Adviesrapport

Casus Zuyd L U I S T E R

Casusgroep 4 – Finn Alberts, Maud Derhaag, Peter Derks en Brent Vliex

# Introductie

Binnen Zuyd Hogeschool zijn er verschillende dienstverleners voor als studenten even niet lekker in hun vel zitten. Zo zijn er SLB’ers, decanen, vertrouwenspersonen, studentpsychologen, maar ook luisterstudenten. Zuyd L U I S T E R is er voor studenten die willen praten met medestudenten over hun problemen of dingen waar ze mee zitten. De gesprekken zijn laagdrempelig en blijven privé.

In het schooljaar 2019-2020 hebben 1514 studenten een afspraak gemaakt bij een studentpsycholoog. In datzelfde jaar meldden zich slechts 20 studenten bij een luisterstudent. Studenten weten de luisterstudenten dus niet goed te vinden (Starmans & Persoon, 2020).

Zuyd L U I S T E R wil graag haar doelgroep beter bereiken en is op zoek naar een effectieve manier om dit te doen. Er is daarom besloten een spel te maken voor de mobiele telefoon. Dit spel zal een “kies je eigen verhaal”-spel zijn, waarbij studenten scenario’s voorgelegd krijgen en gevraagd wordt wat zij zouden doen. Ze krijgen hierbij de keuze uit één of meerdere opties, waarbij iedere optie invloed heeft op welke verhaallijn de student zal gaan volgen.

Op basis van de keuzes die de student maakt kan worden gekeken hoe het met de student gaat. Dit wordt gedaan door meerdere scores bij te houden, zoals bijvoorbeeld motivatie, sociale contacten, energie, enzovoorts. Heeft een student slechte scores, dan kan het misschien helpen om met een luisterstudent te gaan praten. Deze scores hebben ook als doel confronterend te zijn. Misschien weet de student zelf namelijk nog niet dat het niet zo goed met hem/haar gaat en zorgt de score ervoor dat ze bij zichzelf hierover gaan nadenken.

Studenten kunnen meerdere ‘savegames’ hebben, wat inhoudt dat ze het spel meerdere keren tegelijk kunnen spelen. Alle scenario’s, antwoorden en scorecategorieën zijn door Zuyd L U I S T E R vrij aan te passen, via een beheerscherm binnen de applicatie.

# Probleemanalyse en probleemstelling

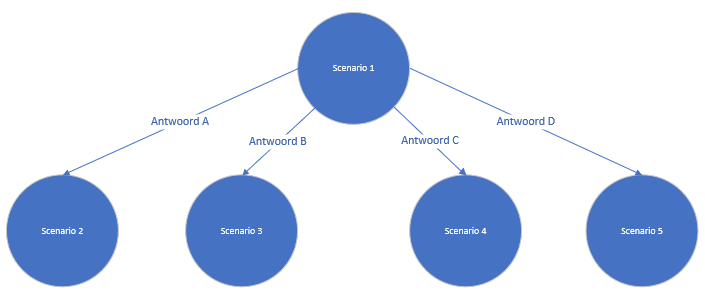
Om het bovenstaande idee te realiseren is er een manier nodig om de verschillende scenario’s met bijbehorende antwoorden op te slaan, inclusief de punten die de verschillende antwoorden opleveren. Daarbij moeten deze scenario’s ook aan elkaar gelinkt kunnen worden.

Verder is er een manier nodig om tijdens het spelen het behaalde aantal punten per categorie bij te houden. Daarnaast moet het mogelijk zijn dat aan het eind van het spel de hoogst haalbare scores te zien zijn. Doordat niet altijd dezelfde scenario’s worden doorlopen zijn deze maximale scores variabel.

Omdat de scorecategorieën ook aanpasbaar moeten zijn, is ook hier een manier nodig om dit bij te houden en hiermee te werken.

# Aanpak van het probleem

De applicatie zal worden ontwikkeld in Xamarin Forms. Xamarin Forms werkt met C#, waarin object-georiënteerd programmeren goed wordt ondersteund. Het is dus een logische keuze om te werk te gaan met behulp van meerdere klassen. Het voordeel van klassen is dat we meerdere variabelen kunnen bijhouden in één object. Hierdoor blijft de code overzichtelijk.

Binnen het spel zijn meerdere scenario's die aan elkaar gelinkt zijn. Omdat ieder scenario naar meerdere scenario's kan doorverwijzen, is het logisch om gebruik te maken van een graafstructuur, zoals weergegeven in Figuur 1. De graafstructuur lijkt op een decision tree, maar er zijn een paar verschillen. De overeenkomsten zijn dat ze beide een root-node hebben. Ook zijn beide structuren acyclisch en directed. Het grote verschil is dat een decision tree altijd maar één parent heeft terwijl een graafstructuur één of meer parents heeft. Daarom is de graafstructuur de beste structuur om hier te gebruiken.

Figuur 1 Graafstructuur

Ieder scenario (vertex) heeft een naam, beschrijving en scorecategorie. De naam wordt gebruikt voor het beheren van de scenario's. Door de scenario's een naam te geven is het scenario makkelijk terug te vinden als een beheerder een scenario wil aanpassen.

In de beschrijving van het scenario staat het daadwerkelijke scenario. Dit kan dus een vraag zijn als: “Je vriend(in) heeft het uitgemaakt. Wat doe je?”

De scorecategorie van het scenario geeft aan voor welke categorie de speler bij het scenario punten kan halen. Alle mogelijke categorieën die een scenario kan hebben worden los opgeslagen met een los object per categorie.

Ieder scenario bevat verder tussen de één en vier antwoorden. Deze antwoorden bevatten allereerst de inhoud van het antwoord. Een voorbeeld van een antwoord bij het eerdergenoemde scenario zou kunnen zijn: “Ik jank en ga met een bak ijs op de bank zitten.”

De antwoorden bevatten naast de inhoud van het antwoord ook het aantal punten wat dit antwoord oplevert en een verwijzing naar het vervolgscenario. Bijvoorbeeld: in Figuur 1 zal antwoord A een verwijzing bevatten naar scenario 2.

De speler kan één van deze antwoorden kiezen. De keuze die de speler hierin maakt, zal dus bepalen wat het volgende scenario wordt. Dit betekent dus ook dat de keuze van de speler bepaalt hoe de graaf doorlopen wordt. Ook bepaalt de keuze hoeveel punten de speler behaalt binnen de scorecategorie van het scenario. Zo wordt aan het einde van het spel duidelijk hoe de speler scoort per categorie en geeft de applicatie een indicatie hoe het mentaal gezien met de speler gaat.

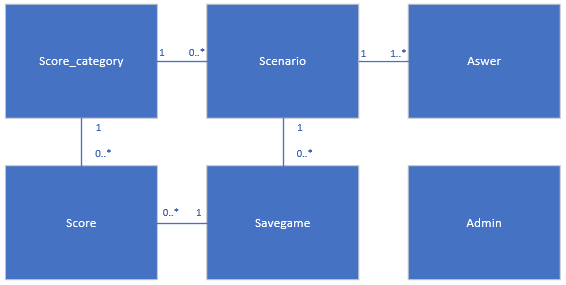
Naast de behaalde score van de speler, wordt ook de hoogst haalbare score opgeslagen. Deze hoogst haalbare score wordt op het einde gebruikt om aan de speler te laten zien hoe hij heeft gescoord. Omdat er veel verschillende manieren zijn om door de graaf te lopen kunnen deze maximale scores erg verschillen en moeten deze dus tijdens het spelen worden bijgehouden.

Iedere categorie waar de speler punten voor heeft gehaald worden samen met de maximale score voor die categorie opgeslagen in een “score”-object. Al deze objecten koppelen we aan elkaar door middel van één “savegame”-object.

De graafstructuur uit Figuur 1 mag geen cyclus bevatten, hierdoor zou het anders mogelijk worden om in een lus te komen tijdens het spelen en zou de speler dus nooit het einde kunnen bereiken. Dit soort graaf wordt een DAG (directed acyclic graph) genoemd.

Tot slot moet nog worden bijgehouden wie de beheerders van de app zijn en wie dus de scenario’s en categorieën mogen bewerken. Voor elke beheerder zal een beheerder object worden aangemaakt. Deze beheerders zijn ook binnen de applicatie te bewerken. Er wordt ook onderscheid gemaakt tussen beheerders en hoofdbeheerders, waarbij het verschil is dat hoofdbeheerders ook beheerders kunnen bewerken.

# Resultaten en realisatie

Er zijn meerdere klassen nodig om de applicatie te realiseren. Een overzicht van alle klassen en welke relatie ze tot elkaar hebben is te zien in Figuur 2. 

Figuur 2 Klassendiagram

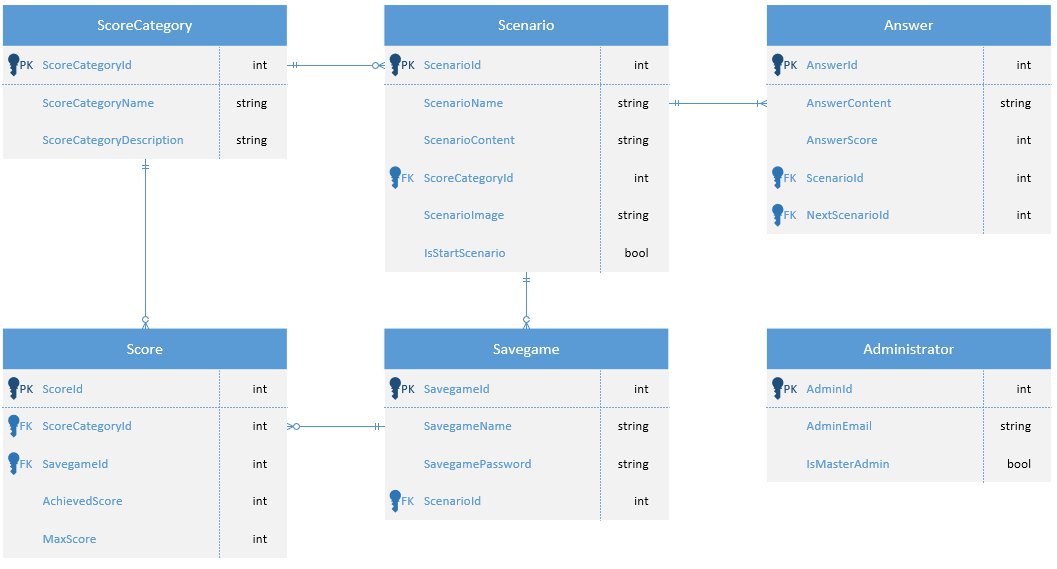
Centraal staat hier de “scenario”-klasse. Voor ieder scenario binnen het spel is er een instantie van deze klasse. Deze klasse bevat de inhoud van het scenario en tot welke scorecategorie deze behoort. Verder wordt er per scenario een boolean bijgehouden waarin wordt opgeslagen of het scenario het start-scenario is. Zo is er per scenario te achterhalen of dit de root is. Ieder scenario heeft ook enkele “answer”-objecten aan zich gekoppeld. Per “answer”-object staat een antwoord met daarbij de score die dat antwoord oplevert. Ook is er per “answer”-object een verwijzing naar het volgende scenario opgeslagen. Het “answer”-object verwijst altijd naar het id-nummer van het scenario. Als het “answer”-object verwijst naar een id-nummer van –1 dan is het einde van het spel bereikt.

Voor het bijhouden van de verschillende scorecategorieën wordt er gebruik gemaakt van de “score\_category”-klasse. Deze klasse bevat buiten een ID alleen de naam en beschrijving van de categorie.

Omdat een student meerdere keren tegelijk het spel kan spelen (meerdere ‘savegames’), wordt er voor iedere speelsessie een “savegame”-object aangemaakt. Dit object bevat een naam en optioneel een wachtwoord wat moet worden ingevoerd om verder te gaan met betreffende speelsessie. Ook staat er binnen een “savegame”-object een verwijzing naar het scenario waar de speler gebleven is. Aan deze savegame worden “score”-objecten gekoppeld. Voor iedere categorie waar de speler punten heeft gehaald, wordt de behaalde score bijgehouden, samen met de tot dan hoogst haalbare score voor deze categorie. Door voor iedere scorecategorie een los object aan te maken blijft het aantal scorecategorieën dynamisch.

In de “admin”-klasse wordt bijgehouden welke gebruikers beheerdersrechten hebben en welke beheerders hoofdbeheerders zijn. Dit wordt gedaan door middel van de Zuyd-e-mailadressen van administrator-accounts bij te houden. De beheerders hebben de rechten om binnen het spel de scenario’s, antwoorden en scorecategorieën te bewerken. De hoofdbeheerders hebben ook het recht om nieuwe beheerders toe te voegen en bestaande beheerders te verwijderen of te promoveren naar hoofdbeheerder. [Luister@zuyd.nl](mailto:Luister@zuyd.nl) zal hard-coded een hoofdbeheerder zijn.

Voor de implementatie van deze klassen is een ERD-diagram uitgewerkt, zie hiervoor Figuur 3.



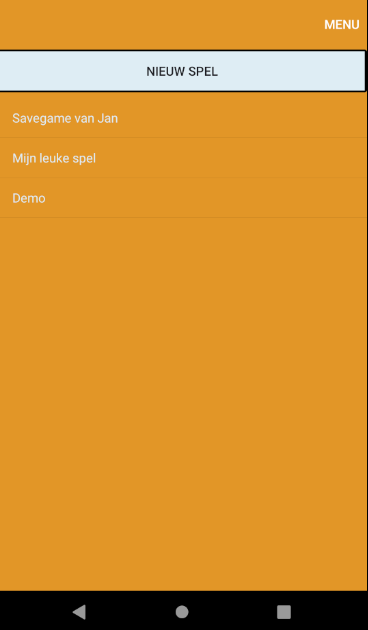
Figuur 3 ERD-diagram

Omdat we gebruik maken van een database zullen alle variabelen via het CRUD-principe (Create, Read, Update and Delete) te beheren moeten zijn. De Big O notaties van de tijdcomplexiteit voor deze bewerkingen is te zien in Tabel 1.

|  |  |
| --- | --- |
| **Bewerking** | **Big O** |
| Create | O(1) |
| Read | O(n) |
| Update | O(n) |
| Delete | O(n) |

Tabel 1 Big O complexiteit van CRUD-bewerkingen

Wanneer een speler het spel wil gaan spelen, krijgt deze eerst de keuze om een bestaande savegame te openen, of om een nieuwe savegame aan te maken, zie Figuur 4.



Figuur 4 Selecteren van een savegame

Wanneer er wordt gekozen om een nieuwe savegame aan te maken, wordt er een nieuw “savegame”-object aangemaakt. De speler kan een eigen avatar kiezen, zodat hij/zij zich beter kan inleven. Als de speler een bestaande savegame wil spelen, worden de gegevens van deze savegame opgehaald uit de database. Daarna wordt de gebruiker naar het eerste scenario gestuurd. Dit is het scenario waar de “IsStartScenario”-property op “true” staat. Er is altijd maar één scenario waar dit het geval is. Een voorbeeld van hoe een scenario eruitziet zie je in Figuur 5. Zoals te zien is staat bovenaan een afbeelding met daaronder het scenario en onderaan de keuzes waaruit de speler kan kiezen.



Figuur 5 Voorbeeld van een scenario

Wanneer een keuze wordt aangetikt worden een paar dingen uitgevoerd. Allereerst wordt gekeken welk antwoord de hoogste score had. Vervolgens wordt gekeken wat de score was van het gekozen antwoord. Deze beide waardes worden opgeteld bij de waardes in het “score”-object van de categorie van het scenario. Als er nog geen “score”-object is voor de betreffende categorie, wordt een nieuw object aangemaakt.

Nadat de scores zijn bijgewerkt wordt de speler doorgestuurd naar het volgende scenario. Om het volgende scenario te vinden wordt gekeken naar de “NextScenarioId”-property van het gekozen antwoord. Het is ook mogelijk dat dit het laatste scenario was (in dat geval is de waarde van de “NextScenarioId”-property -1). In dat geval wordt de speler naar het einde gestuurd.

De code die wordt uitgevoerd bij het maken van een keuze is te zien in Bijlage 1.

Op het eindscherm zijn de behaalde scores te bekijken, een voorbeeld hiervan is te zien in Figuur 6.



Figuur 6 Voorbeeld van een eindscherm

Bovenaan staat de totaal behaalde score met daaronder de scores per categorie. Onderaan staat een gepast advies aan de speler en zijn er links voor contact op te nemen met Zuyd L U I S T E R en om de workshops te bekijken. Voordat je contact opneemt met Zuyd L U I S T E R is er ook een optie om eerst een vragenlijst over bijvoorbeeld je interesses in te vullen zodat je gekoppeld kunt worden met een luisterstudent die bij jou past. De antwoorden van de ingevulde vragenlijst worden vergeleken met die van de luisterstudenten en zo wordt de speler gekoppeld aan de luisterstudent met de meeste raakvlakken.

# Discussie

Er is gekozen voor een DAG (directed acyclic graph). Om ervoor te zorgen dat de graaf acyclisch blijft, zou het mogelijk zijn om een algoritme te implementeren voor het checken op een cyclus. Een algoritme wat hiervoor gebruikt zou kunnen worden is het algoritme van GeeksForGeeks (GeeksForGeeks, 2020). Dit algoritme heeft een tijdcomplexiteit van , waarbij v het aantal scenario’s is en e het aantal verbindingen tussen de scenario’s. Deze implementatie is echter niet uitgewerkt binnen het prototype, omdat de prioriteiten voor het prototype op andere plekken lagen. Daarbij komt dat het implementeren van zo'n algoritme de tijdcomplexiteit niet ten goede komt. Dit zou namelijk betekenen dat hoe meer scenario's het spel bevat, des te langer het duurt om een scenario toe te voegen of te bewerken. Het algoritme moet immers een steeds grotere graaf controleren. Het ligt nu bij de beheerders om een oneindige loop te voorkomen. Het is binnen het prototype dus mogelijk dat de beheerders een graaf creëren die niet acyclisch is.

Binnen het prototype zijn de functies voor het aanmaken van een avatar en het invullen van een vragenlijst voor de koppeling met een luisterstudent niet uitgewerkt. De reden hiervoor is dat deze functionaliteiten geen prioriteit hadden binnen het prototype. Deze functionaliteiten zouden in de toekomst nog kunnen worden toegevoegd. Een andere functie die nog toegevoegd zou kunnen worden is het kiezen tussen verschillende startscenario’s, om zo meer variatie tussen speelsessies te creëren.

Verder wordt er binnen het prototype gebruik gemaakt van twee lokale databases. Dit is gedaan om de werkelijkheid zo dicht mogelijk te benaderen. Als deze applicatie wordt gerealiseerd wordt er gebruik gemaakt van één lokale database en één online database. De online database zou in dat geval alle gegevens bevatten over de scenario’s, scorecategorieën en beheerders. De lokale database bevat alle gegevens over de savegames en behaalde scores van de speler. Deze scheiding is er om de anonimiteit van de speler te waarborgen.

In het prototype kon geen online database gerealiseerd worden, vandaar dat er is gekozen voor twee lokale databases om deze scheiding te simuleren.

Een ander discussiepunt betreft het hard-coden van [luister@zuyd.nl](mailto:luister@zuyd.nl) als hoofdbeheerder. Het Zuyd L U I S T E R account staat hard-coded als hoofdbeheerder in de applicatie en kan dus niet verwijderd worden (als hoofdbeheerder). Als het account wordt gestolen of gehackt kan niemand dit account verwijderen als (hoofd)beheerder. Echter, aangezien dit een vertrouwelijk account is zou dit account goed beveiligd moeten zijn en is het account in het geval van een hack snel terug te halen.

# Conclusie

De graafstructuur die gekozen en geïmplementeerd is, de DAG (directed acyclic graph) is een gepaste datastructuur voor het bijhouden van de scenario’s. De keuzes die de speler maakt bepalen hoe de graaf doorlopen wordt. Door de gekozen datastructuur zijn er meerdere paden mogelijk om bij een einde te komen. Door de scenario’s centraal te zetten en hieraan de scorecategorieën en antwoorden te koppelen, blijft de code gestructureerd en overzichtelijk.

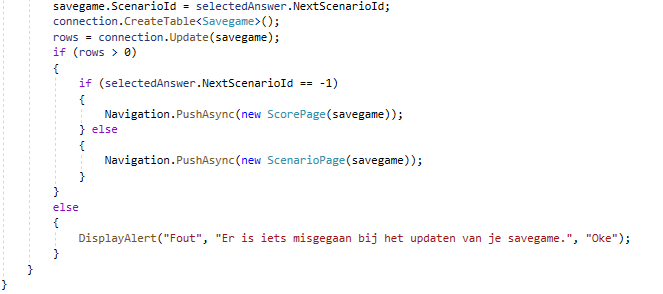
Hetzelfde geldt voor de savegames. Door de scores los te koppelen aan de savegame blijft ook hier de code gestructureerd en overzichtelijk. Daarbij komt ook dat hierdoor niet vast hoeft te staan wat de verschillende categorieën zijn waarvoor punten behaald kunnen worden.

Met al het bovenstaande in acht genomen wordt er aangeraden om voor de toepassing van een “kies je eigen verhaal”-spel, gebruik te maken van een directed acyclic graph datastructuur die gelijkenissen toont met een decision tree.

# Bijlagen

## Bijlage 1





Bovenstaande code doet stap voor stap het volgende: allereerst wordt het gekozen antwoord opgeslagen in een variable genaamd *selectedAnswer*. Vervolgens wordt verbinding gemaakt met de database en wordt de “Score”-tabel aangemaakt en uitgelezen. De uitgelezen scores worden opgeslagen in een list. In deze list wordt vervolgens gezocht naar de score van de categorie van het scenario voor de ingeladen savegame.

In de *if (currentScore.Count == 0)* wordt gekeken of er elementen zijn gevonden. Als er geen elementen zijn gevonden, betekent dit dat er voor de ingeladen savegame nog geen punten zijn behaald voor de categorie en moet een nieuw “score”-object worden aangemaakt. Dit wordt dan ook gedaan door de code die onder het if-statement staat.

Als er wel elementen zijn gevonden betekent dit dat er al eerder punten zijn behaald voor de categorie en de punten dus moeten worden opgeteld bij de al behaalde (en maximaal te behalen) punten. Dit gebeurt in het else-statement.

Als dit allemaal is uitgevoerd, gaat de code verder voor het doorsturen naar het volgende scenario. Hiervoor wordt de code *savegame.ScenarioId = selectedAnswer.NextScenarioId;* uitgevoerd. Deze regel geeft eigenlijk concreet aan dat het nieuwe huidige scenario, het scenario wordt waar het antwoord naar verwijst. Deze wijziging wordt vervolgens doorgevoerd in de database, zodat ook bij het opnieuw laden van de savegame de voortgang wordt behouden.

Tot slot wordt er nog een if-statement, *if (selectedAnswer.NextScenarioId == -1)* uitgevoerd om te controleren of de speler het spel heeft uitgespeeld. Als dat zo is, wordt de speler doorgestuurd naar de scorepagina. Anders wordt hij/zij doorgestuurd naar het volgende scenario.

De tijdcomplexiteit voor het succesvol toevoegen van een nieuwe score is . In Big O is dit dus .

Voor het succesvol bijwerken (ophogen) van een bestaande score is de tijdcomplexiteit . In Big O: . In beide vergelijkingen staat de n voor het aantal “score”-objecten in de database en m voor het aantal savegames.

# Verwijzingen

GeeksForGeeks. (2020, 3 27). *Detect Cycle in a Directed Graph - GeeksforGeeks*. Opgehaald van GeeksforGeeks: https://www.geeksforgeeks.org/detect-cycle-in-a-graph/#:~:text=To%20detect%20cycle%2C%20check%20for,a%20cycle%20in%20the%20tree.

Starmans, M., & Persoon, J. (2020). Casus Blok 2 2020-2021 B2C4 en B2F2.