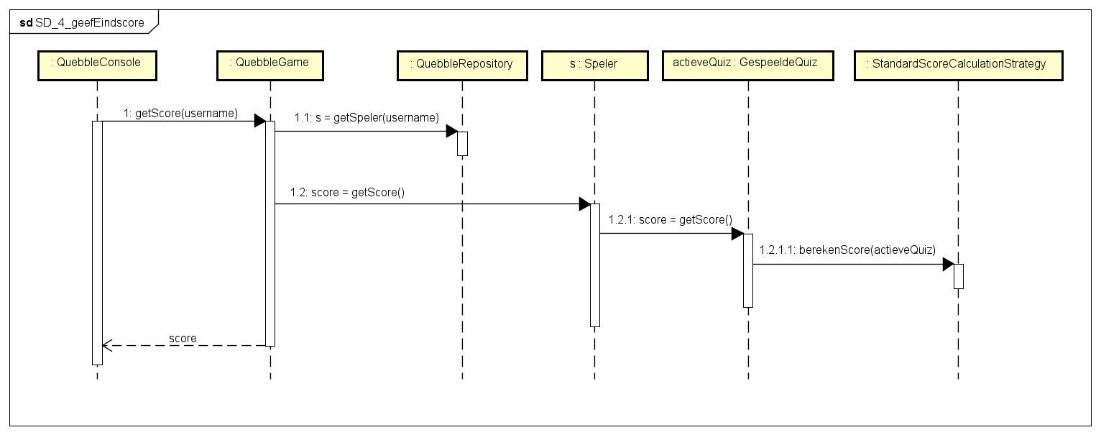


Figuur 10 - Sequence Diagram verwerkGemaaktWoord

### 3.1.8 geefEindscore

De behaalde score wordt berekend en getoond aan de speler.

In dit diagram is duidelijk het toegepaste Strategy Pattern te zien. Uiteraard is ook te zien dat zoals verwacht mag worden de vraag om de score gedelegeerd wordt naar de class die de informatie heeft om deze te berekenen. In plaats van dit zelf te doen wordt er echter gedelegeerd naar een Strategy-object wat het uitwisselen van het algoritme hiervoor in de toekomst eenvoudig geïmplementeerd kan worden. Ook deze keuze is in het vorige hoofdstuk al toegelicht.

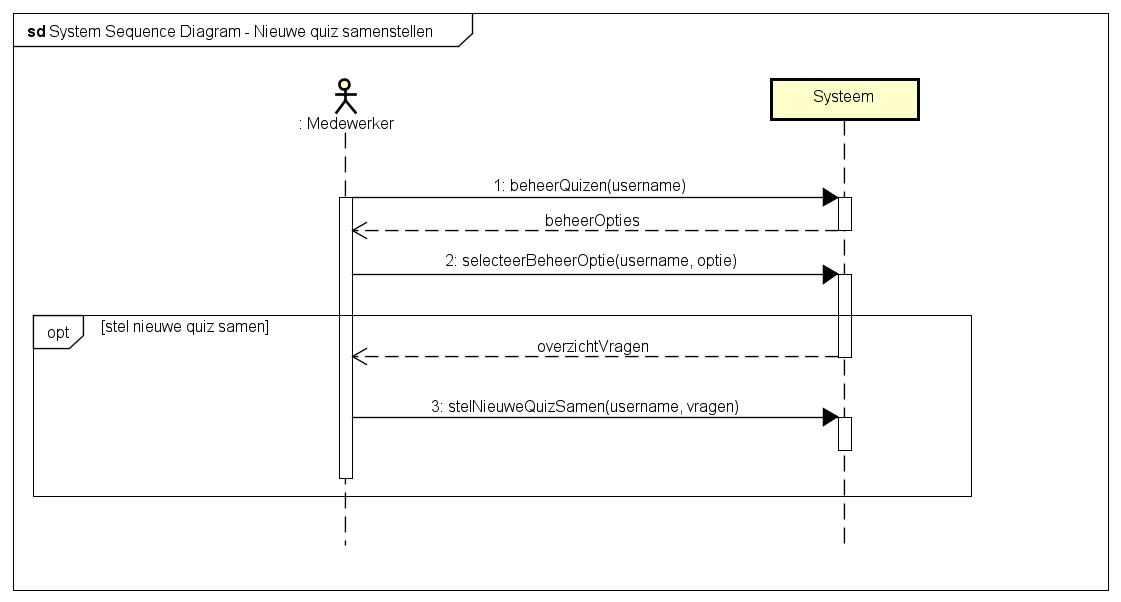


Figuur 11 - Sequence Diagram geefEindscore

## 3.2 Quiz beheren

### 3.2.1 System sequence diagram – quiz samenstellen

Een medewerker geeft aan de quizzen te willen beheren. Het systeem geeft een overzicht van de reeds bestaande quizzen en vragen alsmede de mogelijkheid een nieuwe vraag of quiz toe te voegen. De medewerker geeft aan een nieuwe quiz samen te willen stellen. Het systeem toont een overzicht van vragen waarna de medewerker met deze vragen een nieuwe quiz samenstelt.

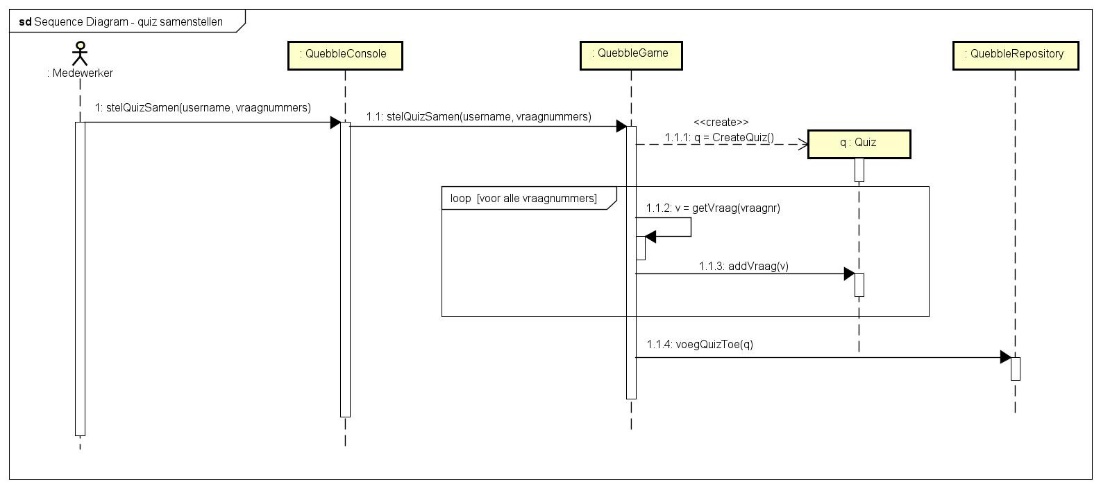


Figuur 12 – System Sequence Diagram Quiz Samenstellen

### 3.2.2 stelQuizSamen

Een Quiz wordt samengesteld uit meegeven vragen en wordt vervolgens toegevoegd aan QuebbleRepository.

In dit diagram zijn geen significante ontwerpkeuzes te vinden. Het is één van de mogelijke implementaties van deze functionaliteit, anderen zouden ook prima geweest zijn. Hooguit kan genoemd worden dat de verantwoordelijkheden netjes gedelegeerd worden waar van toepassing.

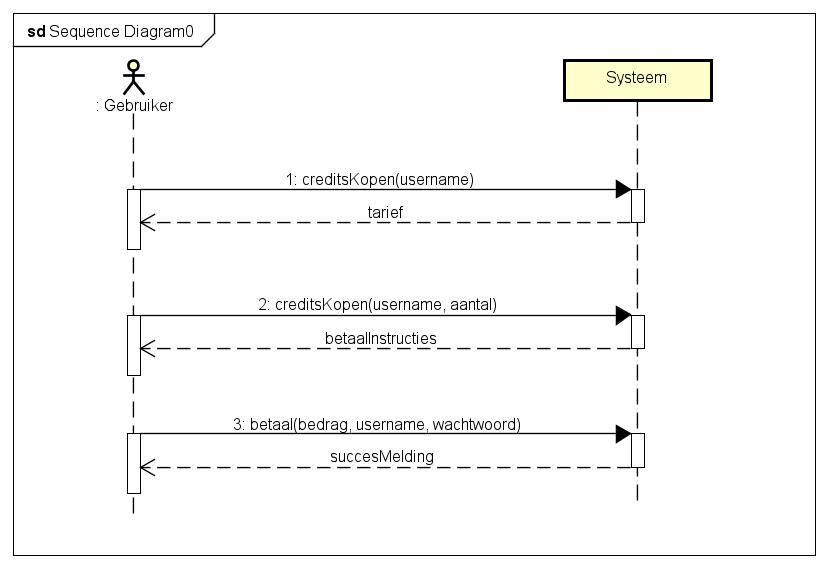


Figuur 13 - Sequence Diagram Quiz Samenstellen

## 3.3 Credits bijkopen

### 3.3.1 System sequence diagram

Een speler geeft aan credits te willen bijkopen. Het systeem toont een keuze menu voor het aantal credits en de daar bijhorende kosten. De speler maakt een keuze en wordt doorgestuurd naar de afhandeling van de betaling. Na het succesvol afhandelen van de betaling krijgt de klant een succesmelding en zijn de credits aan zijn account toegevoegd.



Figuur 14 - Sequence Diagram Credits Kopen

# Ontwerpkeuzes

In dit hoofdstuk worden niet voor de hand liggende ontwerpkeuzes nader toegelicht en afgezet tegen de alternatieven. Concreet gaat dit om de keuzes geen State Pattern toe te passen terwijl hier een duidelijke use case voor te herkennen is. Ten tweede zal de keuze voor het gebruik van de IPrintableToConsole nader toegelicht worden. Andere ontwerpkeuzes liggen meer voor de hand, zijn elders in het verslag al benoemd en zullen hier niet meer behandelt worden.

## 4.1 IPrintableToConsole

Zoals in hoofdstuk 2 besproken is er voor gekozen om QuebbleConsole onafhankelijk te houden van concrete implementaties van spelfases en -objecten door deze class te binden in deze interface. Door deze keuze kent alleen GespeeldeQuiz de daadwerkelijke implementaties. Deze keuze geeft het duidelijke voordeel dat het uitbreiden van spelfases en -objecten kan gebeuren zonder andere classes aan te passen dan GespeeldeQuiz. Het nadeel is dat spelobjecten hun eigen presentatielogica moeten bevatten in plaats van deze te encapsuleren binnen QuebbleConsole. In dit geval is de flexibiliteit met betrekking tot uitbreiden als belangrijker gezien dan het encapsuleren van presentatielogica.

## 4.1 State Pattern

Door de in de vorige paragraaf toegelichte keuze is een perfecte use case ontstaan voor het State Pattern. Met behulp van dit pattern zou het verwerken van reacties en het ophalen van acties door GespeeldeQuiz nog verder gedelegeerd kunnen worden naar de implementaties van GameState. Zodoende zou zelfs GespeeldeQuiz onafhankelijk blijven van concrete implementaties en zou het verder uitbreiden van quizfunctionaliteit nog flexibeler worden. In de huidige situatie is dit echter als niet zinvol geacht. Reden hiervoor is dat voor het verwerken van reacties kennis nodig is van de state van GespeeldeQuiz en deze state ook aangepast moet kunnen worden. Hier zijn meerdere oplossingen voor maar in alle gevallen leidt dit ofwel tot zeer sterke koppeling tussen GameState en GespeeldeQuiz ofwel veel van de logica wordt alsnog verwerkt binnen GespeeldeQuiz en het deel dat gedelegeerd wordt is minimaal. Beide scenario’s worden op het moment als niet winstgevend geacht. Wel kan het zijn dat dit in de toekomst verandert wanneer er meer spelfases en -objecten worden toegevoegd. De applicatie is zo ontworpen dat het op dat moment eenvoudig is te refactoren en alsnog een volledig State Pattern te implementeren.