# Software Design Description

## Provo

---

Bosman, Niels // 646983
El Kaddouri, Iliass // 658025

Version: 1.0 (12/03/2022)

# Inhoudsopgave

# 1 Introduction

## 1.1 Overall description

Een algemene beschrijving van dit project is reeds beschreven in hoofdstuk 1.1 van het Software Requirements Specification.

## 1.2 Purpose of this document

In dit document wordt het ontwerp van de Provo applicatie uitvoerig beschreven. Hierbij worden de ontwerpkeuzes onderbouwd met onder andere (System) Sequence Diagrammen en een design class diagram. Tevens worden een aantal belangrijke ontwerpkeuzes uitvoerig beschreven en beargumenteerd.

## 1.3 Definitions, acronyms and abbreviations

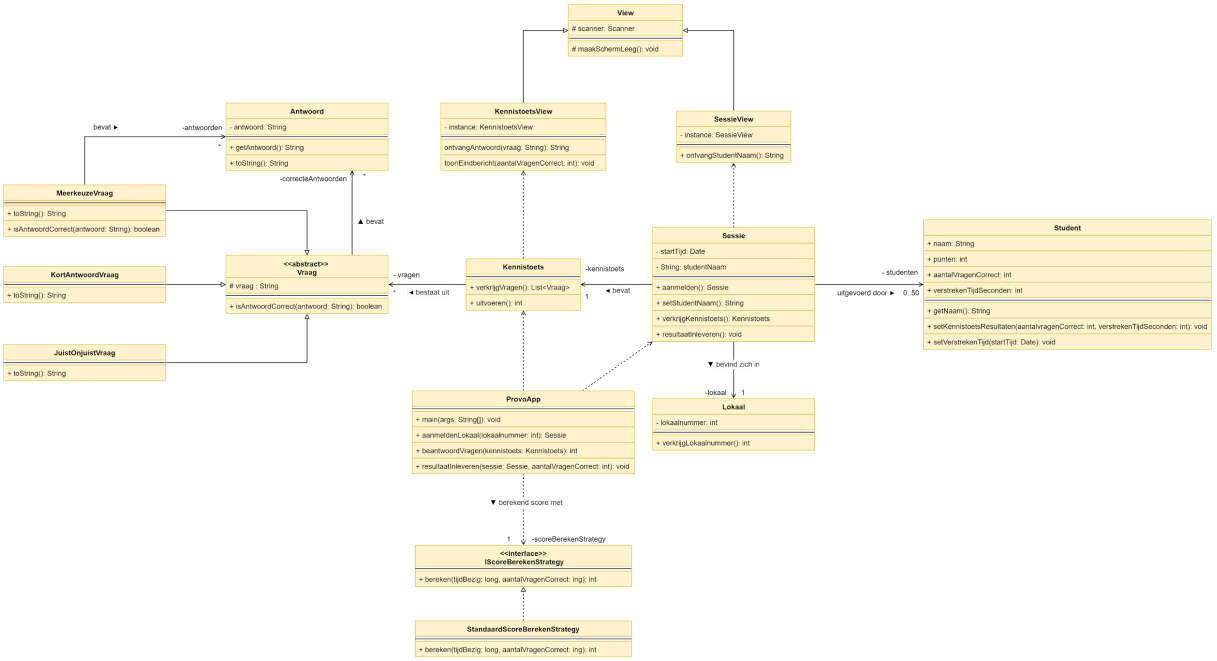
| Term | Description |
|---|---|
| GRASP | GRASP is een Engels acroniem dat staat voor Generalized Responsibility Assignment Software Patterns/Principles. Het bestaat uit 9 richtlijnen die kunnen worden gebruikt om in objectgeoriënteerde systemen verantwoordelijkheden toe te kennen aan klassen of objecten. Elke richtlijn behandelt een combinatie van een veelvoorkomend probleem en een algemeen toepasbare oplossing voor dat probleem met betrekking tot het ontwerp van softwaresystemen.[1] |
| SOLID | SOLID (Single responsibility, Open-closed, Liskov substitution, Interface segregation and Dependency inversion) is een acroniem dat gebruikt wordt in verband met het programmeren van computers.[2] |
| GOF | Design Patterns are a software engineering concept describing recurring solutions to common problems in software design.<br>The authors Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides are often referred to as the GoF, or Gang of Four.[3] |

[1] https://nl.wikipedia.org/wiki/GRASP
[2] https://nl.wikipedia.org/wiki/SOLID
[3] https://www.journaldev.com/31902/gangs-of-four-gof-design-patterns

# 2 Detailed Design Description

## 2.1 Design Provo Game

### 2.1.1 Design Class Diagram




Figuur 2: Design Class Diagram.

#### 2.1.1.1 Afwijkingen domeinmodel

Bovenstaande Design Class Diagram is opgesteld op basis van de geïmplementeerde use case *"uitvoeren kennistoets"*. Om deze reden zijn er concepten die niet nodig zijn voor het uitvoeren van een kennistoets achterwege gelaten, het betreft de klassen voor *"Account"*, *"BasisAccount"*, *"PremiumAccount"*, *"BetalingsProvider"* en *"Docent"*. Om te demonstreren dat wij begrijpen hoe het strategy pattern werkt en gemodelleerd moet worden hebben wij voor de use case *"genereer overzicht"* een *IScoreBerekenStrategy* interface en een *StandaardScoreBerekenStrategie* klasse toegevoegd. Door deze strategy pattern is het gemakkelijk om aan de eis te voldoen de puntentelling zo flexibel mogelijk te maken zonder enig implementatie detail te tonen.

Daarbij zijn er in dit Design Class Diagram de *"KennistoetsView"* en de *"SessieView"* toegevoegd. Het doel van deze klassen zijn om de UI logica te scheiden van de business logic van de app. Deze klassen erven van "View" en hebben beide de taak input van een kennistoets-afnemer af te ontvangen.

Sessie is in dit voorbeeld vergelijkbaar met het associatie-concept *"KennistoetsSessie"* geïmplementeerd in het domeinmodel.

#### 2.1.1.2 MVC Pattern en Domain Driven Design

Er is gekozen voor een aangepaste versie van het MVC pattern. De overeenkomst met MVC is dat er een duidelijke scheiding is tussen de Views en Models. Wat echter ontbreekt zijn controllers.

We hebben dit met opzet gedaan, omdat de laatste jaren DDD[4] erg opkomend en populair aan het worden is. De principes van DDD beschrijven dat elk domein model, bijvoorbeeld een kennistoets, moet beschikken over alle attributen en methoden die te maken hebben met een kennistoets. Wanneer men deze principes volgt, kom je al gauw tot de conclusie dat controllers overbodig zijn. Tevens is dit ook voor de ontwikkelaar erg handig, omdat de code natuurlijk en intuïtief georganiseerd is.

[4] https://domaindrivendesign.org/ddd-domain-driven-design/

## 2.1.2 Sequence diagrams

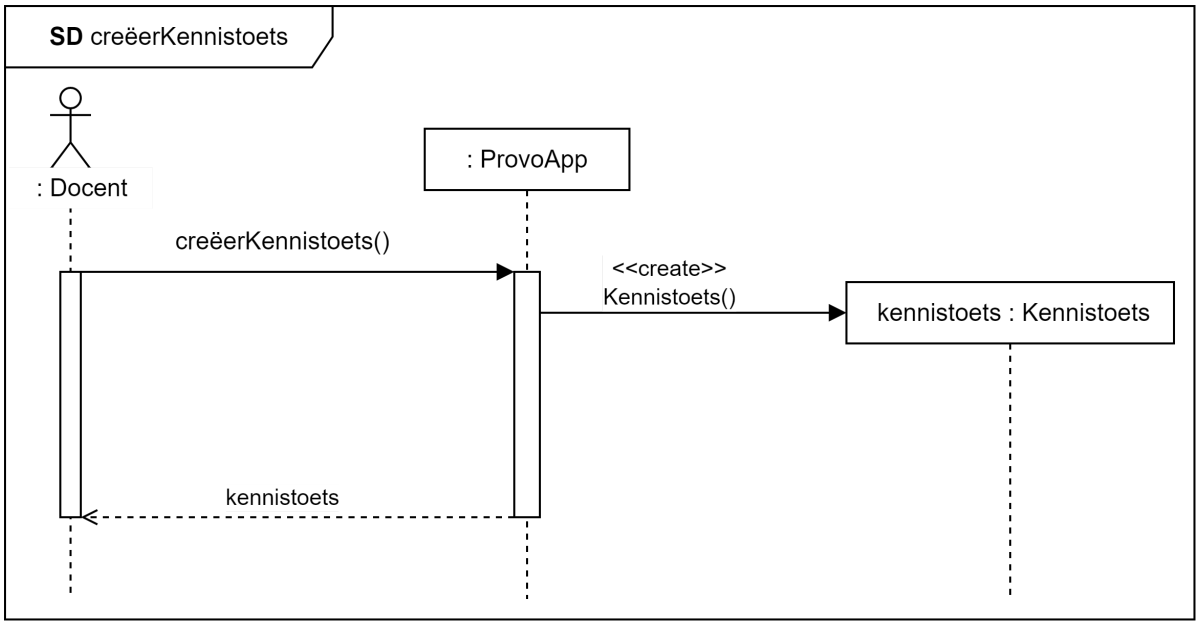
Onderstaande diagrammen presenteren de volledige flow voor de 4 belangrijkste use cases, welke zijn beschreven in hoofdstuk 3 van het Provo - SRS. Er zijn echter verschillen tussen de diagrammen en de werkelijke implementatie.

Zo hebben we alleen nuttige klassen gemodelleerd, omdat de Sequence Diagrammen handig zijn om de algemene flow van een applicatie weer te geven zonder daarbij implementatie specifieke keuzes in mee te nemen. Een concreet voorbeeld hier is de *View* klasse, welke overgeërfd wordt door *KennistoetsView* en *SessieView*. Deze klasse bevat twee methodes voor het correct representeren van de applicatie, het is echter niet noodzakelijk.

Tot slot hebben we een *Database* klassen, welke fungeert als een database. In de context van deze applicatie kun je *Database* als een blackbox beschouwen.

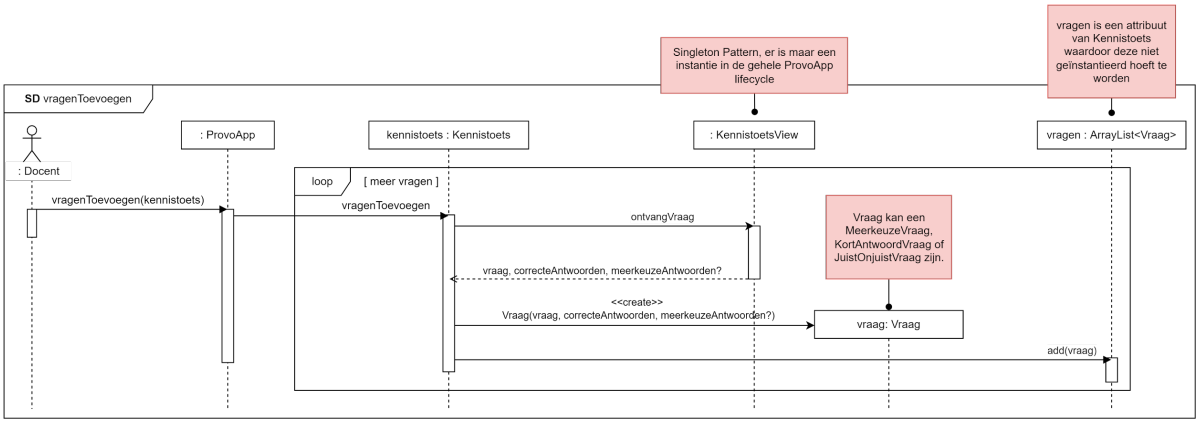
### 2.1.2.1 Aanmaken kennistoets

#### 2.1.2.1.1 Creëer kennistoets



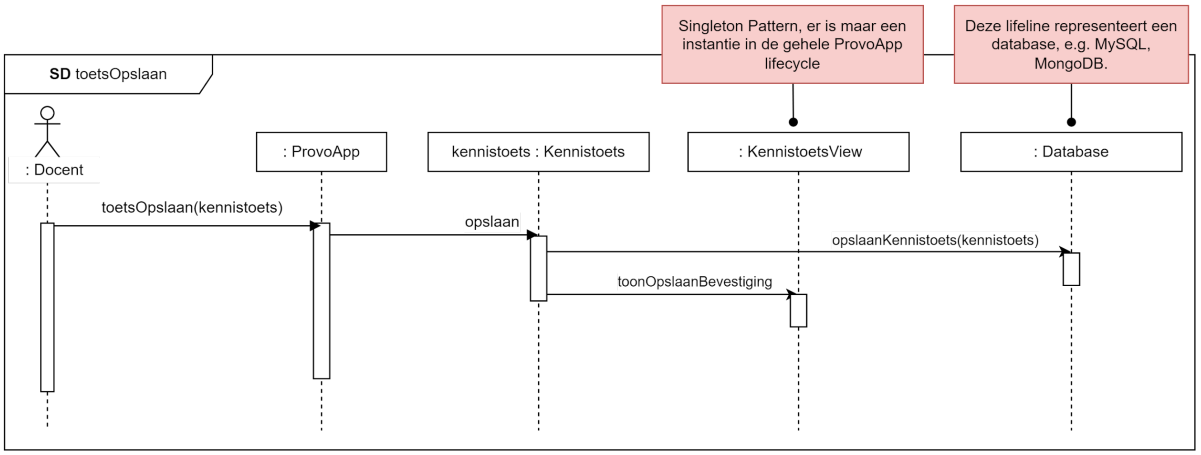

Figuur 3: Sequence diagram creëerKennistoets

#### 2.1.2.1.2 Vragen toevoegen




Figuur 4: Sequence Diagram *vragenToevoegen(kennistoets)*.

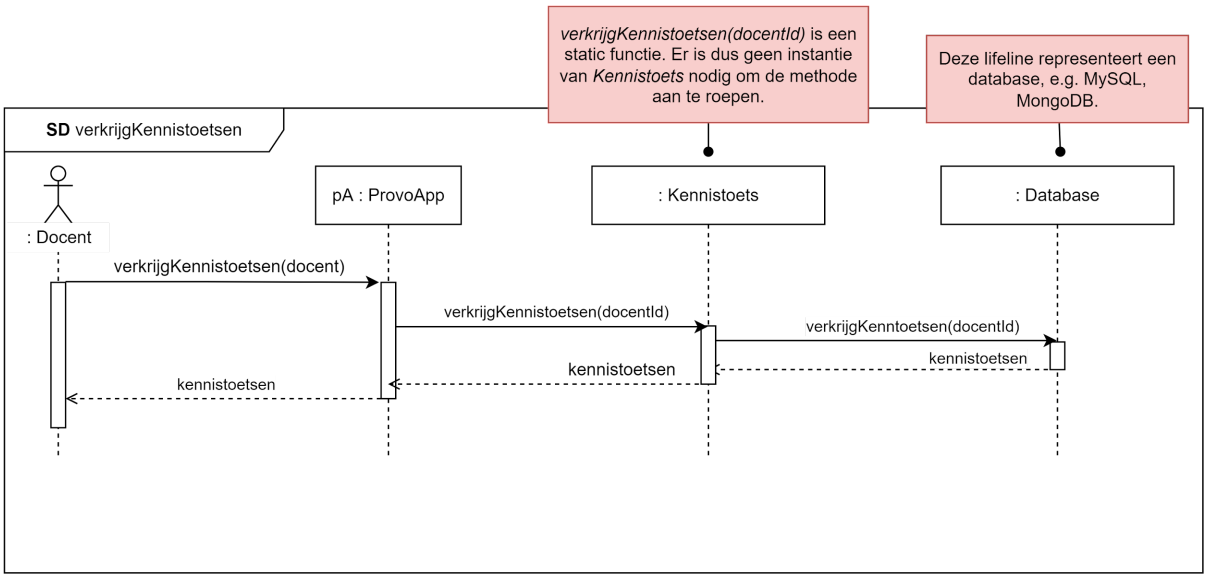
### 2.1.2.1.3 Toets opslaan




Figuur 5: Sequence Diagram *toetsOpslaan(vragen)*

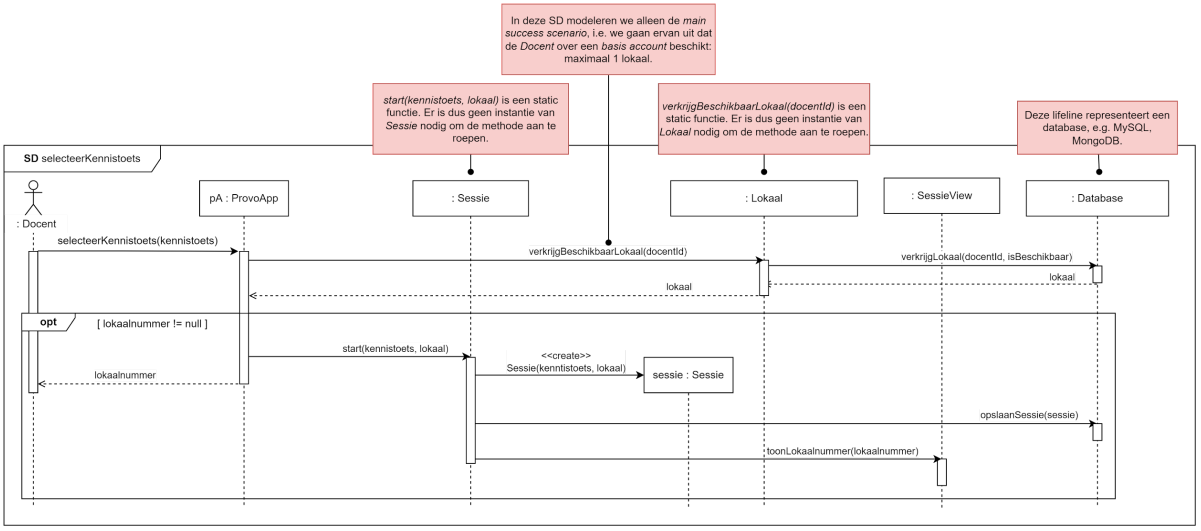
## 2.1.2.2 Starten kennistoets

### 2.1.2.2.1 Verkrijg kennistoetsen




Figuur 6: Sequence Diagram *verkrijgKennistoetsen(docent)*

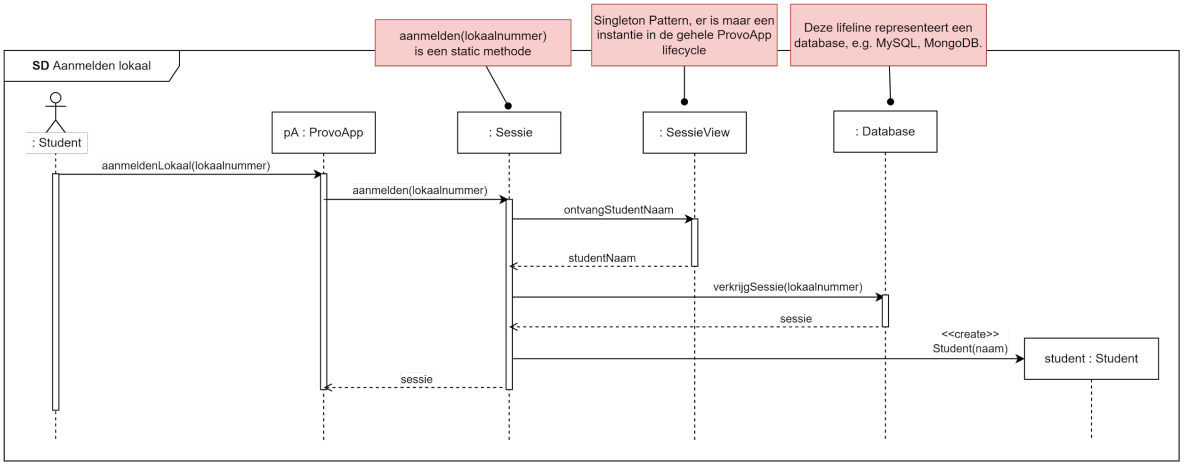
### 2.1.2.2.2 Selecteer kennistoets




Figuur 7: Sequence Diagram voor *selecteerKennistoets(kennistoets)*

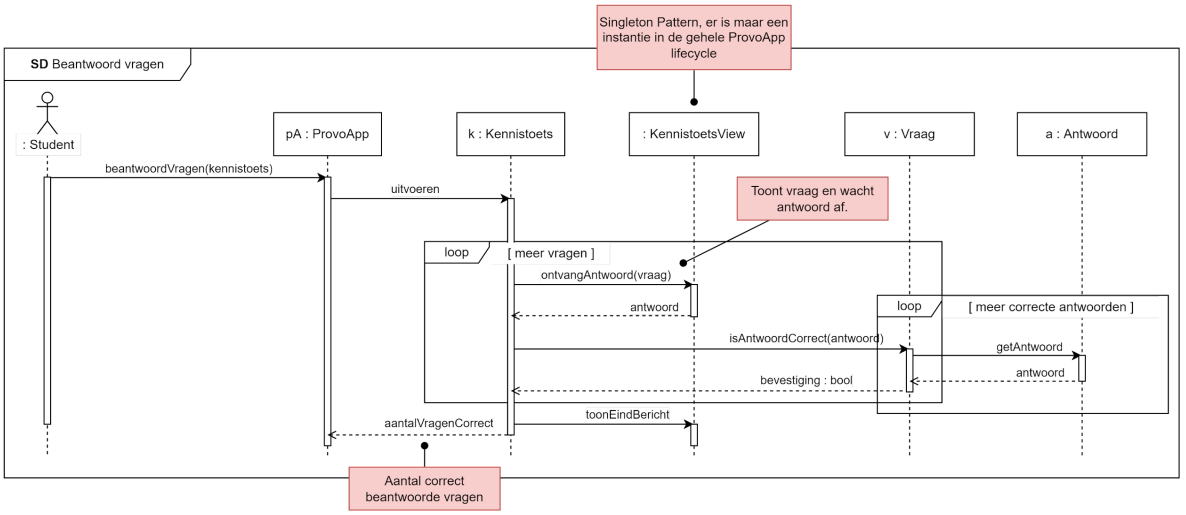
### 2.1.2.3 Uitvoeren kennistoets

#### 2.1.2.3.1 Aanmelden lokaal



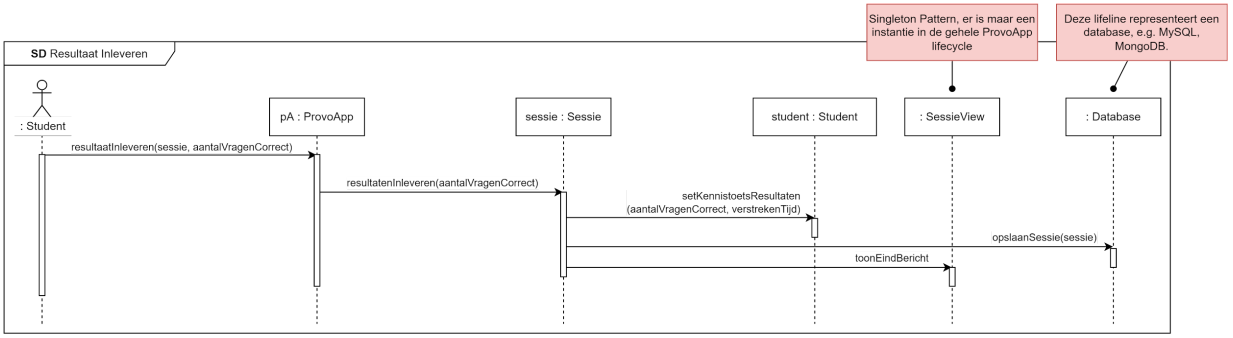

Figuur 8: Sequence Diagram voor *aanmeldenLokaal(lokaalnummer)*

#### 2.1.2.3.2 Beantwoord vragen



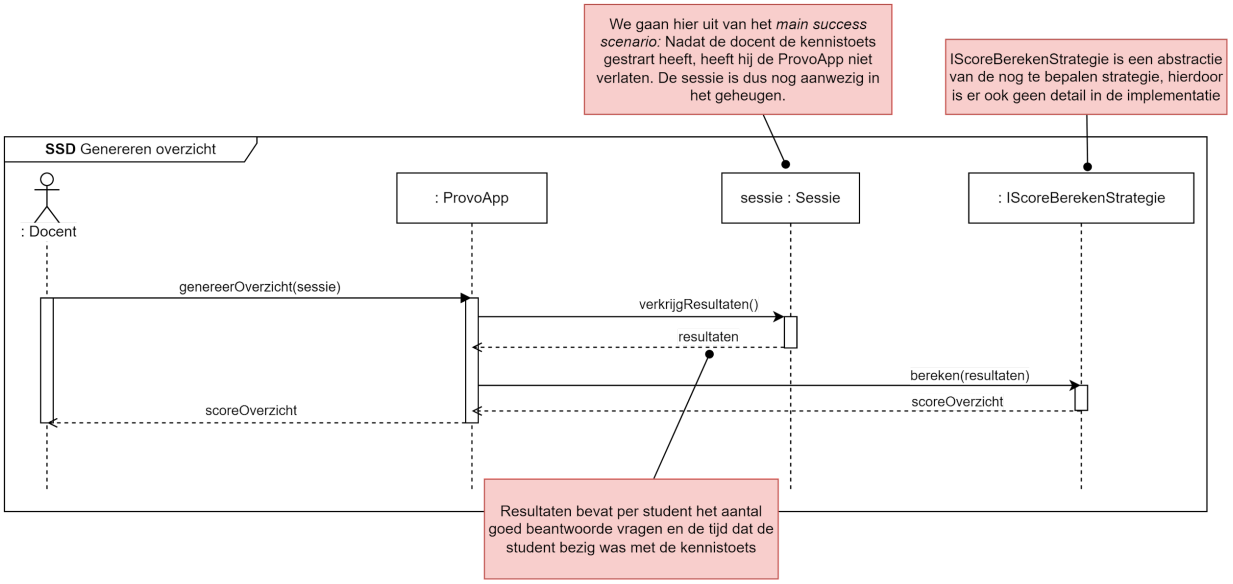

Figuur 9: Sequence Diagram voor *beantwoordVragen(kennistoets)*.

#### 2.1.2.3.3 Resultaat inleveren




Figuur 10: Sequence Diagram voor *resultaatInleveren(kennistoets, naam)*.

### 2.1.2.4 Genereren overzicht




Figuur 11: Sequence Diagram voor *genereerOverzicht(sessie)*.

### 2.1.3 Design decisions made

Eerder in dit document zijn er al een aantal design decisions genoemd. Dit hoofdstuk zal dieper ingaan op de ontwerpkeuzes met betrekking tot de geleerde design principles en patterns.

#### 2.1.3.1 GRASP

TIjdens het ontwerp en ontwikkelproces van de Provo applicatie is er rekening gehouden met de negen richtlijnen van GRASP. Elk principe zal apart worden elaboreert.

##### 2.1.3.1.1 Low coupling and high cohesion

We hebben dit principe toegepast door elke klasse logische verantwoordelijkheden te geven, zodat elke klasse een minimaal aantal dependencies heeft. Een specifiek voorbeeld hiervan is de klasse *Kennistoets* verantwoordelijkheid heeft over *Vraag*, omdat logischerwijs een kennistoets vragen bevat en gebruikt. Alleen *Kennistoets* heeft een dependency naar *Vraag,* wat zorgt voor low coupling. Er is weinig impact wanneer *Vraag* aangepast wordt.

##### 2.1.3.1.2 Creator

*Sessie* maakt een *Student* aan, omdat een *sessie* object alle deelnemende studenten bijhoudt. *Sessie* is ook de enige die een dependency heeft naar *Student*. Tevens gebruikt *Sessie Student* voor het bijhouden van kennistoets resultaten en het berekenen van de score.

##### 2.1.3.1.3 Information expert

De klasse *Kennistoets* is tevens de klasse die ook de kennistoets daadwerkelijk uitvoert, omdat deze klasse alle informatie bevat om het uit te voeren. Het bevat alle vragen en diens verantwoordelijkheden en een *KennistoetsView* om te communiceren met de Actor.
De klasse *Vraag* hebben methodes *toString* en *isAntwoordCorrect,* omdat het de enige klasse is die kennis heeft over *Antwoord*.

##### 2.1.3.1.4 Protected variations

Bij het implementeren van de variabele puntentelling strategieën is er gebruik gemaakt van een interface, om zo het aankooppunt hetzelfde te houden wanneer het gedrag van de puntentelling strategie veranderd. Hierdoor kunnen er gemakkelijk strategieën worden toegevoegd of gewisseld zonder (hele veel) code aan te passen.

##### 2.1.3.1.5 Polymorphism

Om de verschillende soorten vragen te implementeren is er gebruik gemaakt van Polymorfisme, om de gewenste functionaliteiten voor elke soort vraag te garanderen. Hierdoor hoeft er geen kennis te zijn van het type vraag en kan toch de gewenste functionaliteit uitgevoerd worden, een voorbeelden hiervan zijn *toString* en *controleerAntwoord*.

#### 2.1.3.2 SOLID Design principles

##### 2.1.3.2.1 Single-responsibility principle

Om er voor te zorgen dat er geen klassen ontstaan die meerdere taken vervullen hebben we onze applicatie zo ontworpen dat alle taken van een klasse logisch zijn voor die klasse. Zo is het bijvoorbeeld zo dat onze *Kennistoets* klasse alleen het doel heeft om een kennistoets uit te voeren en de vragen die hierbij horen op te halen. Door dit zo te ontwikkelen zorgen we ervoor dat we lagere coupling hebben met andere klassen.

#### 2.1.3.2.2 Open-closed principle

In het ontwerp van het systeem hebben we de keuze gemaakt om het score berekenen door middel van een strategy pattern toe te passen. Dit strategy pattern zorgt ervoor dat ons systeem closed voor modificatie is doordat alle verschillende mogelijke strategieën het IScoreBerekenStrategy interface moeten implementeren. Hierdoor is het ook erg open voor extensie omdat er gemakkelijk een nieuwe strategie kan worden gemaakt door simpelweg dit interface te implementeren.

#### 2.1.3.2.3 Dependency inversion

Om er voor te zorgen dat de vragen zo flexibel mogelijk gebruikt kunnen worden hebben wij in het ontwerp van de Provo Game er voor gekozen Vraag een abstracte klasse te maken. Dit zorgt ervoor dat in plaats van direct de verschillende type vragen te gebruiken (*MeerkeuzeVraag*, *KortAntwoordVraag* en *JuistOnjuistVraag*) we de mogelijkheid hebben om alleen een dependency te hebben naar de abstracte klasse *Vraag.* Het voordeel hiervan is dat we gemakkelijk van type vragen kunnen veranderen zonder dat onze *Kennistoets* klasse hier direct last van heeft.

### 2.1.2.3 GoF Patterns

#### 2.1.2.3.1 Strategy Pattern

Educom had het verzoek eenvoudig van puntentelling strategie te kunnen wisselen. Om aan deze requirement te voldoen hebben we het Strategy pattern toegepast. In het Design Class Diagram is te zien dat ProvoApp een *IScoreBerekenStrategy* bevat. Zo kan er eenvoudig gewisseld worden van puntentelling strategie zonder enige logica aan te passen: Open close principe van SOLID.

#### 2.1.2.3.2 Facade Pattern

Elke klasse die moet communiceren met de Actor heeft een eigen View. Zo bevat *Kennistoets KennistoetsView* en *Sessie SessieView.* Zo wordt er voldaan aan het Facade pattern: De actor communiceert alleen door middel van deze *View* klassen.