De Medische Vragenlijst

Eindverslag Bacheloreindproject IN3405

Versie 1.1

Onderzoeksbureau Soffos
TU Delft Faculteit EWI

Auteurs:
C.W. Ipema (1509195)
S. Oosterwaal (1509322)
D.A.A. Pelsmaeker (1367838)

Beoordelingscommissie:
dr. M.T.I.W. Schüsler-van Hees
ir. H.J.A.M. Geers
ir. B.R. Sodoyer

15 juli 2011

Voorwoord

Als afsluiting voor de bacheloropleiding Technische Informatica aan de TU Delft, is het vereist om een project in de vorm van een stage uit te voeren. Dit project moet in groepsverband worden uitgevoerd, in opdracht van een bedrijf en onder begeleiding van de TU Delft. De projectgroep van dit eindverslag bestaat uit drie studenten: Charlotte Ipema, Sebastiaan Oosterwaal en Daniël Pelsmaeker. Het project is uitgevoerd in de periode 18 april tot en met 15 juli 2011 en is voor en in samenwerking met het Onderzoeksbureau Soffos tot stand gekomen.

Graag nemen wij de gelegenheid om een aantal personen te bedanken. Allereerst willen we ir. H.J.A.M. Geers en dr. E. Dolstra bedanken voor het begeleiden van ons project vanuit de TU Delft. Verder willen de volgende personen van Soffos bedanken: dr. M.T.I.W. Schüsler-van Hees voor de algemene begeleiding, O.M. Schüsler BSc voor de technische begeleiding, en drs. J.F. Schüsler en mr.drs. T.T. Schüsler voor de uitgebreide ondersteuning en de mogelijkheid dit project uit te voeren.

Samenvatting

Onderzoeksbureau Soffos gebruikt onder andere vragenlijsten om het effect van een interventie te meten. Om de werklast van het huidige systeem te verminderen is het doel van dit project het ontwikkelen van een nieuwe back-end serverapplicatie voor het automatiseren van het beheer, beschikbaar stellen en verwerken van onder andere vragenlijsten.

De serverapplicatie is modulair opgebouwd en kent een rollen-gebaseerd authenticatiesysteem, meerdere talen, en gebruikt een MySQL database om alle gegevens in op te slaan. De communicatie met een front-end verloopt met JSON via REST. Dankzij programmeerbibliotheken en hulpmiddelen van derden, zoals Python, Eclipse en de SQLAlchemy ORM werd het programmeren van de applicatie eenvoudiger.

Het was voor de projectgroep een uitdaging om te werken voor een bedrijf en alle requirements eenduidig op tafel te krijgen. De serverapplicatie is ontwikkeld in 13 weken volgens de Agile methodiek gebruik makend van Test-Driven Development en is op één onderdeel na succesvol binnen de planning afgerond.

Definities

Actor
Gebruiker van het systeem. Gebruikers kunnen meerdere rollen vervullen.

Admin
Rol in de serverapplicatie met de meeste gebruikersrechten.

Back-end
Serverapplicatie met een front-end als interface. Handelt alle bedrijfslogica af.

Boolean
Waarde die waar of onwaar kan zijn.

Cliënt
Persoon bij wie een interventie plaats vindt. Vult over het effect daarvan vragenlijsten in.

Dictionary
Lijst met $sleutelwoord = waarde$ combinaties.

Dimensie
Parameter die gemeten kan worden.

Extensie
Verzameling objectief vergaarde gegevens, zoals bloeddruk of hartslag. Wordt ingevuld door een user over een individuele cliënt.

Extensievraag
Vraag die alleen in extensies voor kan komen.

Front-end
Interface voor een back-end. Bevat geen bedrijfslogica.

Functionele requirements
Criteria die de werking van specifieke onderdelen van een applicatie beschrijven.

Groepsrapport
Rapport met resultaten van vragenlijsten over een door de user gedefinieerde groep van cliënten.

HTTP
HyperText Transport Protocol, het protocol van het World Wide Web.

HTTP errorcode
Een HTTP statuscode die een fout aangeeft.

HTTP statuscode
Code die aangeeft of een actie succesvol is uitgevoerd, en zo niet, wat het probleem was.

Identifier
Getal dat uniek is in een databasetabel en ondubbelzinnig een databaserij identificeert. Identifiers worden ook gebruikt om de objecten die corresponderen met een databaserij te identifieren.

Individueel rapport
Rapport met betrekking tot één of meerdere vragenlijsten betreffende een individuele cliënt.

Integer
Geheel getal.

Interface
Geheel van bedieningselementen en statusberichten waarmee een actie uitgevoerd kan worden.

Interventie
Geheel van activiteiten waarmee users problemen kan voorkomen of oplossen om zo een bijdrage te leveren aan het bevorderen en in stand houden van gezondheid en welzijn van cliënten.

JSON
JavaScript Object Notation, een taal om objectrepresentaties mee uit te wisselen.

Meetinstrument
Verzamelnaam voor extensies en vragenlijsten die worden gebruikt voor het verkrijgen van meetgegevens.

Niet-functionele requirements
Criteria waarmee de kwaliteit van een applicatie beoordeeld kan worden.

Opdrachtgever
Organisatie, bedrijfsafdeling of persoon die met Soffos een contract heeft afgesloten voor het uitvoeren van een project.

ORM
Object-relational mapping, gebruikt om database tabellen te relateren aan objecten.

PHP
Programmeertaal gebruikt voor het ontwikkelen van de test website.

Project
Geheel van deelnemende users en cliënten en vastgestelde vragenlijsten en extensies.

Python
Programmeertaal waarin de serverapplicatie geschreven is.

Query
Vraag naar specifieke gegevens aan een gegevensbeheersysteem, geformuleerd volgens een bepaald format.

Resource
Object in de serverapplicatie die een URL heeft en bereikt kan worden door middel van een request.

Request
Aanvraag aan een resource met een bepaalde URL volgens het HTTP protocol. De vier meest gebruikte requesttypes zijn *GET*, *POST*, *PUT* en *DELETE*.

Requirements
Eisen die aan een applicatie worden gesteld. Deze zijn op te delen in *functionele* en *niet-functionele requirements*.

Serialiseren
Converteren van een object naar een bepaalde codering zodat het object op een ander moment of op een andere machine weer uit de codering afgeleid kan worden.

SIG
Software Improvement Group, een bedrijf gespecialiseerd in de analyse van de kwaliteit en onderhoudbaarheid van software.

SSL
Secure Sockets Layer, een protocol waarmee gegevens versleuteld worden.

String
Eén of meer regels tekst in een programmeertaal.

SVN
Subversion, een versiebeheer systeem.

Tijdslot
Periode die per vragenlijst of extensie is ingesteld waarbinnen de data moeten worden aangeleverd.

User
Professional in het medisch domein die interventies pleegt.

Vragenlijst
Lijst met vragen aan een cliënt met als doel het meten van het effect van een interventie.

Vragenlijstvraag
Vraag die alleen in één vragenlijst voor kan komen.

Inhoudsopgave

1 Introductie **1**

I Probleemstelling en domeinanalyse **2**

2 Probleemstelling **3**
2.1 Huidige situatie . . . 3
2.2 Doelstelling . . . 3
2.3 Opdrachtformulering . . . 3

3 Domeinanalyse **5**
3.1 Het bedrijf . . . 5
3.2 De activiteiten . . . 6
3.3 Eigenschappen van vragenlijsten en rapporten . . . 8

II Proces **10**

4 Methodiek **11**
4.1 Agile . . . 11
4.2 Test Driven Development . . . 11

5 Planning **13**
5.1 Planning vooraf . . . 13
5.2 Werkelijke uitvoering . . . 14

6 Problemen en uitdagingen **16**
6.1 Server . . . 16
6.2 Werken voor een bedrijf . . . 16
6.3 Agile . . . 17

7 Hulpmiddelen **18**
7.1 Python . . . 18
7.2 Eclipse . . . 18
7.3 ORM . . . 18
7.4 SVN . . . 19

III Ontwerp en implementatie 20

8 Ontwerp 21
8.1 Model . . . 21
8.2 Databaseontwerp . . . 26
8.3 ORM . . . 29
8.4 REST communicatie . . . 29
8.5 Modules . . . 30

9 Implementatie 32
9.1 Communicatie tussen back-end en front-end . . . 32
9.2 Requirements terugzoeken . . . 33
9.3 Technische bijzonderheden . . . 33
9.4 SOLID . . . 38
9.5 Code conventies . . . 40

10 Codekwaliteit 41
10.1 Unittests . . . 41
10.2 Gebruikerstest . . . 41
10.3 SIG beoordeling . . . 42
10.4 Pylint . . . 42

IV Resultaten en aanbevelingen 44

11 Resultaten 45
11.1 Geschreven code . . . 46

12 Aanbevelingen 47
12.1 Onze aanbevelingen . . . 47
12.2 Toekomstig werk . . . 47

V Bijlagen 49

A Plan van aanpak 50
A.1 Inleiding . . . 50
A.2 Opdrachtomschrijving . . . 50
A.3 Aanpak . . . 53
A.4 Projectinrichting . . . 54
A.5 Kwaliteitsborging . . . 54

B Functional requirements 56

C Requirements implementation 60

D Orientationreport 61
D.1 Introduction . . . 61
D.2 SOLID programming in Python . . . 61
D.3 Back-to-front communication . . . 64
D.4 REST framework for Python . . . 65

D.5 MySQL library for Python . . . 67
D.6 Unit test framework for Python . . . 67
D.7 Documentation library for Python . . . 68
D.8 Inversion of Control framework for Python . . . 68

E Communication specification **71**

F Mission spectrum **72**
F.1 About the company . . . 72
F.2 Current Situation . . . 72
F.3 Problem . . . 72

Lijst van figuren

4.1 Implementatiecyclus van een feature volgens Test-Driven Development. 12

5.1 Gantt-chart van de planning zoals deze vooraf was vastgesteld. 14
5.2 Gantt-chart van de planning zoals deze achteraf was. 14

8.1 De Admin, User en Client rollen bij de Actor. 22
8.2 De vragenlijsten en extensies en hun instanties. 23
8.3 De groepen waarin cliënten kunnen zitten. 24
8.4 De mails en hun relatie met de actoren. 25
8.5 De gekozen oplossing voor het modelleren van teksten in meerdere talen. 25
8.6 De alternatieve oplossing voor het modelleren van teksten in meerdere talen. . . 26
8.7 Databaseontwerp: projecten, actoren, admins, users en cliënten 27
8.8 Databaseontwerp: meetinstrumenten (vragenlijsten en extensies) samen met hun instanties en antwoorden 28
8.9 Databaseontwerp: E-mails voor actoren 29
8.10 Overzicht van de modules en hun onderlinge relaties. 31

9.1 Illustratie van het volgordealgoritme. 34
9.2 Illustratie van het gijzelaarssysteem. 36
9.3 Sequence diagram van een standaard HTTP request. 37

11.1 De verschillende soorten geschreven code. 46

Hoofdstuk 1

Introductie

Hoe effectief is de nieuwe behandeling die uw dokter op u toepast? Werkt acupunctuur echt? Hebben de nieuwe medicijnen het gewenste effect en geen nare bijwerkingen? "*Meten is weten*", is dan een veelgehoorde reactie. Onderzoeksbureau *Soffos* uit Rijen probeert antwoord te vinden op dit soort medische vragen door middel van vragenlijstenonderzoek onder cliënten. Door cliënten bij wie een interventie plaats vindt vooraf en achteraf te vragen om een vragenlijst in te vullen, kan er een beeld worden gevormd van de effectiviteit daarvan. De vragenlijsten worden nu via de e-mail afgenomen en de handmatige verwerking hiervan kost veel tijd. In sommige gevallen hebben cliënten moeite om de e-mailvragenlijst te openen, correct in te vullen en terug te sturen. Vanwege deze redenen is aan ons gevraagd om een applicatie te bouwen waarmee via een website vragenlijsten makkelijker kunnen worden aangemaakt, toegewezen, ingevuld en verwerkt, om zo de werklast te verminderen.

Dit eindverslag bestaat uit de volgende delen:

- Deel I beschrijft de achtergronden van het probleem en de te ontwikkelen applicatie. De probleemstelling wordt kort besproken in hoofdstuk 2, terwijl de huidige situatie uiteen wordt gezet in de domeinanalyse in hoofdstuk 3.
- Deel II gaat over het proces van ontwerp tot implementatie. Hierin wordt de gebruikte ontwikkelmethodiek besproken in hoofdstuk 4, de planning in hoofdstuk 5, de problemen en uitdagingen in hoofdstuk 6 en de gebruikte hulpmiddelen in hoofdstuk 7.
- Deel III behandelt het ontwerp in hoofdstuk 8, de implementatie in hoofdstuk 9 en de codekwaliteit in hoofdstuk 10.
- Deel IV de resultaten besproken in hoofdstuk 11, en de aanbevelingen in hoofdstuk 12.
- Deel V bevat de bijlagen, met het plan van aanpak in bijlage A, de functionele requirements in bijlage B, de implementatie van de requirements in bijlage C, het oriëntatierapport in bijlage D en de communicatie specificatie in bijlage E.

Deel I

Probleemstelling en domeinanalyse

Hoofdstuk 2

Probleemstelling

Dit hoofdstuk behandelt de huidige situatie en de problemen die dit oplevert, gevolgd door het doel en de opdrachtomschrijving van dit project.

2.1 Huidige situatie

Via de website www.soffos.nl konden users zich registreren en werden vragenlijsten afgenomen bij cliënten. Deze site is een aantal jaar geleden gebouwd en voldoet niet meer om het groeiende aantal projectaanvragen af te handelen. Toen Soffos op enig moment een indicatie kreeg dat de website niet meer 100% betrouwbaar zou zijn, is er besloten om de website niet meer te gebruiken en de vragenlijsten af te nemen per e-mail.

Het overnemen en verwerken van de resultaten van de website en de e-mails is een arbeidsintensief proces. Momenteel wordt hiervoor een groot excel-bestand gebruikt, waar met diverse berekeningen de uiteindelijke resultaten verkregen kunnen worden. Dit excel-bestand levert rapporten aan in een vooraf vastgestelde vorm. Nu het onderzoeksveld uitgebreid is, en om de werklast te verminderen, zal er een nieuw systeem moeten komen.

2.2 Doelstelling

De doelstelling van het project is om de werklast van het aanbieden en verwerken van vragenlijsten en extensies aan cliënten en users te verminderen. Uiteindelijk moet Soffos – naast onderhoud en gebruikersondersteuning – alleen verantwoordelijk zijn voor het invoeren van nieuwe vragenlijsten en extensies, het versturen van de rekeningen en het schrijven van statistische rapporten en wetenschappelijke artikelen.

2.3 Opdrachtformulering

Er moet een nieuw back-end (de serverapplicatie) ontworpen en geïmplementeerd worden die toegankelijk is vanaf verschillende front-end systemen (interfaces). De front-end kan bijvoorbeeld een website of een administratief programma zijn. Deze back- en front-end systemen moeten – onafhankelijk van de specifieke implementatiedetails – via een protocol kunnen communiceren. De serverapplicatie moet de bedrijfslogica voor het hele proces omvatten: van het aanmaken en invullen van vragenlijsten en extensies tot het verwerken van de resultaten in rapporten. De

serverapplicatie dient goed onderhoudbaar en uitbreidbaar te zijn, zodat in de toekomst nieuwe functionaliteit gemakkelijk toegevoegd kan worden.

Hoofdstuk 3

Domeinanalyse

De nu volgende domeinanalyse begint met een korte beschrijving van het bedrijf en de markt waarin zij werkt. Daarna volgt een beschrijving van hoe in de huidige situatie vragenlijsten en extensies worden gemaakt, ingevuld en verwerkt, en welke rol de projecteigenaren, admin, users en cliënten vervullen.

3.1 Het bedrijf

Onderzoeksbureau Soffos is een Nederlands bedrijf dat werkt binnen het domein *gezondheidszorg* en dat gespecialiseerd is in het meten van het effect van een interventie, door middel van meetinstrumenten. Het bedrijf heeft een aantal lopende en afgeronde projecten, en voert onderhandelingen over nieuwe projecten. Deze projecten variëren in grootte en tijdsduur. Sommige projecten duren niet langer dan een jaar, terwijl voor andere nog geen einddatum is vastgesteld. Eén van de projecten heeft meer dan 250 *users*, waarvan sommigen ruim 150 ingevulde vragenlijsten hebben ingestuurd. Aangezien Soffos via het internet werkt, is het werkgebied van Soffos slechts beperkt door talen en wetten. Soffos werkt momenteel in Nederland, Duitsland en België, en er zijn plannen om uit te breiden naar Zwitserland.

Hoewel de markt voor eenmalige psychologische vragenlijsten groot, actief en groeiende is, is de markt voor effectmetende instrumenten waarin Soffos werkt een niche markt, waarop een beperkt aantal spelers actief is. In de reguliere zorg is *evidence-based medicine* een uitgangspunt, terwijl de complementaire zorg minder als evidence based wordt beschouwd. Veel behandelaars uit de complementaire zorg zien de voordelen of het belang van effectmetingen niet of nauwelijks, en zijn daardoor terughoudend. Echter, nu het besef groeit dat het concept van evidence-based medicine ook een waarde heeft voor de complementaire zorg, gaan meer en meer mensen en organisaties – inclusief zorgverleners en zorgverzekeraars – effectmetingen gebruiken om de kwaliteit van een interventie of methode te bepalen. Dit betekent dat de markt aan het groeien is. Soffos adverteert momenteel niet, omdat het de vereiste capaciteit niet heeft.

Het bedrijf streeft een groei van tenminste 10% per jaar na. Dit probeert Soffos te realiseren door de naamsbekendheid te vergroten, het werkgebied uit te breiden naar andere landen, belanghebbenden te overtuigen van het nut van het concept *evidence-based medicine* in de complementaire zorg en door te investeren in nieuwe zelf-ontwikkelde vragenlijsten.

3.1.1 Medewerkers

Het bedrijf is een familiebedrijf. De taken van de medewerkers zijn divers. Allen houden ze zich bezig met product- en strategieontwikkeling, en verder heeft elk van de medewerkers specifieke kennisgebieden van waaruit ze advies kunnen geven. De rollen en verantwoordelijkheden zijn in beginsel als volgt:

Marij Schüsler apotheker:
Directeur, hoofdonderzoeker, algehele leiding, externe vertegenwoordiging

Fred Schüsler socioloog:
Management bedrijfsvoering, externe vertegenwoordiging

Jenny Schüsler econoom:
Adviseur

Olaf Schüsler informaticus:
Webmaster, automatisering

Taco Schüsler jurist en econoom:
Adviseur

3.2 De activiteiten

Een opdrachtgever sluit een contract met Soffos af om een meetproject te starten. Een *user* pleegt vervolgens een *interventie* om het fysieke of geestelijke welzijn van een cliënt te verbeteren. Een interventie kan divers zijn: van het zetten van een gebroken ledemaat, het voorschrijven van medicijnen, tot het toepassen van acupunctuur en andere vormen van complementaire behandelmethoden. De situatie van een cliënt kan worden besproken aan de hand van specifieke parameters. De relevante parameters worden *dimensies* genoemd, en kunnen gemeten worden met behulp van *vragenlijsten* en *extensies*. Vragenlijsten zijn lijsten van vragen die worden voorgelegd aan de cliënt, en extensies zijn klinische parameters, zoals de hartslag of bloeddruk, die de user opneemt over die cliënt. De kernactiviteiten van Soffos kunnen dan samengevat worden als: het rapporteren over het effect van een interventie door het uitvoeren van pre- en postinterventie metingen van de dimensies door middel van vragenlijsten en extensies. We zullen elk van deze aspecten behandelen.

3.2.1 Opdrachtgever

Een opdrachtgever kan een user zijn, maar is vaak een grotere organisatie. Zo'n organisatie kan zowel commercieel als niet-commercieel zijn, bijvoorbeeld een overheidsinstelling, technologiebedrijven of scholen. Een organisatie die een meetproject start, heeft er baat bij om inzicht in het effect van een interventie te verkrijgen.

Als een opdrachtgever een bedrijf is, dan zijn er twee mogelijkheden: het bedrijf heeft zelf users in dienst, die werknemers van het bedrijf behandelen, of het bedrijf heeft een product of behandelingstraject ontwikkeld waar het bedrijf de effectiviteit van wil meten. In het tweede geval hoeven de users niet bij het bedrijf in dienst te zijn, maar zijn er zelfstandig werkende users. Als een organisatie de users zelf in dienst heeft, dan levert die organisatie aan Soffos een lijst met users aan, en betaalt de organisatie voor het onderzoek. Als users niet bij het bedrijf in dienst zijn, melden users zichzelf aan bij Soffos voor het meetonderzoek en betalen ze er zelf voor.

3.2.2 User

Een user is degene die de interventie pleegt, ofwel de persoon die een cliënt begeleidt. Voorbeelden zijn artsen, psychologen, therapeuten, personeelsmanagers en coaches. Een user kan zelf ook opdrachtgever zijn van een project, bijvoorbeeld als hij de effectiviteit van de eigen behandeling wil weten. In dit geval is de user de opdrachtgever en tegelijkertijd de enige user die participeert in het project.

3.2.3 Cliënt

Een cliënt is een persoon bij wie de interventie plaats vindt. In vrijwel alle gevallen is een cliënt onder behandeling bij een user, maar het komt ook voor dat een cliënt geen user heeft. Zo wordt er in een bepaald project onderzoek gedaan naar de werking van speciale apparatuur, die cliënten kunnen aanschaffen. Het bedrijf dat deze producten verkoopt, vraagt dan rechtstreeks of cliënten mee willen werken aan een meetonderzoek. Deze cliënten hebben dus geen user.

Cliënten die wel een user hebben, worden door hun users aangemeld bij Soffos, waarna Soffos contact opneemt met de cliënt. Een cliënt heeft alleen contact met Soffos voor het invullen van vragenlijsten. Users krijgen de ingevulde vragenlijsten in principe niet te zien, wel de verslagen ervan. Sommige cliënten stellen er prijs op als ze anoniem kunnen blijven, dan wordt er een passende oplossing gezocht.

3.2.4 Vragenlijsten

Vragenlijstenonderzoek is in beginsel een *black-box* methode: zonder de details van een interventie te weten, wordt er gevraagd naar het resultaat. Een of meer vragenlijsten worden aan een cliënt voorgelegd. Sommige vragenlijsten zijn gevalideerd: hiervoor is de vragenlijst afgenomen in combinatie met interviews, en dan is er gekeken of er een duidelijke relatie is tussen de op deze manieren verkregen antwoorden. Vragenlijsten hebben een sleutel, waarmee de resultaten van een vragenlijst worden omgezet naar dimensies. Deze dimensies zijn parameters waarmee het welzijn van een cliënt kan worden uitgedrukt.

Vragenlijstenonderzoek is in beginsel subjectief, want zelfs een simpele invloed als het weer kan ervoor zorgen dat een cliënt een vragenlijst anders invult. De invloed van verstorende factoren wordt verminderd door bij een grote groep cliënten vragenlijsten af te nemen. Er wordt geprobeerd om het bewust beïnvloeden van de vragenlijst te herkennen en tegen te gaan door controlevragen te stellen. Een cliënt wordt door zijn user alleen ingeschreven voor relevante vragenlijsten, zodat bijvoorbeeld een werkloze cliënt geen werk gerelateerde vragenlijsten hoeft in te vullen. Het gebeurt dat een user zelf een vragenlijst invult om ervaring op te doen, van de resultaten krijgt de user dan ook een rapport.

In aanvulling op de informatie uit de vragenlijsten worden er gegevens gevraagd over een cliënt aan de user. Deze lijsten van gegevens worden *extensies* genoemd. In de extensies worden aan de user meetbare, objectieve gegevens gevraagd. Voorbeelden zijn de bloeddruk, de hartslag of het cholesterolgehalte. Welke factoren er in een extensie worden gevraagd hangt af van het project. Extensies worden door Soffos per project samengesteld en bevatten meestal drie tot zeven klinische parameters. In sommige extensies wordt ook gevraagd naar aspecten van de interventie zelf, waardoor de interventie niet langer als een black-box wordt beschouwd. De in een extensie gevraagde gegevens worden altijd door de user ingevuld, maar hoeven niet altijd door de user te zijn gemeten. Soms is er sprake van een apparaat dat dit doet, of een specifieke test, uitgevoerd in een extern laboratorium. Het kan gebeuren dat er voor een cliënt alleen extensie parameters worden vastgesteld, en geen vragenlijsten worden ingevuld.

Aanvankelijk werden de vragenlijsten op papier per post bij de cliënten thuis gestuurd, met een gefrankeerde retourenvelop. Later werden de vragenlijsten afgenomen via een speciaal daarvoor ontworpen website. Doordat Soffos geen vertrouwen meer heeft in de website, gebeurt het afnemen van vragenlijsten nu tijdelijk per e-mail.

3.2.5 Rapporten

Van de antwoorden op vragenlijsten worden rapporten gemaakt. Er zijn vier soorten rapporten, namelijk $t = 0$ rapporten, verschilrapporten, groepsrapporten en statistiekrapporten. De $t = 0$ rapporten en verschilrapporten tonen de resultaten op de gemeten dimensies voor een met naam genoemd individu, en in het geval van een verschilrapport gebeurt dit voor twee metingen. Groepsrapporten laten de minimale, gemiddelde en maximale scores zien op de gemeten dimensies voor een groep van cliënten. De statistiekrapporten laten over een groep cliënten bepaalde statistische resultaten zien die niet in de andere rapporten voor komen, zoals overkoepelende resultaten die niet gerelateerd hoeven te zijn aan een enkel project. Een statistiekrapport is een mogelijke bron voor bijvoorbeeld het schrijven van een wetenschappelijk artikel.

De rapporten bevatten persoonlijke informatie. Basisregel daarbij is dat een partij alleen relevante informatie te zien krijgt, en informatie die tot een specifieke cliënt te herleiden is, is alleen beschikbaar na expliciete toestemming van die cliënt. Voor groepen kleiner dan zes personen wordt daarom in beginsel nooit een groepsrapport gemaakt, omdat dan de privacy in het geding zou kunnen komen. Ook worden de $t = 0$ en verschilrapporten alleen aan de user van een cliënt gestuurd. Een opdrachtgever heeft meestal geen directe relatie met de cliënt (behalve als de opdrachtgever ook de user van de cliënt is) en krijgt daarom ook de individuele resultaten van de cliënt niet te zien.

3.2.6 Pre- en post-interventie metingen

In de meeste projecten is er sprake van een pre- en een post-interventie meting, respectievelijk $t = 0$ en $t = 1$ genaamd. De pre-interventie meting is de nulmeting. Soms zijn er meer dan twee meetmomenten in een project. Het gebeurt ook soms dat er slechts een retrospectieve post-interventie meting plaats vindt.

3.2.7 Privacy

Het beschermen van privacy is een hoofdzakelijke voorwaarde in dit veld, omdat er met medische gegevens van cliënten wordt omgegaan. Cliënten willen niet dat hun persoonlijke ingevulde antwoorden aan derden worden overgebracht. Soffos is aangemeld bij het CBP, en moet zich aan de regels van het CBP houden. De *Wet bescherming persoonsgegevens* stelt geen beperkingen aan de opslagtermijn van gegevens, maar gegevens moeten wel worden vernietigd als de verstrekker van de gegevens hierom verzoekt, of als er vooraf een termijn is afgesproken. Ook hoeven gegevens volgens de Wet bescherming persoonsgegevens niet te worden geanonimiseerd, dit kan soms wel nodig zijn.

3.3 Eigenschappen van vragenlijsten en rapporten

In dit hoofdstuk zetten we de huidige eigenschappen van vragenlijsten en rapporten op een rij. Verschillende antwoordtypen die vragen kunnen hebben, worden besproken. Ook worden er enkele eigenschappen van rapporten besproken.

3.3.1 Vragenlijsten

Meestal zijn er twee invulmomenten voor de vragenlijsten van een cliënt, maar afhankelijk va het project kunnen dat er ook meer of minder zijn. De invulmomenten kunnen per cliënt verschillen, en zijn afhankelijk van de interventie en de individuele duur daarvan.

De vragenlijsten bevatten lopende tekst en uitleg, en meestal tussen de 10 en de 50 vragen. Elke vraag kan één van de volgende soorten antwoorden hebben. Dit kan binnen een vragenlijst verschillen.

Open
Elk antwoord kan worden ingevuld.

Cijfer
Er moet een cijfer gegeven worden binnen een bepaald bereik, bijvoorbeeld 1 tot 10 of 0 tot 100.

Numeriek
Er moet een numerieke waarde gegeven worden.

Likert schaal
Er moet een antwoord op een schaal (bijvoorbeeld van 1 tot 5) worden gekozen.

Booleaans
Kies een antwoord uit twee tegengestelde mogelijkheden. Voorbeelden zijn: er moet ja/nee, waar/onwaar, van toepassing/niet van toepassing worden ingevuld.

Eén van meerdere mogelijkheden
Er moet één antwoord uit meerdere van elkaar gescheiden mogelijkheden worden gekozen.

Meerdere van meerdere mogelijkheden
Er moeten nul, één of meer antwoorden uit een lijst mogelijkheden worden gekozen. De mogelijkheden kunnen elkaar overlappen.

3.3.2 Rapporten

De rapporten zijn specifiek voor een bepaalde vragenlijst. Behalve de statistiekrapporten bevatten rapporten slechts eenvoudige gegevens van een of meer cliënten, weergegeven als percentages in een tabel of diagram. Verder bevatten de rapporten een vaste tekst, die onafhankelijk is van de gemeten waarden van de dimensies, en enkele gegevens over de cliënt(en) waar de resultaten betrekking op hebben, zoals geslacht, opleiding, burgerlijke staat en eventueel naam (in het geval van een $t = 0$ of verschilrapport).

De berekeningen die in een rapport worden gebruikt zijn:

- Minimum
- Maximum
- Gemiddelde
- Som
- Score berekening: het gemiddelde van de score op meerdere onderling samenhangende vragen, geschaald naar een waarde tussen 0 en 100%.

Deel II

Proces

Hoofdstuk 4

Methodiek

Dit hoofdstuk zal de methodiek beschrijven die tijdens het gehele project is aangehouden.

4.1 Agile

Tijdens de implementatiefase van het project hebben we voor een methodiek gekozen die lijkt op de Agile methode. Agile kan gebruikt worden als raamwerk voor het ontwikkelen van software. Hierbij stelt een week een iteratie voor, wat betekent dat elke week een aantal features worden geïmplementeerd. Deze features worden geselecteerd uit een geprioriteerde lijst van requirement welke tijdens de oriëntatiefase tot stand zijn gekomen na uitgebreid overleg met Soffos. De requirements die gebruikt zijn bij het ontwikkelen van de serverapplicatie zijn te vinden in bijlage B.

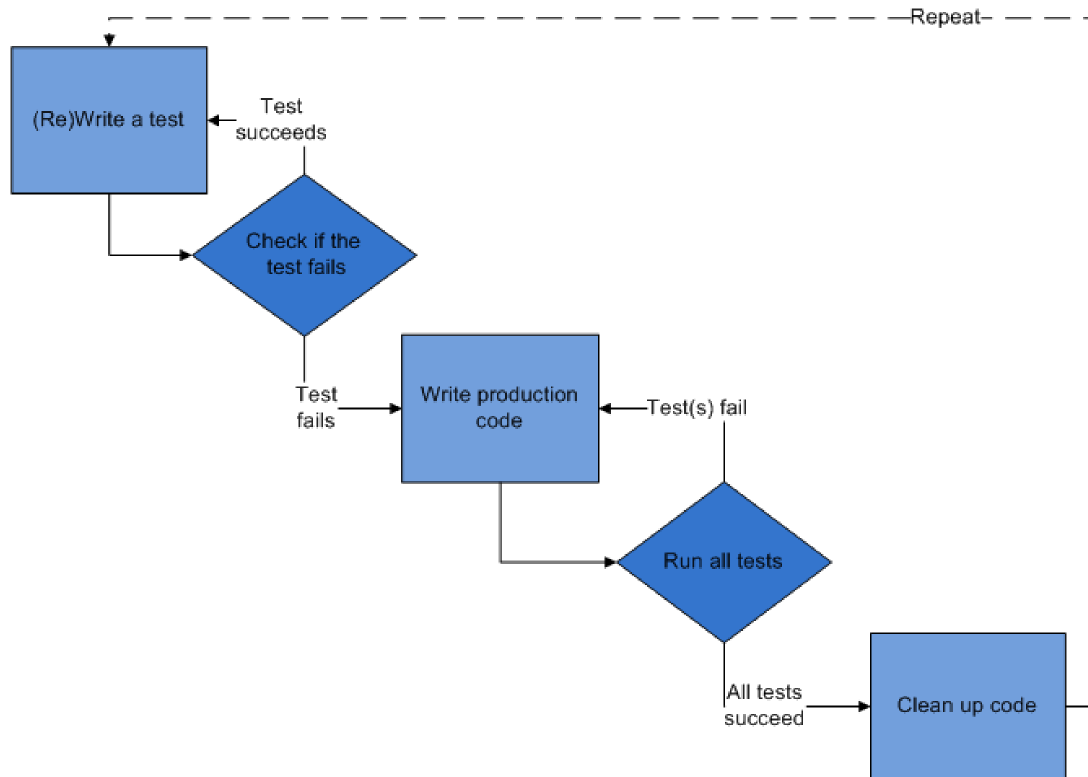
Aan het begin van elke dag maakt iedereen in het projectteam een SVN update van de code en de documentatie. Features worden vervolgens verdeeld over de groepsleden en per onderdeel wordt een design gemaakt. De groepsleden volgen het principe van *test-driven development* tijdens het implementeren en alleen volledig geteste features mogen worden geïntegreerd in de build en worden gecommit naar SVN. Hierdoor bevat de SVN nooit code die niet getest of niet werkend bevonden is. Het doel was om aan het einde van elke week de geselecteerde features te hebben geïmplementeerd. De niet (volledig) geïmplementeerde features worden doorgeschoven naar de planning van de week erop.

4.2 Test Driven Development

Tijdens het implementeren van een feature werd er gewerkt volgens het idee van Test-Driven Development. Deze methode bestaat uit een aantal stappen. Allereest werd het raamwerk voor de functie aangemaakt en goed gedocumenteerd. Vervolgens werd hiervoor een testsuite opgesteld om *statement coverage* te behalen. Statement coverage wilt zeggen dat de testsuite tests bevat die elke statement in de code test, bijvoorbeeld voor elk `if` statement wordt zowel de code als de conditie `True` is, en de code als de conditie `False` is getest.

Wanneer alleen deze testsuite ontwikkeld is moet deze falen wanneer hij uitgevoerd wordt. Als dit niet het geval is, is de testsuite niet compleet en moet deze herzien worden. Wanneer alle tests niet slagen kan de methode geïmplementeerd worden volgens de specificaties van de methode. Na de implementatie wordt er nagegaan of alle tests geslaagd zijn. Als er een test niet slaagt zal de implementatie moeten worden aangepast zodat de test wel slaagt. Vervolgens

wordt er opnieuw een SVN update gedaan om de nieuwste code van SVN te integreren met de lokaal gewijzigde code van de applicatie. Wanneer vervolgens alle tests die aanwezig zijn in de serverapplicatie succesvol zijn, kan de nieuwe feature gecommit worden naar SVN. Door deze werkwijze strikt aan te houden is er altijd een werkende serverapplicatie aanwezig op de SVN, die voldoet aan de requirements.



Figuur 4.1: Implementatiecyclus van een feature volgens Test-Driven Development.

Er zitten echter een aantal nadelen aan Test-Driven Development. Wanneer alle tests succesvol zijn, betekent dit niet automatisch dat de volledige applicatie correct functioneert. Verder geldt dat aangezien zowel de test als de implementatie door dezelfde persoon geschreven zijn, het voor kan komen dat bepaalde zaken over het hoofd gezien worden.

Hoofdstuk 5

Planning

In dit hoofdstuk zullen wij de planning en de uitvoering van het project bespreken. Allereerst zal er een kort overzicht gegeven worden van onze planning zoals die was opgesteld in de beginfase van het project. Vervolgens zal worden beschreven hoe het project uiteindelijk uitgevoerd is en zal er een vergelijking gemaakt worden tussen de planning vooraf en de werkelijke uitvoering.

5.1 Planning vooraf

Vanwege de gebruikte Agile-methodiek was er in de oriëntatiefase slechts een globale planning gemaakt met een indeling in fasen en een schatting van de duur van elke fase.

Van te voren werd er met de volgende planning rekening gehouden:

Oriëntatiefase (week 1-2)
De oriëntatiefase zou bestaan uit het schrijven van het *Requirements Analysis Document*, welke de requirements van de serverapplicatie beschrijft die in samenwerking met Soffos tot stand zijn gekomen. Ook zou een *Plan van Aanpak* gemaakt worden waarin een duidelijk overzicht gegeven wordt van de aanpak. Verder zouden we in het *Oriëntatieverslag* verschillende technische hulpmiddelen afwegen die ons konden helpen bij het ontwikkelen van de serverapplicatie. Een globale opzet van het ontwerp van de applicatie en de interactie en afhankelijkheden tussen de modules zouden we beschrijven in het *Object Design Document*.

Implementatiefase (week 3-9)
In deze fase zou er wekelijks gekozen worden uit een aantal features en requirements die vervolgens zouden worden ontworpen, geïmplementeerd, getest en gedocumenteerd. We zouden gaan werken met korte iteraties van slechts een week. Aan het begin van elke week zou er een planning gemaakt worden voor die week. Vervolgens zou er aan het eind van elke werkweek een werkend prototype van de serverapplicatie zijn, die gebruikt kan worden voor het testen van de functionaliteiten en het demonstreren van de voorgang aan Soffos.

Acceptatietests (week 9-10)
We zouden twee weken werken aan de acceptatietests, waarbij er gekeken zou worden of alle componenten onderling goed samenwerken en of het systeem aan de requirements voldoet.

Afrondingsfase (week 10-11)
In deze laatste fase zou het eindverslag geschreven worden. Ook zou de eindpresentatie gemaakt en gepresenteerd worden.

Een weergave van de hierboven beschreven planning is uitgewerkt in figuur 5.1.

Taak	Begin	Eind	Duur	Week
Oritiëntatiefase	Week 1	Week 2	2 weken	1–2
Implementatiefase	Week 3	Week 9	7 weken	3–9
Acceptatietests	Week 9	Week 10	1 week	9–10
Afrondingsfase	Week 10	Week 11	1 week	10–11

Figuur 5.1: Gantt-chart van de planning zoals deze vooraf was vastgesteld.

5.2 Werkelijke uitvoering

Zoals in veel projecten liep de uiteindelijke uitvoering van het project niet helemaal gelijk met de bedoelde planning. De werkelijke uitvoering van de verschillende fasen is weergegeven in figuur 5.2. De oriëntatiefase heeft twee weken geduurd met een kleine uitloop in week 3. De implementatiefase is volgens planning gestart in week 3. Het was aan het begin van het project niet voorzien dat we ook een front-end testwebsite zouden moeten schrijven, en daardoor is de implementatiefase een week uitgelopen tot week 10. Aan het eind van elke week stond er zoals bedoeld een werkend prototype online op de website. De acceptatie- en gebruikerstests hebben in week 11 plaatsgevonden, tegelijk met de afrondingsfase die volgens planning van start is gegaan in week 10. De presentatie zal worden gehouden in week 13, dus de uitloop van de implementatiefase kon worden opgevangen in de laatste paar weken.

Taak	Begin	Eind	Duur	Week
Oritiëntatiefase	Week 1	Week 3	3 weken	1–3
Implementatiefase	Week 3	Week 10	8 weken	3–10
Acceptatietests	Week 11	Week 11	1 week	11
Afrondingsfase	Week 10	Week 13	4 week	10–13

Figuur 5.2: Gantt-chart van de planning zoals deze achteraf was.

Omdat er gekozen was voor de Agile-methodiek werd er aan het begin van elke week tijdens de implementatiefase een planning gemaakt voor die week waarin de requirements werden geselecteerd die geïmplementeerd zouden worden. Hieronder worden kort de werkzaamheden opgesomd die in elke week plaats hebben gevonden. De implementatiedetails van de verschillende modules zijn te vinden in hoofdstuk 8 van dit verslag.

Week 18, 2011

- authenticatiemodule
- ontwerpen van de database

- opzetten van de front-to-back communicatie

Week 19, 2011

- autenticatiemodule
- vragenlijst- en extensiemodule
- opzetten van de front-to-back communicatie

Week 20, 2011

- vragenlijst- en extensiemodule
- start met front-end testwebsite

Week 21, 2011

- vragenlijst- en extensiemodule
- gijzelaarsmodule
- front-end website bijwerken met nieuwe pagina's

Week 22, 2011

- vragenlijst- en extensiemodule
- front-end website bijwerken met nieuwe pagina's

Week 23, 2011

- cliëntgroepenmodule
- vragenlijst- en extensiemodule
- projectenmodule
- front-end website bijwerken met nieuwe pagina's

Week 24, 2011

- talenmodule
- e-mailmodule
- projectenmodule
- front-end website bijwerken met nieuwe pagina's

Week 25, 2011

- multi-language systeem
- vernieuwde authenticatie module (vanwege nieuwe requirements)
- front-end website bijwerken met nieuwe pagina's

Hoofdstuk 6

Problemen en uitdagingen

Tijdens het bacheloreindproject verliep niet alles vlekkeloos: er waren wat obstakels bij het werken met de server, en het werken in opdracht van een bedrijf is ook een uitdaging. In dit hoofdstuk zullen we deze uitdagingen en problemen bespreken met daarbij ook onze oplossing.

6.1 Server

De serverapplicatie zal draaien bij een externe host die door Soffos is aangesteld. Omdat wij op de server maar beperkte uitvoerrechten hadden, was het niet mogelijk om bepaalde hulpprogramma's en libraries van anderen te installeren. Deze libraries waren nodig om de serverapplicatie te laten werken, dus zochten we naar een geschikte oplossing. De libraries zijn uiteindelijk toegevoegd in een aparte `lib`-map bij de code zonder ze te installeren, waardoor de serverapplicatie toch goed kon werken.

6.2 Werken voor een bedrijf

Het bacheloreindproject was voor ons de eerste keer dat we zelfstandig een grote opdracht uitvoerden voor een bedrijf. Hierbij kwamen een aantal uitdagingen om de hoek die we vooraf niet volledig hadden verwacht. Het was bijvoorbeeld lastig om de requirements vanuit het bedrijf goed te verwoorden: het bleek erg moeilijk te zijn om Soffos precies de juiste vragen te stellen, zodat de requirements uiteindelijk eenduidig verwoord worden. Bij de vergaderingen tijdens het project kwamen vaak weer nieuwe requirements of wijzigingen in requirements naar voren. Het was lastig om de serverapplicatie te implementeren terwijl de requirements regelmatig veranderden. Achteraf was het handig geweest om eerder na de interviews met Soffos een bevriezing van requirements voor te stellen. Vanaf dat moment mogen de requirements dan niet meer wijzigen.

Een andere uitdaging in het werken voor een bedrijf was de lengte van vergaderingen. In elke vergadering viel er veel te bespreken, de vergaderingen duurden daardoor dan vaak een heel aantal uren. Achteraf gezien was het wellicht handiger geweest om door de agendapunten goed te bepalen en een deel daarvan als 'huiswerk' te doen kortere vergaderingen te kunnen houden met het bedrijf. Zeker in weken die door feestdagen slechts drie of view werkdagen hadden, bleef er minder tijd over om aan het project te werken.

6.3 Agile

De Agile-methodiek was voor iedereen uit de projectgroep nieuw en was daarom een goede uitdaging voor het bachelorproject. Door wekelijks een lijst van features te documenteren en implementeren groeide de serverapplicatie langzaam tot een volledig systeem. Tijdens dit proces zijn we weinig problemen tegen gekomen. Voor niet alle features is een perfect design van te voren gemaakt, maar dat heeft ons geen problemen opgeleverd.

Aan het begin van de implementatiefase hebben we een schatting gemaakt van modules en features die af zouden zijn aan het eind. Vanwege Agile was dit moeilijk te voorspellen, maar achteraf is het prototype redelijk goed volgens de schatting gemaakt. Verder kregen we de indruk dat Soffos het waardeerde dat de voortgang per week goed te volgen was, en daardoor kregen ze ook meer inzicht in ons werktempo. De Agile-methodiek hebben we dus als positief ervaren en zullen we zeker in een volgend project weer toepassen.

Hoofdstuk 7

Hulpmiddelen

Tijdens ons project hebben we een aantal hulpmiddelen gebruikt. Zonder deze tools was het project ook wel uitvoerbaar, maar het gebruik van de middelen heeft het verloop van het project een stuk gemakkelijker gemaakt.

7.1 Python

Een keuze voor programmeertaal voor de serverapplicatie hadden we niet, de server waarop de serverapplicatie moest draaien ondersteunde alleen Python. Niemand uit de projectgroep had eerder met Python gewerkt. Python bleek een programmeertaal te zijn die erg makkelijk te leren en gebruiken is. De object-geörienteerde principes die we in Java hebben geleerd, bleken ook van toepassing op Python. Toch zijn er duidelijke verschillen merkbaar: binnen Python speelt de inspringing van een regel code een grote rol, terwijl het type van een variabele juist in Java veel belangrijker is.

7.2 Eclipse

Eclipse is een software-ontwikkelomgeving die tijdens het project gebruikt wordt voor de ontwikkeling van de serverapplicatie. Pydev is een handige Python plugin die beschikbaar was voor Eclipse, en daardoor was het makkelijk om testsuites te bouwen en fouten in de applicatie op te sporen.

7.3 ORM

Door het gebruik van een ORM werd de communicatie tussen de serverapplicatie en de database vereenvoudigd. Een ORM is een *object relational mapper*, die tabellen uit de database relateert aan objecten in onze applicatie. In plaats van overal zelf query's te maken met behulp van een *expressie boom*, konden we simpele query's maken op objecten, die werden uitgevoerd door de ORM. Door de objecten binnen de applicatie te definiëren, was het eenvoudig om een test database op te zetten voor de test suite zonder extra moeilijkheden. De ORM die wij gebruikt hebben is *SQLAlchemy*. Deze ORM werkte nauw samen met de server die we gebruikte en was dus een voor de hand liggende keuze.

7.4 SVN

Subversion (SVN) is een versiebeheersysteem dat wij gebruikt hebben om eenvoudig features toe te kunnen voegen aan een nieuwe versie van de applicatie en oude versies terug te kunnen halen, indien nodig. Tevens zorgde Subversion ervoor dat conflicten in de code (als meerdere teamleden aan hetzelfde bestand hadden gewerkt) opgelost konden worden en dat het gehele project voor alle betrokkenen makkelijk toegankelijk was.

Deel III

Ontwerp en implementatie

Hoofdstuk 8

Ontwerp

In dit hoofdstuk wordt het ontwerp van de serverapplicatie uiteen gezet. Het is gebaseerd op de eisen zoals die gesteld zijn door Soffos (bijlage B) en de domeinanalyse van hoofdstuk 3.

8.1 Model

Het model van de serverapplicatie omvat twee hoofdmodules en nog een aantal kleinere modules. Eén hoofdmodule voor het beheer van de *authenticatie en rollen* van de actor, en een tweede voor het beheer van *vragenlijsten, extensies, vragen en antwoorden* die gerelateerd zijn via een *project*. Verder zijn er nog de relatief zelfstandige modules voor het beheren van *cliëntgroepen*, het toevoegen van *beschikbare talen* en het versturen van *e-mails*.

8.1.1 Autenticatie en rollen

Er zijn drie groepen actoren die van de applicatie gebruik zullen maken:

- de administrator, die onder andere projecten, vragenlijsten en extensies kan toevoegen, bewerken en verwijderen;
- de user, die vragenlijsten voor hun cliënten kan openstellen en de rapporten hiervan ontvangt, en die extensies van een cliënt kan invullen;
- de cliënt, wiens voornaamste taak het invullen van vragenlijsten is.

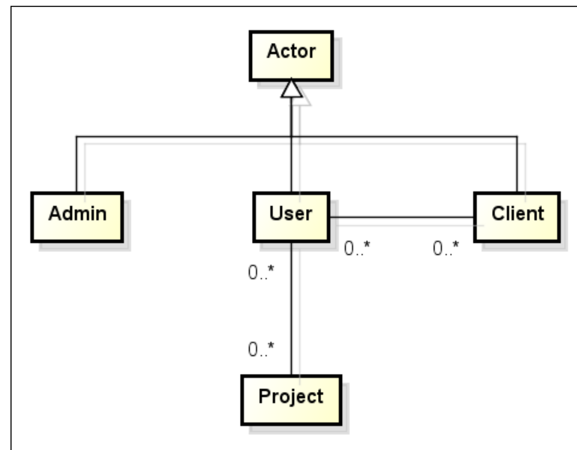
Elk van deze groepen actoren heeft dus een rol binnen de applicatie, en deze drie rollen (Admin, User en Client) zijn de enige rollen die er in het model worden onderscheiden. Een actor kan meerdere rollen hebben. Uit de domeinanalyse bleek dat users vaak ook vragenlijsten voor zichzelf invullen, om daar bijvoorbeeld zelf een rapport van te ontvangen. In dat geval heeft de user actor ook de rol van cliënt.

Cliënten die bij een user in behandeling zijn horen ook in het model bij die user, zodat de user vragenlijsten aan een cliënt kan toewijzen. Cliënten kunnen bij meerdere users in behandeling zijn, dus er bestaat een many-to-many relatie tussen de cliënten en de users. Users zelf participeren in nul, één of meer (onderzoeks)projecten, dus er bestaat ook een many-to-many relatie tussen de users en de projecten. Zie figuur 8.1.1.

Volgens de requirements die Soffos en het *College Bescherming Persoonsgegevens* aan de serverapplicatie stellen moet deze goed beveiligd moet worden zodat ongeautoriseerde toegang

tot de gegevens of de applicatie niet mogelijk is. Er zal een inlogsysteem moeten komen, waarbij een actor zijn of haar gebruikersnaam en wachtwoord moet invullen om toegang te krijgen tot de applicatie.

Elke actor vermeldt verplicht een gebruikersnaam, wachtwoord en een *achternaam*, en verder optioneel het *e-mail adres*, *voornamen* en *tussenvoegsel*. Elke user heeft nog specifieke betalingsgegevens naast de algemene actorgegevens. Cliënten hebben een aantal demografische gegevens die samen met elke ingevulde vragenlijst en extensie moeten worden bewaard, zodat is af te leiden wat ten tijde van het invullen bijvoorbeeld de burgerlijke staat of de woonplaats van de cliënt was.

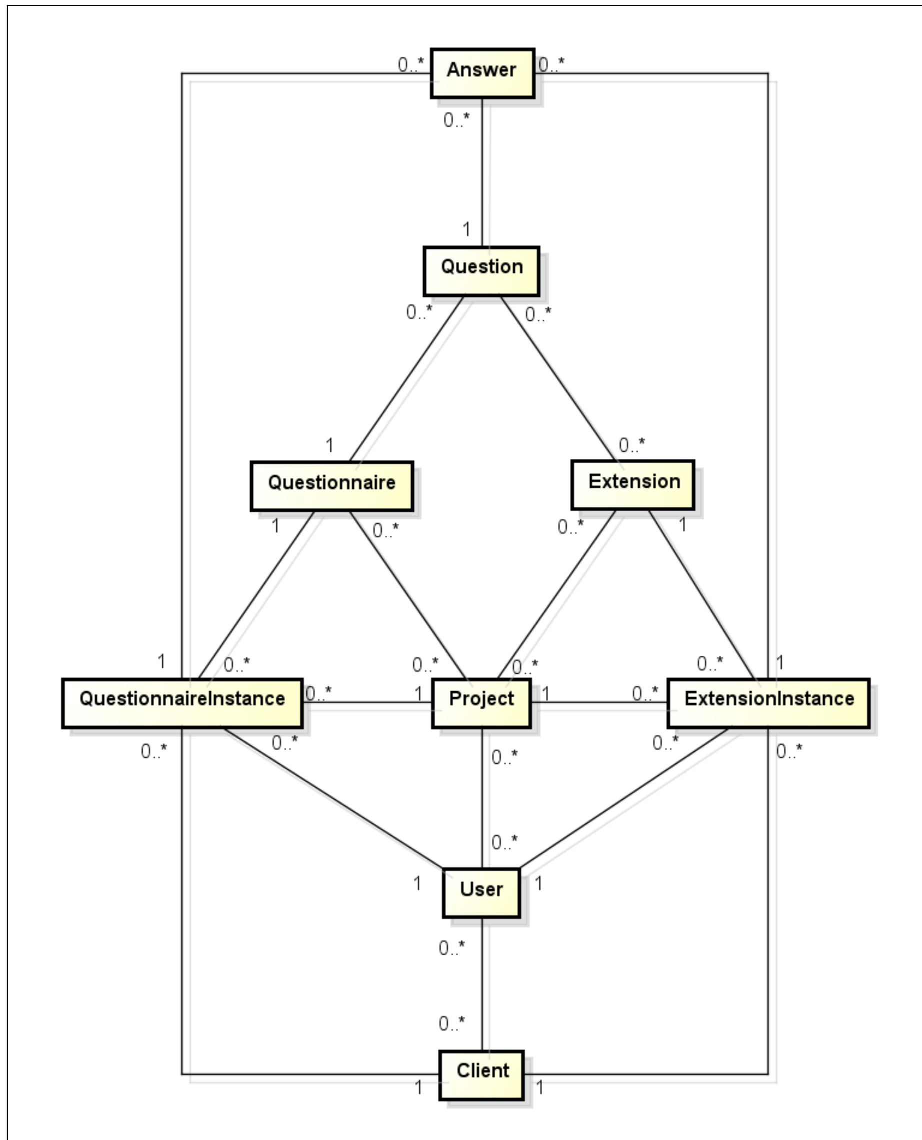


Figuur 8.1: De Admin, User en Client rollen bij de Actor.

8.1.2 Meetinstrumenten, vragen en antwoorden

Er zijn twee soorten meetinstrumenten: vragenlijsten en extensies. Vragenlijsten worden voorgelegd aan cliënten, en extensies aan de user van elke cliënt. Vragenlijsten en extensies kunnen onderdeel zijn van nul of meer projecten. Er bestaat daarom een many-to-many relatie tussen projecten en meetinstrumenten. Deze meetinstrumenten verschillen alleen per project, niet per user, cliënt of tijdstip.

Zowel vragenlijsten als extensies bevatten een aantal vragen. Elke vraag heeft een tekst (de eigenlijke vraag) en een soort antwoord (zie domeinanalyse in hoofdstuk 3). Hoewel een one-to-many relatie tussen vragen en vragenlijsten voor de hand ligt, is er de requirement dat extensies uit een collectie bestaande geformuleerde vragen moeten kunnen worden samengesteld. Een vraag kan dus onderdeel zijn van nul of meer extensies, en dus moet er een many-to-many relatie zijn tussen extensievragen en extensies.



Figuur 8.2: De vragenlijsten en extensies en hun instanties.

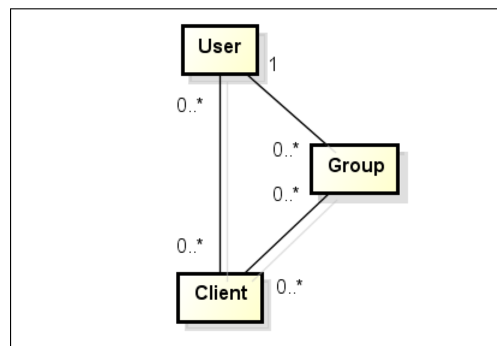
Een user kan een vragenlijst voor een cliënt openstellen door een tijdslot te bepalen waarin de cliënt de vragenlijst mag invullen. Dezelfde vragenlijst kan meerdere keren worden opengesteld voor dezelfde cliënt in verschillende tijdslots zodat bijvoorbeeld een een pre- en een postinterventiemeting kan worden gedaan. De bij de vragenlijst horende extensie is dan ook beschikbaar voor de user zelf om in te vullen. Een beschikbaar gestelde vragenlijst of extensie noemen we een *instantie* van een vragenlijst of extensie. Deze *instanties* bevatten specifieke informatie, zoals de cliënt waarvoor de vragenlijst wordt opengesteld, welke vragenlijst wordt opengesteld en het tijdslot. Alleen als een vragenlijst door een cliënt, of een extensie door een user is ingevuld worden de gegeven antwoorden opgeslagen bij de betreffende vragenlijst- of extensieinstantie. Instanties hebben een many-to-many relatie met de project-user-cliënt-meetinstrument combi-

natie, en gaan dus over een bepaalde meetinstrument voor een bepaalde cliënt met een bepaalde user in een bepaald project. Deze relaties worden weergegeven in figuur 8.1.2. Zoals uit deze figuur blijkt, is er weinig verschil tussen vragenlijsten en extensies: bij een extensie kan de vraag meerdere keren worden gebruikt in verschillende extensies, maar bij een vragenlijst kan dit niet. We hebben daarom besloten om vragenlijsten en extensies samen te behandelen, en dit heeft tot gevolg dat het mogelijk is – in het model – om een vraag in meerdere vragenlijsten te gebruiken.

8.1.3 Cliëntgroepen

In de requirements staat dat er een mogelijkheid moet zijn voor de user om cliënten in groepen in te delen. De cliënten in een groep moeten allemaal in behandeling zijn bij de user die de groep aanmaakt, en de user kan zelf betekenis geven aan een groep. Zo kunnen bijvoorbeeld rokers, zwangere vrouwen, of cliënten die later met de behandeling zijn gestart in een groep worden ingedeeld.

Een user kan Soffos vragen om een groepsrapport over een bepaalde groep cliënten, en de user kan deze groepen zelf indelen. Een cliënt kan in meerdere groepen zitten – zelfs bij dezelfde user – en dus is er een many-to-many relatie tussen de cliëntgroepen en de cliënten. Elke groep wordt beheerd door exact één user, dus er is een one-to-many relatie tussen een user en de groepen die hij of zij beheert. De relaties tussen groepen, users en cliënten worden samengevat in figuur 8.1.3.

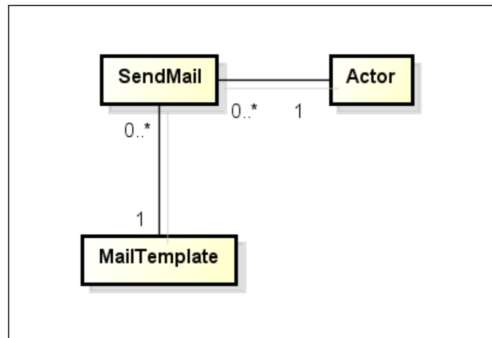


Figuur 8.3: De groepen waarin cliënten kunnen zitten.

8.1.4 E-mail

Volgens de requirements moet een nieuw geregistreerde actor (user of cliënt) een gebruikersnaam en wachtwoord ontvangen per e-mail. Ook moet er uiteindelijk een notificatiesysteem komen waarmee een cliënt een e-mail krijgt als er bijvoorbeeld een nieuwe vragenlijst beschikbaar is. Uit het feit dat e-mails met de door cliënten gegeven antwoorden van het oude systeem niet altijd arriveerden bij Soffos ontstond de requirement dat bijgehouden moest worden welke e-mails er verstuurd zijn, en dat deze te reproduceren moeten zijn.

Van de verzonden e-mails moeten de gegevens uit alle velden worden opgeslagen om de e-mail exact te kunnen reproduceren om deze opnieuw te sturen. Er is een one-to-many relatie tussen de actor en de mail die naar het e-mail adres van die actor is verzonden, zoals weergegeven in figuur 8.1.4.

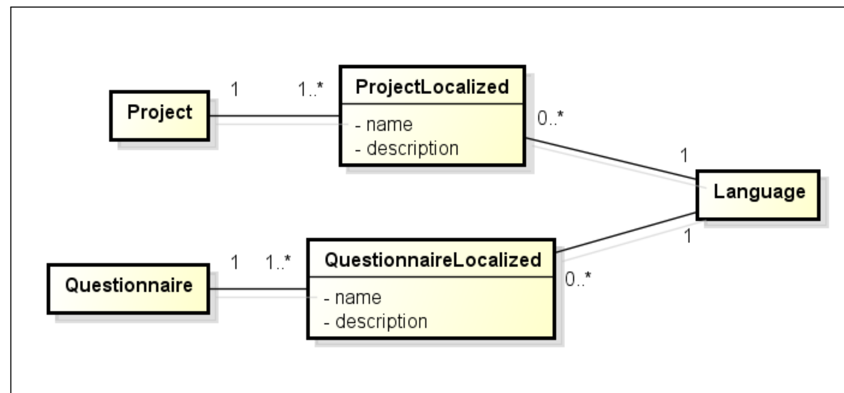


Figuur 8.4: De mails en hun relatie met de actoren.

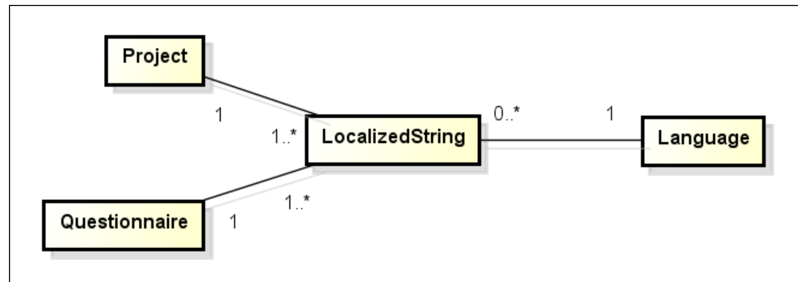
8.1.5 Meertalen-ondersteuning

Soffos is behalve in Nederland ook actief in Duitsland en België, en wil uiteindelijk verder kunnen uitbreiden. Hierdoor ontstond de requirement dat het systeem meerdere talen moet kunnen ondersteunen voor alle informatie die getoond kan worden aan een user of cliënt.

De naam van een project, de naam van een meetinstrument, de vraagtekst en de e-mails moeten in meerdere talen beschikbaar komen. Er zijn twee mogelijkheden om dit te doen: elk stuk tekst een uniek nummer geven, en daarnaar verwijzen vanuit het meetinstrument, de vraag, het project of de mail (figuur 8.1.5); of elk object een tegenhanger geven met de vertaalde teksten (figuur 8.1.5). We hebben gekozen voor de laatste oplossing vanwege twee redenen: omdat de te doorzoeken lijst dan nooit langer is dan het aantal objecten waarvan de vertaling kan worden gezocht maal het aantal talen; en omdat er niet meerdere zoekacties nodig zijn voor een object met meerdere te vertalen velden (bijvoorbeeld *titel* en *omschrijving* van een project).



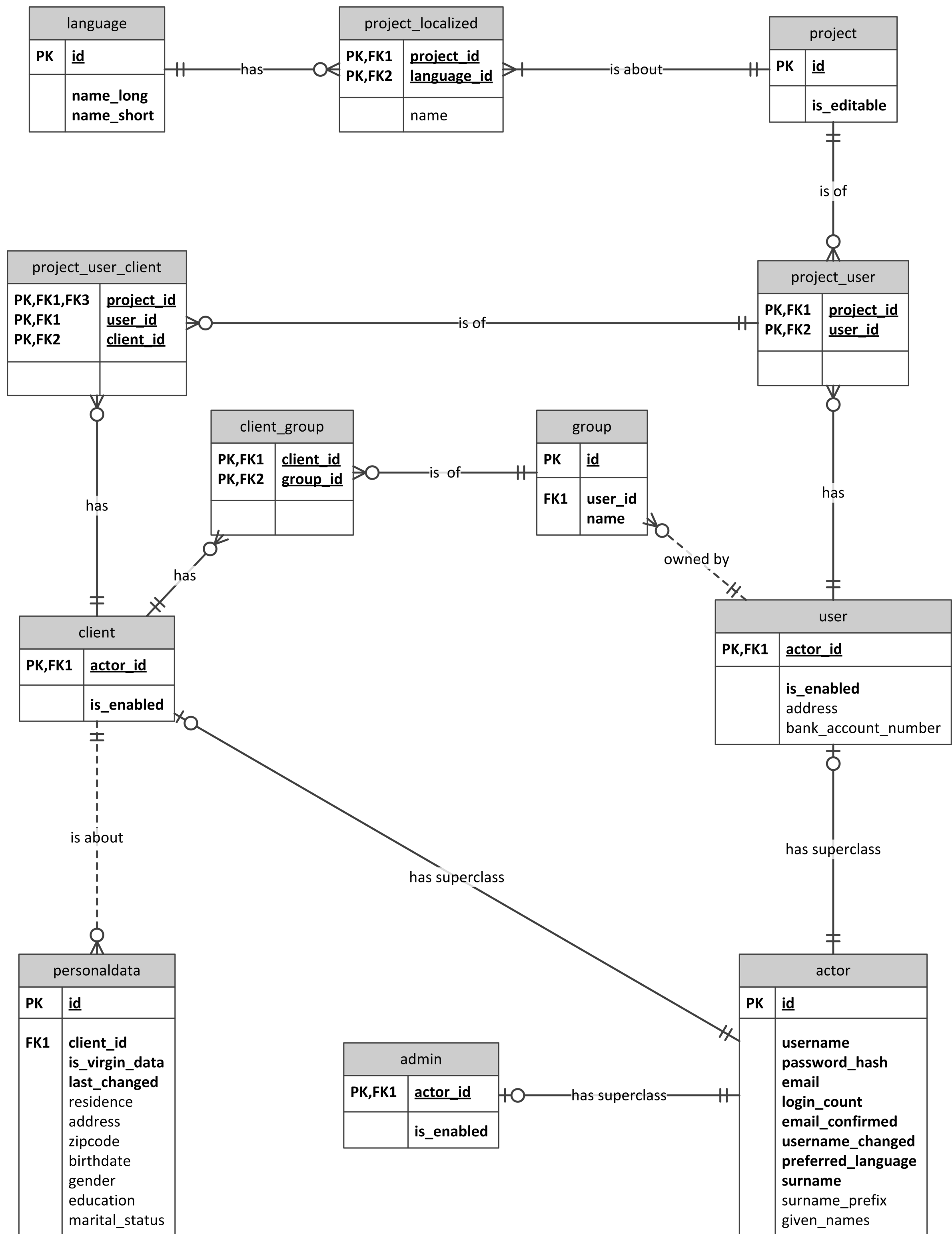
Figuur 8.5: De gekozen oplossing voor het modelleren van teksten in meerdere talen.



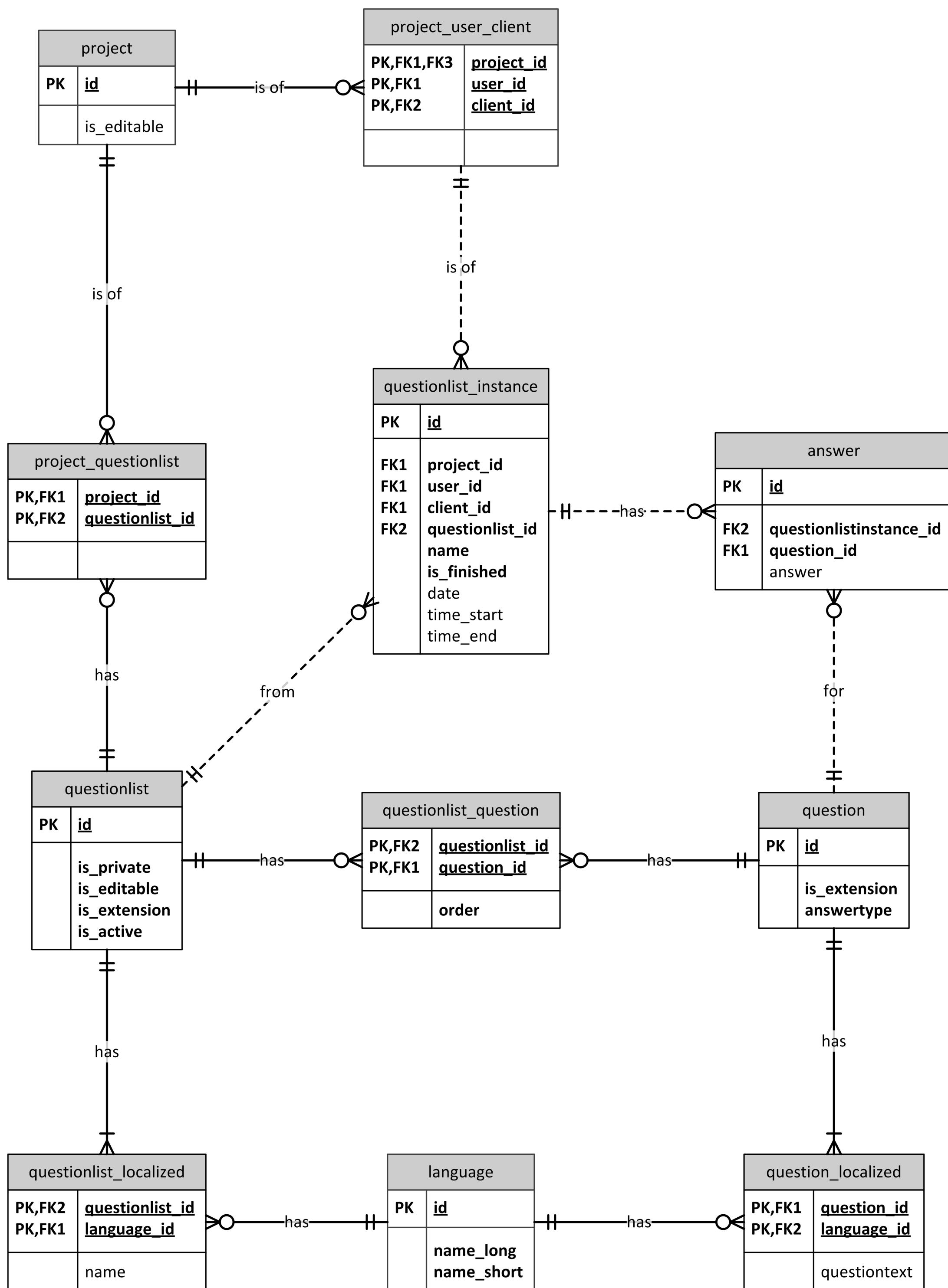
Figuur 8.6: De alternatieve oplossing voor het modelleren van teksten in meerdere talen.

8.2 Databaseontwerp

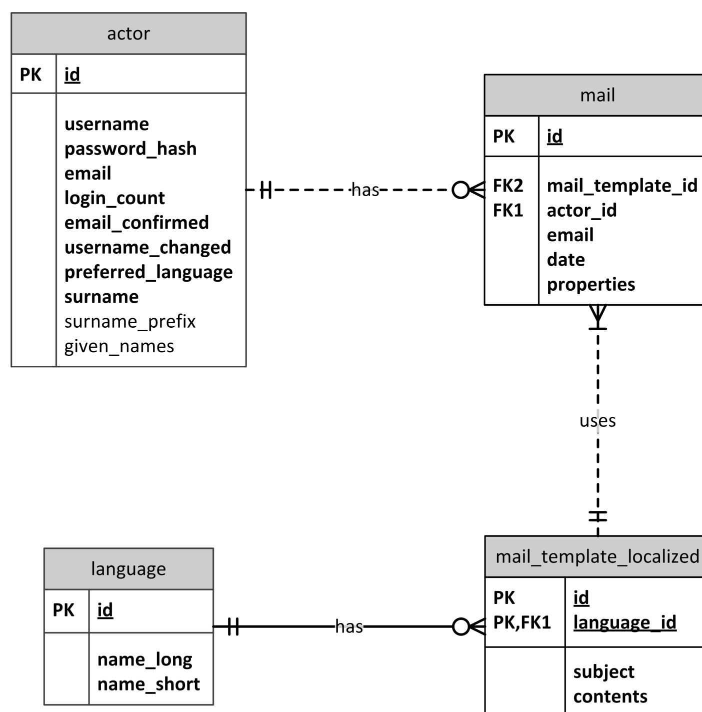
Het hiervoor beschreven model moet in een database kunnen worden opgeslagen. Het model dicteert veel van de benodigde tabellen, en verdere tabellen volgen logisch uit het model. Zo is er een cross-table nodig om de many-to-many relaties weer te geven, en komt er een `_localized` tabel die correspondeert met de objecten waarvoor de tekst vertaalt moet kunnen worden, volgens de ontwerpkeuze die in de vorige sectie is behandeld. Figuur 8.2 toont hoe de admin, user en cliënt rollen samenhangen met de actor informatie, en wat hun relatie tot een project is in het databaseontwerp. In figuur 8.2 staan de relaties tussen vragenlijsten en extensies, hun instanties en hun antwoorden. En tenslotte staat in figuur 8.2 de extra tabellen die nodig zijn om e-mails te kunnen sturen en reproduceren.



Figuur 8.7: Databaseontwerp: projecten, actoren, admins, users en cliënten



Figuur 8.8: Databaseontwerp: meetinstrumenten (vragenlijsten en extensies) samen met hun instanties en antwoorden



Figuur 8.9: Databaseontwerp: E-mails voor actoren

8.3 ORM

Om te voorkomen dat we direct op de database moeten werken, hebben we ervoor gekozen om het eerder besproken model in een groot aantal klassen te implementeren in de *core* module (zie sectie 8.5). De relatie tussen de velden van de klassen en de database zijn gedefinieerd door middel van een *Object Relational Mapper*. De door ons gebruikte ORM *SqlAlchemy* is een veelgebruikte ORM bibliotheek voor Python, en die maakt het mogelijk om het type database (bijvoorbeeld MySQL, Microsoft SQL Server, SQLite) te abstraheren zodat we voor de rest van de code slechts met de objecten en – in uitzonderlijke gevallen – direct met de SQLAlchemy tabel-objecten kunnen werken.

Het gebruik van een ORM heeft ook een voordeel voor de geschreven *unit-tests* (zie 10.2): zonder enige code te hoeven veranderen kunnen we voor elke unit-test een virtuele lege SQLite database gebruiken waarin de benodigde gegevens worden geplaatst. Maar voor de serverapplicatie in de uiteindelijke hostomgeving kan de eigenlijke MySQL database worden gebruikt.

8.4 REST communicatie

De communicatie tussen het front-end (bijvoorbeeld een website) en het back-end (de serverapplicatie) verloopt via de zogenaamde *REST* architectuur, wat staat voor *Residual State Transfer*. Het idee wat hieraan ten grondslag ligt is dat de bestaande HTTP infrastructuur van het internet heel geschikt is om internetservices mee aan te bieden, en dat er niet iets nieuws voor hoeft

te worden bedacht. Door gebruik te maken van de bestaande HTTP status- en foutcodes, bestaande HTTP requesttypes en andere principes die al wijdverbreid worden gebruikt op internet om webpagina's te ontvangen, is de overhead van een REST applicatie heel klein.

Het HTTP protocol definieert een aantal – op internet veelgebruikte – *request*-types, waaronder: *GET* voor het opvragen van een pagina, *POST* voor het versturen van een formulier, *PUT* voor het uploaden van een bestand, en *DELETE* voor het verwijderen van een pagina. Welke request-types er beschikbaar zijn hangt af van de context. Na elke request stuurt de ontvanger daarvan een *response* terug, met daarin eventuele relevante data en een statuscode die aangeeft of de bewerking geslaagd is, en zo niet, wat dan het probleem is.

Op het internet worden *resources* aangeduid met een *URL*. Bijvoorbeeld, achter de URL `http://nos.nl/audio/239577-tu-delft-opent-afdeling-china.html` schuilt een nieuwsartikel resource. Als men in de webbrowser een URL intypt en op `Enter` drukt, dan wordt er een GET request naar de betreffende server gestuurd met daarbij de volledige URL van de op te vragen resource. De server antwoord dan met de relevante gegevens uit de resource, in dit geval de inhoud van het nieuwsartikel. Op dezelfde manier gebruikt de serverapplicatie resources en hun URL's om allerlei functionaliteiten aan het front-end aan te bieden. Een GET request op de *authenticatie* resource (die bijvoorbeeld te vinden zou kunnen zijn op de URL `https://soffos.eu/server/authenticate` geeft terug of er een gebruiker is ingelogd, en welke rechten deze heeft. Ook de POST, PUT en DELETE requests worden door de serverapplicatie gebruikt, afhankelijk van de betreffende resource en de context.

Voor een uitgebreide beschrijving van de motivatie om voor REST te kiezen, zie sectie D.3 in bijlage D.

8.5 Modules

Om de serverapplicatie zo onderhoudbaar mogelijk te houden, hebben we ervoor gekozen om het geheel op te delen in verscheidene modules. De volgende modules zijn gedefinieerd:

core De basismodule waarin voor allerlei componenten de algemene interface gedefinieerd wordt, en waarin basisklassen gedefinieerd zijn die door allerlei modules gebruikt gaan worden.

dic De module met de *Dependency Injection Container* (DIC) en de bijbehorende configuratie. Deze gebruiken we om de modules echt goed te kunnen scheiden, met zo min mogelijk afhankelijkheden. Zie D.8 in bijlage D voor meer informatie over de gebruikte Inversion of Control bibliotheek.

actors Deze module bevat de resources (objecten met GET, POST, PUT en DELETE methodes) die te maken hebben met het kunnen aanmaken, bewerken en verwijderen van actoren en hun onderlinge relaties (zoals die van user-cliënt).

authentication In deze module staan alle klassen waarmee actoren kunnen inloggen, uitloggen, en waarmee geverifieerd kan worden dat de huidige ingelogde actor de rechten heeft om een bepaalde actie uit te voeren.

communication Deze module bevat de server en het sessie object die verantwoordelijk zijn voor de communicatie met een front-end.

language De resources die te maken hebben met het kunnen toevoegen van talen zitten in deze module.

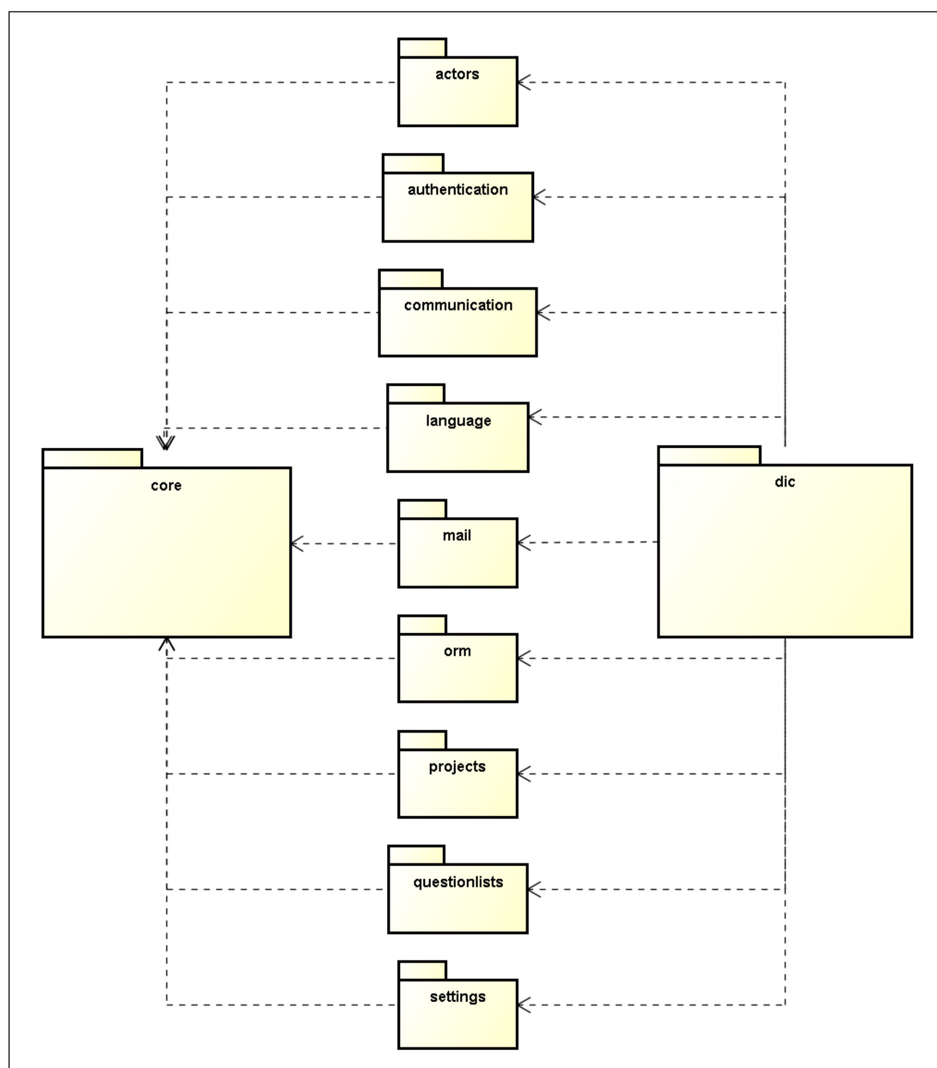
mail Alle resources die mails kunnen versturen, reproduceren en bewaren, en mails kunnen genereren vanaf een template, zitten in deze module.

orm De ORM relaties tussen de objecten en de database worden in deze module ingesteld.

projects Resources voor het maken, bewerken en verwijderen van projecten en het kunnen toevoegen van users en meetinstrumenten aan projecten gebeurt in deze module.

questionlists Het kunnen aanmaken, bewerken en verwijderen van vragenlijsten, extensies en vragen, het toewijzen van een vragenlijst aan een cliënt, en het opslaan van de ingevulde meetinstrumenten gebeurt met de resources uit deze module.

settings Om te voorkomen dat voor wijzingen in de instellingen (zoals de poort waarop de applicatie luistert, of de map waarin tijdelijke gegevens worden opgeslagen) de broncode moet worden aangepast, bevat deze module een aantal klassen die het lezen van instellingen uit een bestand vergemakkelijkt.



Figuur 8.10: Overzicht van de modules en hun onderlinge relaties.

Hoofdstuk 9

Implementatie

In dit hoofdstuk zullen we uitgebreid in gaan op de implementatie van de serverapplicatie. Hierbij zullen we verschillende technische bijzonderheden van het systeem bespreken, en laten we zien hoe het systeem makkelijk uitbreidbaar is en hoe de serverapplicatie aangesproken kan worden.

9.1 Communicatie tussen back-end en front-end

De communicatie tussen de serverapplicatie back-end en de mogelijke front-ends moet goed gespecificeerd worden. De communicatie tussen de serverapplicatie en de front-end verloopt via *REST*, wat in het kort inhoud dat de methodes worden aangeroepen door middel van HTTP requests. Zie hoofdstuk 8.4 voor een uitgebreidere beschrijving van REST. Elke resource in de serverapplicatie heeft een unieke URL, en kan worden aangeroepen door een GET, POST, PUT of DELETE HTTP request te doen op de resource, met de juiste parameters. De serverapplicatie zal dan reageren met een HTTP statuscode en eventuele relevante informatie, gecodeerd als JSON. Het aanroepen van een request die niet beschikbaar is, niet bestaat of niet toegestaan is (vanwege het huidige authenticatieniveau van de ingelogde gebruiker) resulteert in een HTTP error, zoals bijvoorbeeld `not_authenticated` (HTTP errorcode 401), `invalid_arguments` (HTTP errorcode 400) en `not_found` (HTTP errorcode 404). De verschillende HTTP requests hebben in alle resources dezelfde functie. Deze functies zijn hieronder kort uiteengezet:

GET
: Ophalen van gegevens. Bijvoorbeeld GET op de ActorsResource geeft een lijst van actoren terug.

POST
: Het aanmaken van een nieuw object. Bijvoorbeeld POST op de ActorResource maakt een nieuwe Actor aan, terwijl POST op de ProjectResource een nieuw Project aanmaakt.

PUT
: Het aanpassen van een bestaand object. Bijvoorbeeld PUT op de ClientResource wordt gebruikt om de persoonlijke gegevens aan te passen.

DELETE
: Het verwijderen van een object. Bijvoorbeeld DELETE op de GroupResource zal de gegeven groep verwijderen.

De verschillende resources die aanspreekbaar zijn bevinden zich elk in een apart codebestand. Omdat de verschillende methodes volledig zijn gedocumenteerd, is het mogelijk om automatisch de specificaties te genereren. Deze zijn te vinden in de *Communication specification* in bijlage E. In deze documentatie is te vinden welke argumenten verplicht en welke optioneel zijn voor de resource en welke verschillende HTTP errorcodes er teruggegeven kunnen worden.

9.2 Requirements terugzoeken

Aan het eind van het bacherloreindproject wordt een prototype van het systeem afgeleverd aan Soffos. Aan dit prototype moet nog verder gewerkt worden; het is dus nog niet klaar voor gebruik. Wanneer iemand hieraan verder werkt moet diegene een goed beeld krijgen van welke requirements er geïmplenteerd zijn en welke nog niet. Om ervoor te zorgen dat de serverapplicatie makkelijk uitbreidbaar is voor anderen dan onze projectgroep, hebben we een document geschreven waarin staat welke requirements geïmplementeerd zijn en waar in de broncode de implementatie van elke requirement te vinden is. Op deze manier zijn de requirements makkelijker controleerbaar, onderhoudbaar en uitbreidbaar. Dit document is het *Requirements implementation* document, te vinden in bijlage C.

9.3 Technische bijzonderheden

In deze sectie zullen we verschillende technische bijzonderheden van de serverapplicatie bespreken. Hieronder valt het *volgordealgoritme*, het *gijzelaarssysteem*, de *invoercontrole bij requests* en de *JSON-serialisatie van objecten*. Dit zijn allemaal essentiële onderdelen van deze serverapplicatie.

9.3.1 Volgordealgoritme

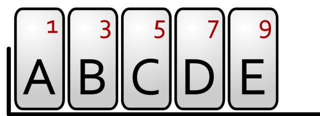
Er is een requirement dat de volgorde van de vragen in een vragenlijst vast ligt. We hebben het volgende algoritme gebruikt voor het oplossen van dit probleem. Voor het bepalen van de volgorde van vragen binnen een vragenlijst of extensie, hebben we elke vraag een volgordenummer gegeven bij het toevoegen aan een vragenlijst of extensie. Deze volgordenummers zijn niet aansluitend. De volgorde van vragen binnen een vragenlijst is te bepalen door de vragen oplopend te sorteren op volgordenummer en ze in die volgorde weer te geven. We hebben ervoor gekozen om het sorteren niet de verantwoordelijkheid te laten zijn van de front-end. In plaats daarvan verwachten we vanuit de front-end een *identifier* van een vraag die al in de vragenlijst zit; de `previous_id`. De gespecificeerde vraag wordt dan ingevoegd na die vraag. Het algoritme dat we gebruiken om de volgordenummers van een vraag binnen een vragenlijst te bepalen is verkort weergegeven in het codevoorbeeld onderaan deze sectie.

Ons algoritme bepaalt de `previous` en de `next` question, de huidige vraag wordt onthouden in `current`. `Current` krijgt een volgordenummer dat het gemiddelde is van het volgordenummer van `previous` en `next`, naar boven afgerond: $\lceil \frac{previous+next}{2} \rceil$. Een voorbeeld hiervan is gegeven in figuur 9.1. In figuur 9.1(a) wordt een vragenlijst weergegeven met vijf vragen (A, B, C, D, E) en hun volgordenummers $(1, 3, 5, 7, 9)$. In figuur 9.1(b) wordt een simpele wijziging doorgevoerd: vraag E wordt voor vraag B geplaatst. Het nieuwe volgordenummer van vraag E wordt nu $\frac{1+3}{2} = 2$.

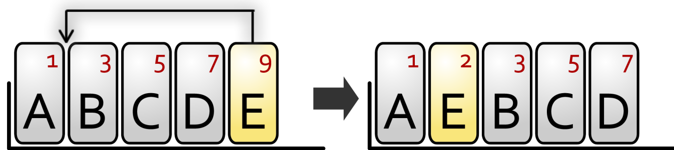
Het kan zijn dat `current` en `next` door het verplaatsen van `current` hetzelfde volgordenummer krijgen. Een voorbeeld hiervan staat in figuur 9.1(c): vraag D wordt ingevoegd vóór vraag E. Hierdoor krijgt D volgordenummer 2, wat gelijk is aan het volgordenummer van E. Dus de vólgende vraag E (`next`) moet nu als het ware ingevoegd worden ná D (`current`), dus wordt

`current previous` en `next current`, en wordt de nieuwe `current` E ingevoegd na `previous` D. Op die manier propageert het veranderen van het volgordenummer door de verzameling vragen totdat alle vragen een uniek volgordenummer hebben binnen de vragenlijst. Het algoritme kijkt dus nu naar vraag E, deze vraag wordt ingevoegd na vraag D. Vraag E krijgt nu volgordenummer 3. Nu is er weer een probleem: vragen E en B hebben beide volgordenummer 3. Vraag B wordt nu ingevoegd na vraag E en krijgt dus volgordenummer 4. Omdat nu alle vragen een uniek volgordenummer hebben binnen de vragenlijst was dit de laatste stap.

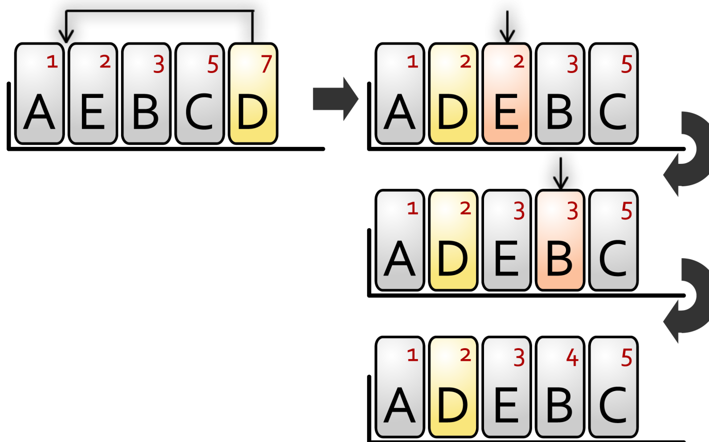
We hebben ervoor gekozen om bij het toevoegen van een nieuwe vraag aan het einde van een vragenlijst een volgordenummer te kiezen wat niet direct volgt op het laatste volgordenummer. Elke nieuw toegevoegde vraag krijgt een volgordenummer wat `gap_size` hoger ligt dan het hoogste volgordenummer in de vragenlijst (namelijk dat van de laatste vraag). Door een macht van 2 te kiezen voor `gap_size` kost het minstens $log_2(gap_size)$ operaties voordat er een volgordenummer-conflict ontstaat tussen de laatste vraag en de nieuw ingevoegde vraag.



(a) Situatie vooraf



(b) Een simpele wijziging



(c) Een ingewikkeldere wijziging

Figuur 9.1: Illustratie van het volgordealgoritme.

Volgordealgoritme

```
1  def moveQuestion(self, questionlist_id, question_id, previous_id):
2      previous = QuestionListQuestion.get(self.server.db,
           questionlist_id, previous_id)
3      current = QuestionListQuestion.get(self.server.db,
           questionlist_id, question_id)
4      gap_size = 16
5
6      while True:
7          next = QuestionListQuestion.getNext(self.server.db,
               questionlist_id, previous.question_id if previous is not
               None else None)
8          if (next is not None and current.question_id == next.
               question_id):
9              if (previous is None):
10                 next = QuestionListQuestion.getNext(self.server.db,
                       questionlist_id, next.question_id)
11             else:
12                 break
13
14         if (previous is not None and next is not None):
15             order = math.divideCeil((next.order + previous.order), 2)
16         elif (previous is not None):
17             order = previous.order + gap_size
18         elif (next is not None):
19             order = math.divideCeil(next.order, 2)
20         else:
21             order = gap_size
22
23         current.order = order
24         previous = current
25         current = next
26
27         if (current is None or order != current.order):
28             break
```

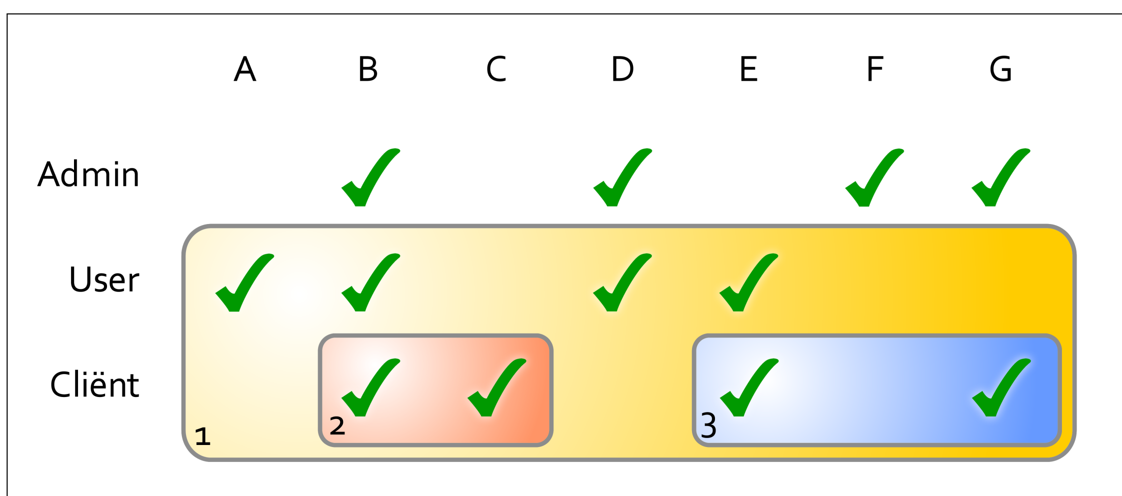
9.3.2 Gijzelaarssysteem

Requirement 6 en 22 van de *should haves* stelt dat het mogelijk moet zijn voor een admin of een user om een andere actor te "gijzelen". Hieronder wordt verstaan dat iemand zich voor kan doen voor de applicatie als een andere actor. Hierbij moet het als admin mogelijk zijn een willekeurige user of cliënt te kunnen gijzelen en voor een user een cliënt die gekoppeld is aan hem of haar. Het overnemen van een andere actor is in een aantal situaties erg handig, bijvoorbeeld wanneer een cliënt niet beschikt over een computer met internetverbinding. In dit geval kan de cliënt zijn vragenlijst op papier invullen en geven aan zijn user. Vervolgens kan de user door deze cliënt te gijzelen, de vragenlijst voor hem invullen.

Om het gijzelen van een actor mogelijk te maken hebben we een resource ontwikkeld die dit regelt voor het gehele systeem. Wanneer er een andere actor wordt overgenomen zal de huidige

ingelogde gebruiker in de serverapplicatie veranderen. Omdat een actor meerdere rollen heeft, kan een admin dus ook een cliënt-rol hebben. Wanneer een user deze cliënt overneemt, is het niet de bedoeling dat ook de admin-rol overgenomen wordt, want anders kan een user op die manier een admin-rol overnemen en kan hij of zij meer in het systeem dan de bedoeling is. Om dit te voorkomen is het alleen mogelijk om de rollen van een andere actor over te nemen die lager zijn dan de hoogste rol van de actor die iemand over wilt nemen.

In figuur 9.2 staat een situatie geïllustreerd waarin te zien is hoe het gijzelaarssysteem werkt. Links staan de verschillende rollen en bovenin staat de verschillende actoren van het systeem. Elke actor die een admin is kan alle rol/actor combinaties overnemen die zich bevinden in vlak 1. Een admin kan dus niet een andere admin-rol overnemen. In dit voorbeeld kunnen actoren A en B de rol/actor combinaties uit vlak 2 overnemen. Ook hier geldt dus weer dat alleen een rol overgenomen kan worden als deze lager is dan de hoogste rol van de huidige actor. Hetzelfde geldt voor actor E die alleen de rol/actor combinaties over kan nemen uit vlak 3.

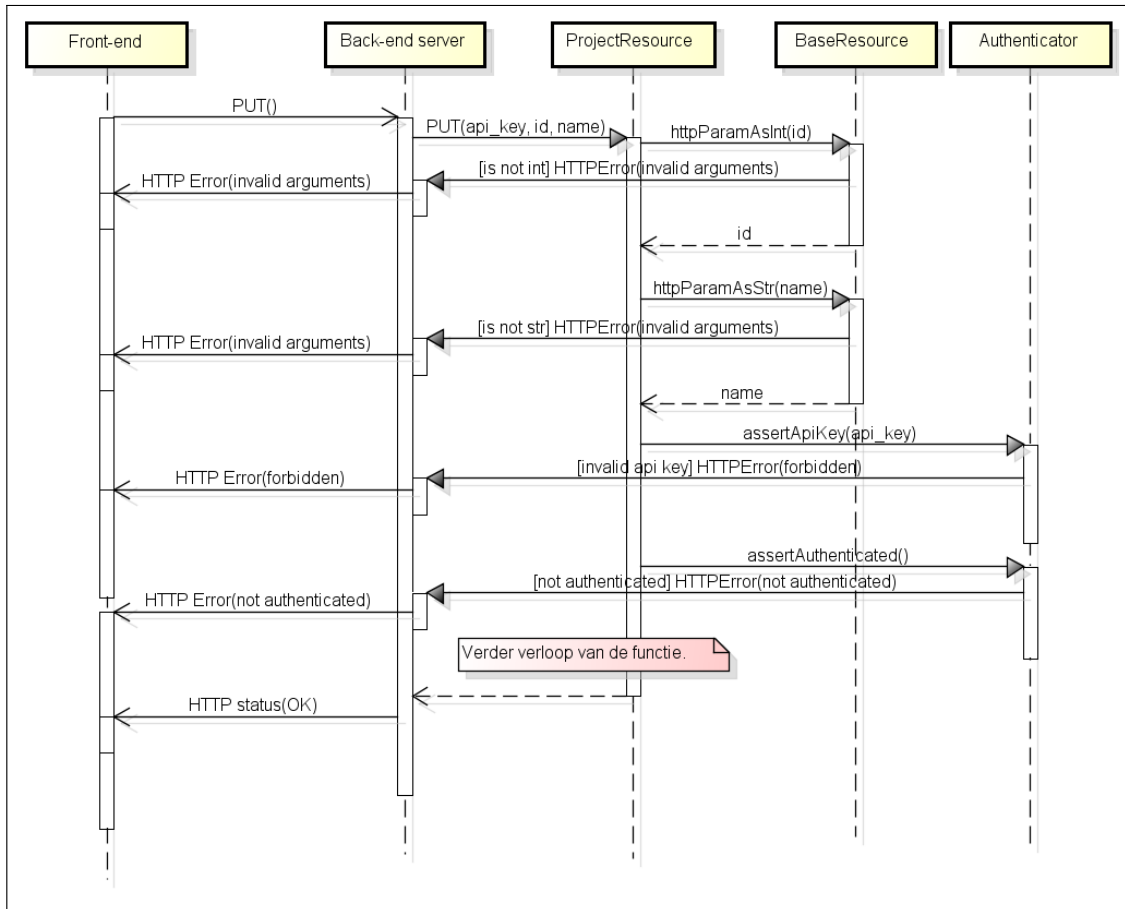


Figuur 9.2: Illustratie van het gijzelaarssysteem.

9.3.3 HTTP Invoercontrole

De serverapplicatie maakt gebruik van de verschillende requesttypes van HTTP, zoals GET en POST. Omdat de data hierbij verzonden wordt als tekst, is het niet mogelijk om bepaalde argumenten met bepaalde types te versturen. Om ervoor te zorgen dat er enige controle is over het type van het argument dat verstuurd wordt hebben we een aantal methodes ontworpen om de invoer te controleren. De verschillende types waarop de input kan worden gecontroleerd zijn: `integer`, `string`, `date`, `boolean`, `datetime`, `list` (van de eerdergenoemde types) en `dict` (dictionary).

Wanneer na controle blijkt dat een bepaald argument niet het vereiste type heeft zal de serverapplicatie een *invalid arguments* HTTP errorcode (400) terugsturen naar de front-end die de resource probeert aan te spreken. Een illustratie van dit proces is te zien in figuur 9.3.



Figuur 9.3: Sequence diagram van een standaard HTTP request.

9.3.4 JSON-serialisatie van objecten

Door het gebruik van een *Object-Relational Mapper* (ORM) framework bestaaner in de server-applicatie objecten die corresponderen met tabellen in de database. Bij een GET-request op de back-end moeten deze objecten worden *geserialiseerd* naar JSON codering.

Om dit te realiseren heeft elk ORM object een methode geërfd van zijn superklasse `BaseObject` waarin de serialisatie plaats vindt. In deze methode worden alle attributen binnen het object omgezet in JSON codering. Als een attribuut als waarde een ander `BaseObject` heeft, dan wordt dat subobject ook geserialiseerd. Omdat sommige attributen niet openbaar gemaakt mogen worden (zoals een wachtwoordhash) is het mogelijk om aan te geven welke attributen moeten worden uitgesloten. Door deze 'regels' standaard vast te leggen voor elk ORM objecttype was het vervolgens erg makkelijk om objecten om te zetten naar JSON codering.

Zoals in het codevoorbeeld hieronder te zien is, wordt verschillende typen attributen verschillend behandeld. De types die worden onderscheiden zijn: een `BaseObject` (regel 1), een `datetime` of `date` object (regel 4), een lijst van `BaseObjects` (regel 6) en een dictionary van `BaseObjects` (regel 14). Wanneer de waarde niet aan een van de bovenstaande voorwaarde voldoet, geen private attribuut is en geen methode, zal de string-representatie van de attribuutwaarde gebruikt worden (regel 23).

JSON-serialisatie van objecten

```python
if recursive and isinstance(value, BaseObject):
    # Convert the BaseObject into dictionary form.
    dictionary[key] = value.toDictionary()
elif isinstance(value, datetime.datetime) or isinstance(value, datetime.date):
    dictionary[key] = value.isoformat()
elif isinstance(value, list):
    # Include the list, where each BaseObject is converted into dictionary form.
    dictionary[key] = []
    for bs in value:
        if isinstance(bs, BaseObject):
            dictionary[key].append(bs.toDictionary())
        else:
            raise TypeError()
elif isinstance(value, dict):
    # Include the dictionary, where each BaseObject value is converted into dictionary form.
    dictionary[key] = {}
    for k, bs in value.iteritems():
        if isinstance(bs, BaseObject):
            dictionary[key][k] = bs.toDictionary()
        else:
            raise TypeError()
elif not callable(value) and not key.startswith('_'):
    dictionary[key] = value
```

9.4 SOLID

De SOLID principes (single responsibility, open/closed, Liskov substitution, interface segregation, dependency inversion) zijn oorspronkelijk bedacht voor statische object-georiënteerde programmeertalen, zoals Java en C#. Om ze te kunnen toepassen op de Python serverapplicatie is lastig, omdat Python een dynamische programmeertaal is. In statische programmeertalen is er de garantie dat als een methode om een object van een bepaald type vraagt, de methode die ook krijgt (en niet iets compleet anders). In een dynamische programmeertaal zoals Python is die garantie er niet, en kan dan ook niet worden aangegeven welk type objecten een methode wil ontvangen. Er zijn veel discussies gevoerd over hoe de SOLID principes dan toch in een dynamische taal kan worden gebruikt, en een overzicht van de door ons gekozen oplossingen staat in sectie D.2 van het Oriëntatierapport in bijlage D. Nu volgt een beschrijving van hoe wij de vijf SOLID principes uiteindelijk hebben geïmplementeerd.

9.4.1 Single responsibility principle

Volgens het *single responsibility principle* moet voorkomen worden dat een klasse meerdere redenen om te veranderen kan hebben die resulteren is verschillende veranderingen in de code van de klasse. Bijvoorbeeld, een iPhone die kan bellen en foto's maken heeft twee verantwoordelijkheden, en als er een vijf keer zo grote lens in zou moeten worden gezet, dan moet de hele

iPhone veranderen. In de serverapplicatie zijn de responsibilities redelijk goed gescheiden. Het is niet altijd mogelijk om van te voren te anticiperen welke veranderingen er ooit zouden kunnen optreden, en dus of de verantwoordelijkheden wel voldoende gescheiden zijn, maar het is nog niet voorgekomen dat een verandering in één klasse grote veranderingen in een andere – schijnbaar ongerelateerde – klasse teweeg bracht.

9.4.2 Open/closed principle

Volgens het *open/closed principle* moet het makkelijk zijn om functionaliteit later uit te breiden zonder dat dit tot gevolg heeft dat de oorspronkelijke code van de functionaliteit veranderd moet worden. Bijvoorbeeld bij de implementatie van een sorteerfunctie, kan men ervoor kiezen om intern een functie die twee objecten vergelijkt te gebruiken, of een *hook* te maken waarmee later een andere vergelijkingsfunctie ingezet kan worden, om zo de functionaliteit uit te breiden. Het open/closed principle is meer van toepassing bij het schrijven van een functionaliteitenbibliotheek dan op onze serverapplicatie, maar omdat de applicatie is geschreven in Python krijgen we wat ondersteuning voor het open/closed principle er gratis bij. In Python kunnen alle functies worden *overstemd* (*overridden*), en dankzij Python's *duck-typing* kan elk object in de plaats van een ander object worden gebruikt. Hiermee kan bestaande functionaliteit worden uitgebreid.

9.4.3 Liskov substitution principle

Volgens het *Liskov substitution principle*, zoals dat op de enige realistische manier is toe te passen in Python, moeten functies met dezelfde naam en hetzelfde aantal argumenten onderling uitwisselbaar zijn zonder dat dit de veronderstelde pre- en postcondities van de functie verandert. Dit komt doordat Python een dynamische taal is, waarin er geen garantie is over het type van een objectinstantie, en dus ook geen garantie over de functies die door de instantie worden geïmplementeerd. In de praktijk blijkt het lastig om functies altijd een unieke naam te geven, omdat dan de namen onnodig verboos worden. Toch is het gelukt om alle functienamen in de serverapplicatie uniek te maken, op twee uitzonderingen na: statische functies en resource request functies. Statische functies worden nooit op een objectinstantie aangeroepen, en dus is het altijd ondubbelzinnig vast te stellen welke methode wordt aangeroepen. Hierdoor konden we de gelijknamige functies `Project.get` en `Client.get` implementeren die respectievelijk een `Project`-object en een `Client`-object retourneren, en dus verschillende postcondities hebben. De resource request functies (zoals `POST`, `GET` en `DELETE`) moeten die specifieke namen hebben omdat het door ons gebruikte serverframework dit vereist. Gelukkig worden deze functies nooit direct aangeroepen op een instantie, dus kan er ook nooit verwarring zijn over welke methode wordt aangeroepen.

9.4.4 Interface segregation principle

Interface segregation is niet van toepassing op Python, omdat Python geen interfaces kent. Python gebruikt automatisch alleen de benodigde subset van functies, dus is er ook geen noodzaak om de subset expliciet te definiëten in een interface.

9.4.5 Dependency inversion principle

Volgens *dependency inversion* moeten de afhankelijkheden worden omgekeerd. Een klasse A moet niet afhankelijk zijn van een specifieke implementatie B. In plaats daarvan moet A juist via *dependency injection* een instantie van een klasse krijgen die voldoet aan het contract van klasse B (een klasse die uitwisselbaar is met B). Dankzij de *Spring Python* dependency injection container

(zie D.8) was het gemakkelijk om *inversion of control* toe te passen in de serverapplicatie. Alle benodigde objecten – zoals een authenticatieobject, een mailer, de server of de database – worden via *constructor injection* aan de verschillende klassen gegeven. Dit betekent dat op het moment dat er een instantie van een klasse wordt gemaakt de benodigde objecten worden meegegeven, en dat er verder geen afhankelijkheden meer zijn tussen de klassen en de implementaties van andere klassen.

9.5 Code conventies

Vanwege de betrouwbaarheid en onderhoudbaarheid van de serverapplicatie hebben we ons gehouden aan een aantal code conventies. Door rekening te houden met deze conventies tijdens het implementeren wordt de kans groter om fouten te ontdekken in een vroeg stadium. Hieronder staat een overzicht van een aantal conventies waar we rekening mee hebben gehouden.

Documentatie en commentaar
Door het toepassen van test-driven development werden we geforceerd om elke methode en klasse te voorzien van geschikte documentatie en commentaar. Hierbij hebben we uitgebreid de argumenten beschreven, welk type het argument is en waar het voor dient. Ook hebben we de verschillende error codes beschreven die de methode kan geven en de pre- en postcondities uitgelegd. Verder staat bij elke methode de auteur, waardoor het makkelijk is om iemand aan te spreken over de werking van de methode.

Precondities
De precondities van een interne methodes worden gecontroleerd door het ingebouwde `assert` keyword. Deze checks zijn voornamelijk simpele type checks en controle of een argument `None` is of niet. De precondities voor een request op een resource bestaan voor een groot deel uit het controleren van het type van een argument. Hierbij worden de in sectie 9.3.3 besproken methodes gebruikt.

Extra argumenten requests
Bij het maken van een request, vanuit bijvoorbeeld PHP, kunnen er allerlei verschillende argumenten meegestuurd worden die niet gebruikt worden in de resource. Wanneer er te veel of te weinig argumenten worden meegestuurd bij een request zal dit resulteren in een `404 not found` error. Om dit tegen te gaan moeten alle argumenten (zowel verplichte als optionele argumenten) die de resource gebruikt worden aangegeven in de methode, en de argumenten `*args` en `**kwargs` moeten worden toegevoegd om de extra argumenten op te vangen. Het argument `*args` zal de argumenten opvangen die zonder naam zijn meegestuurd – dit is dus een lijst – en `**kwargs` vangt alle argumenten op die wel een naam hebben gekregen – dit is dus een dictionary.

Opdeling in modules
Elk bestand in Python is een apart module. Om de code zo overzichtelijk mogelijk te maken hebben we ervoor gekozen om in een module (bestand) precies één klasse te definiëren. Hierdoor is het makkelijker om bepaalde code terug te kunnen vinden.

Hoofdstuk 10

Codekwaliteit

In dit hoofdstuk bespreken we onze aanpak om de kwaliteit van de code te garanderen. Het afleveren van hogekwaliteits-code was één van onze doelen voor dit project, en daarnaast vinden we het belangrijk dat de serverapplicatie eenvoudig uitbreidbaar en onderhoudbaar is. Hiervoor is goede documentatie onmisbaar. Om ervoor te zorgen dat de code voor ons een acceptabele kwaliteit heeft, hebben wij gebruik gemaakt van unittests en gebruikerstests. Om de onderhoudbaarheid van de code te controleren, is de code naar SIG gestuurd en door ons zelf gecontroleerd middels een programma dat mogelijke fouten opspoort.

10.1 Unittests

Zoals besproken in hoofdstuk 4 passen wij *test-driven development* toe. In de praktijk houdt dat in dat we altijd eerst documentatie en unittests schrijven voordat een onderdeel geïmplementeerd wordt. Het samenvoegen van alle unittests geeft een zeer uitgebreide testsuite waarmee de serverapplicatie automatisch getest kan worden. Uit onze ervaring is gebleken dat de testsuite fouten in de code vaak makkelijk op kan sporen.

10.2 Gebruikerstest

De serverapplicatie heeft in principe geen gebruikersinterface. Dit maakte het lastig om een gebruikerstest uit te voeren. We hebben daarom besloten om een testwebsite te maken om de functionaliteit van de serverapplicatie makkelijk te kunnen demonstreren. Bij deze gebruikerstest wilden we de volgende punten testen:

- Een groep van ongeveer 15 mensen tegelijk, die:
  - Correcte informatie invoeren versus foutieve informatie invoeren.
  - Formulieren volledig invullen versus velden openlaten bij het invullen.
  - Allerlei invoer gebruiken
  - Allerlei functionaliteiten testen
  - Zoeken naar bugs
- Iemand die na een periode (bijvoorbeeld een uur) van rust voor het systeem begint met testen.

Soffos regelde een groep van ongeveer 20 personen die tegelijkertijd de serverapplicatie konden testen. Hierbij werd elke actie die een *error* (soort fout) opleverde gelogd en vroegen we de testpersonen commentaar te geven over de werking van het systeem. Uit het logbestand bleek dat er een aantal keer een relatief onschadelijke error voor kwam, maar de serverapplicatie crashte niet. Deze errors ontstonden soms wanneer een gebruiker een bepaald speciaal teken gebruikte in een invoerveld. De tekens waren niet goed gecodeerd en de server kon hier niet meer overweg, wat resulteerde in een server error (HTTP error code 500).

Het commentaar van de testpersonen bestond voornamelijk uit opmerkingen over de userinterface, maar over de werking van functionaliteiten waren er maar enkele klachten. Een voorbeeld van een bug die gevonden was door de testpersonen ontstond bij het toevoegen van een bestaande extensievraag aan een extensie: hierbij gaf de serverapplicatie onterecht de foutmelding dat er geprobeerd werd een extensievraag aan een vragenlijst toe te voegen of een vraag aan een extensie.

Een andere bug kwam naar voren wanneer een actor bij het wijzigen van zijn wachtwoord niet hetzelfde wachtwoord en controlwachtwoord intypte. In deze situatie trad er geen error op, maar werkte noch het originele wachtwoord noch een van de nieuwe wachtwoorden om in te loggen. De account van de actor was hierdoor onbruikbaar geworden.

Deze verschillende bugs hebben we na de gebruikerstest aangepast. Hiervoor schreven we per bug een testcase die deze bug uitlokt. Deze testcase faalde in eerste instantie, maar na de wijzigingen in de implementatie slaagden de testcases.

10.3 SIG beoordeling

Tijdens het bachelorproject wordt alle door het projectteam geschreven code twee keer ingestuurd naar SIG. SIG staat voor Software Improvement Group, dit is een bedrijf dat code beoordeeld op onderhoudbaarheid. De eerste beoordeling vond plaats op 14 juni en de tweede beoordeling zal plaatsvinden na het inleveren van dit eindverslag.

Uit de eerste beoordeling is gebleken dat de onze code ruim drie sterren scoort op een schaal van vijf. Dit houdt in dat onze code bovengemiddeld onderhoudbaar is. SIG was erg tevreden met de hoeveelheid test-code om het systeem automatisch te testen. De code van de serverapplicatie scoorde slecht op *Unit Size* en *Unit Complexity*. Laag scoren op unit size betekent dat in onze code veel methodes voorkomen die langer zijn dan wenselijk. Laag scoren op unit complexity betekent dat in de code veel methodes voorkomen die bovengemiddeld complex zijn. In onze code bleek dat de meest complexe methodes ook de langste methodes zijn. Deze twee problemen zijn met dezelfde oplossing op te lossen: splits methodes op.

Het oordeel van SIG is dat de onderhoudbaarheid van de serverapplicatie makkelijk te verbeteren is met een relatief kleine inspanning. Voor de tweede beoordeling zullen we lange en complexe methodes opsplitsen in kortere methodes. Dit zal ervoor zorgen dat de methodes sneller te begrijpen en daarmee ook makkelijker te testen zijn. Ook zullen we nog enkele kleine schoonheidsfoutjes verbeteren, zoals de aanwezigheid van code in commentaar.

10.4 Pylint

Na de eerste beoordeling van de SIG zijn we gebruik gaan maken van *Pylint*. Pylint controleert de code op verschillende conventies en berekent een cijfer aan de hand van de resultaten. Dit cijfer is op een schaal van -40 tot 10, waarbij wordt aangegeven dat een code boven de 7 als goed bestempeld wordt. De serverapplicatie kreeg na een eerste meting het cijfer 5,1. Na verschillende delen te hebben gesimplificeerd en de verschillende warnings en errors te hebben opgelost hebben

we het cijfer kunnen ophalen tot een 9,98. Hierbij bleven er een aantal warnings over die niet gemakkelijk opgelost konden worden.

Door het gebruik van Pylint hebben we verschillende onopvallende bugs kunnen oplossen die onze testsuite niet had kunnen vinden, bijvoorbeeld het missen van de `*args` en `**kwargs` argumenten zoals besproken in sectie 9.5 Code conventies.

Deel IV

Resultaten en aanbevelingen

Hoofdstuk 11

Resultaten

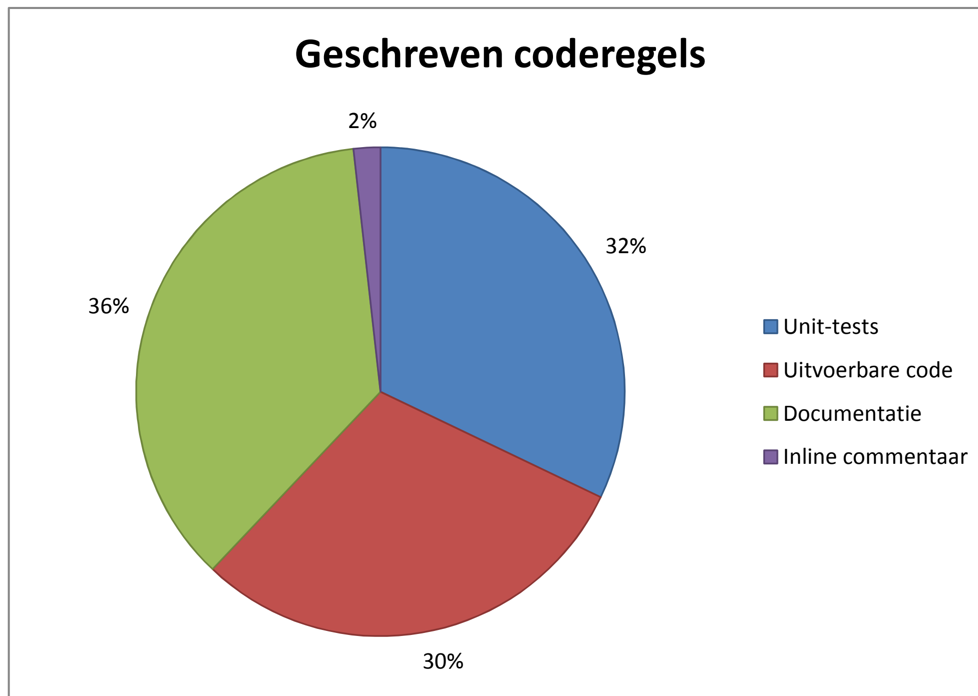
Vanwege de Agile-methodiek die we gekozen hebben hadden we in eerste instantie niet exact vastgesteld wat er aan het einde van het project af zou zijn. Naarmate het project vorderde kregen we een beter beeld van wat er vóór de eindpresentatie afgerond zou zijn. Behalve één onderdeel, namelijk de CVS dump module, is dit uiteindelijk ook allemaal afgerond.

De serverapplicatie kan vragenlijsten en extensies, vragen en antwoorden, projecten, groepen, actoren beheren, en e-mails versturen en reproduceren. Actoren kunnen meetinstrumenten invullen, in- en uitloggen en er is een compleet werkend rol-gebaseerd authenticatiesysteem. Alle titels en teksten van objecten die zichtbaar zijn voor users of cliënten kunnen worden opgeslagen en opgehaald in meerdere talen. De serverapplicatie heeft ondersteuning voor SSL, en voor speciale karakters zoals ë en å. Verder logt de applicatie alle acties en fouten die zich voordoen.

Zoals gezegd kon de CSV dump niet binnen de projecttijd afgerond worden. Daarnaast moeten *dynamische antwoordtypen*, *vragengroepen*, notificaties en rapporten na dit project een keer afgerond worden, zoals beschreven in hoofdstuk 12.

11.1 Geschreven code

Als de lege regels niet worden meegeteld, bevat de serverapplicatie op dit moment ongeveer 14.000 door ons geschreven regels code. De verdeling van de soorten geschreven regels code in de uiteindelijke applicatie is weergegeven in figuur 11.1. Wanneer de ongeveer 4500 regels code in de unit-tests niet worden meegenomen in de berekening, is 56% documentatie en 44% uitvoerbare code.



Figuur 11.1: De verschillende soorten geschreven code.

Hoofdstuk 12

Aanbevelingen

Tijdens ons bacheloreindproject hebben we hard gewerkt en een hoop nieuws geleerd. In dit laatste hoofdstuk bespreken we enkele aanbevelingen voor toekomstige studenten. Ook bespreken we enkele ideeën voor de toekomst van onze serverapplicatie.

12.1 Onze aanbevelingen

Wij willen de studenten er op attent maken dat in de opzet van het bacheloreindproject van de studenten veel zelfstandig werk wordt verwacht. De student wordt veel minder aan de hand meegenomen dan in het voorgaande deel van de opleiding. We raden toekomstige studenten aan om ruim op tijd te beginnen met het zoeken van een eindopdracht. Ook willen we de studenten aanmoedigen om op tijd een TU begeleider te zoeken. Als wij hetzelfde project nog een keer zouden doen, zouden we eerder zijn begonnen met zoeken naar een TU begeleider.

Verder is het belangrijk om de requirements van het project vanaf het begin van het project duidelijk te hebben. Zoals genoemd in hoofdstuk 6, is het lastig om eenduidig de requirements op tafel te krijgen, zodat zowel Soffos als het projectteam hetzelfde beeld heeft van het project. Wat ook enorm helpt is het *bevriezen* van de requirements: op een gegeven moment moet je als projectteam de requirements vastleggen en niet meer veranderen. Als de opdrachtgever toch de requirements wil veranderen, moeten die veranderingen worden doorgeschoven totdat de vastgelegde requirements zijn geïmplementeerd. Dat scheelt veel wijzigingen, frustraties en zorgt voor extra duidelijkheid zowel voor de opdrachtgever als het projectteam.

12.2 Toekomstig werk

De serverapplicatie heeft nu, aan het eind ven het bachelorproject, al een hoop functionaliteit. Toch zijn nog lang niet alle requirements geïmplementeerd, maar wel de requirements die we ons hadden voorgenomen, op één na. In ons ontwerp hebben we er vanaf het begin rekening mee gehouden dat het systeem nog jaren mee moet, en het is naar onze mening nog steeds goed uitbreidbaar en onderhoudbaar.

De volgende modules kunnen nog toegevoegd worden aan de serverapplicatie:

- CSV dump, om gegevens uit het systeem te halen.
- Rapporten automatisch kunnen genereren, voor zowel individuën als voor groepen.
- Rekeningen automatisch kunnen genereren.

- Dynamische antwoordtypen mogelijk maken.
- Notificaties, om users en cliënten herinneringen per mail toe te sturen.
- Het indelen van vragenlijsten in vragengroepen.
- Het indelen van extensies in categorieën.

Zonder veel ingrijpende veranderingen moeten de genoemde modules toe te voegen zijn aan het systeem. Door de enorm uitgebreide hoeveelheid documentatie (meer dan 52% van de broncode) en doordat in de huidige code duidelijk is aangegeven waar elke requirement is geïmplementeerd, is de serverapplicatie goed uitbreidbaar.

Deel V

Bijlagen

Bijlage A

Plan van aanpak

A.1 Inleiding

In dit plan van aanpak zullen we onze aanpak voor ons bachelor eind project bespreken. We behandelen de opdrachtomschrijving, de aanpak, de projectinrichting en ten slotte de kwaliteitsborging. Het doel van dit plan van aanpak is om duidelijk te maken wat onze opdracht inhoudt en welke randvoorwaarden bij het project een rol spelen. Een korte planning en uitleg van methodologie zullen ook terugkomen in dit document.

A.1.1 Schets van het bedrijf

Onderzoeksinstituut Soffos is een bedrijf dat medisch onderzoek doet op basis van medische vragenlijsten. Een project wordt bij Soffos aangevraagd door een bedrijf, waarna wordt bepaald welke vragenlijsten hierbij nodig zijn. Behandelaars, zoals een dokter of een psycholoog, behandelen hun clienten. Behandelaars kunnen zich registreren en hun clienten registreren. Daarna kunnen de clienten de voor hen opengestelde vragenlijsten invullen. Aan de hand van deze gegevens worden overzichtelijke rapporten voor de behandelaars gegenereerd. Ook kunnen de verzamelde gegevens worden gebruikt voor het schrijven van een wetenschappelijk artikel.

A.1.2 Achtergrond en aanleiding van de opdracht

Het huidige systeem dat Soffos hanteert voldoet niet meer aan de huidige eisen. Veel informatieverwerking gebeurt handmatig en dit kost een hoop tijd en moeite. Het bedrijf gaat om met medische gegevens waarbij de veiligheid gewaarborgd moet kunnen worden. Dit is helaas nu niet meer het geval, daarom verloopt alle communicatie via e-mail of telefonisch contact. Het is de bedoeling dat een systeem ontwikkeld wordt dat voor al deze problemen een oplossing biedt.

A.2 Opdrachtomschrijving

In dit hoofdstuk worden het verwachte eindproduct en de eisen waaraan dit product moet voldoen uiteengezet. Ook wordt omschreven wat er wordt verwacht van het projectteam en de opdrachtgever.

A.2.1 De opdrachtgever

Research Institute Soffos
Margietstraat 9
5121 XL Rijen
+31 161 225238
info@soffos.eu

A.2.2 Contactpersonen

Olaf Schüsler (Soffos CTO)
+31 6 273 10473
webmaster@soffos.eu

Marij Schüsler (Soffos CEO)
+31 6 150 42431

Taco Schüsler (Soffos legal council)

Fred Schüsler (Soffos financial director)

A.2.3 Probleemstelling

Via de website www.soffos.nl konden gebruikers zich registreren en werden vragenlijsten ingevuld door patiënten. De verwerking van andere informatie, zoals het opstellen van rapporten en rekeningen, werd met de hand gedaan. Deze site is een aantal jaar geleden gebouwd en voldoet niet meer om het groeiende aantal aanvragen af te handelen. Nu het onderzoeksveld uitgebreid is, is er een nieuw systeem nodig.

Momenteel is de website zo onbetrouwbaar geworden dat alle communicatie via de website via andere kanalen wordt afgehandeld. Het insturen van vragenlijsten gebeurt via de e-mail of op papier per post. Alle gegevens worden overgenomen in een groot excel-bestand, waar formules en grafieken uiteindelijk de gewenste informatie tonen. Dit excel-bestand levert rekeningen en rapporten aan in een vooraf vastgestelde vorm.

A.2.4 Doelstelling

In de nieuwe situatie moet de back-end van het systeem toegankelijk zijn vanaf verschillende front-end systemen. De front-end kan bijvoorbeeld uit een website of een administratief programma bestaan. Deze back-end en front-end systemen moeten onafhankelijk van eigen implementatiedetails via een protocol kunnen communiceren. De back-end dient goed onderhoudbaar en uitbreidbaar te zijn, zodat nieuwe vragenlijsten voor cliënten en extensies voor behandelaars. Het nieuwe back-end systeem heeft twee hoofdeisen:

- De back-end moet makkelijk toegankelijk zijn, bijvoorbeeld met RPC of RMI.
- Het complete systeem moet bruikbaar zijn voor een administrator zonder kennis van informatica. Het back-end project wordt gelijktijdig uitgevoerd met een team dat werkt aan de website front-end van het systeem. Dit team bestaat momenteel slechts uit Olaf Schüsler.

A.2.5 Opdrachtformulering

De opdracht bestaat uit het bouwen van een back-end van het nieuw te ontwikkelen vragenlijstensysteem van Soffos. Het systeem dient benaderbaar te zijn vanuit verschillende platforms, de site zelf en aan administratief programma. Het systeem wordt ontwikkeld met het oog op de toekomst, dus het moet uitbreidbaar zijn met nog te ontwikkelen programma's en vragenlijsten.

Het systeem dient eenvoudig toegankelijk te zijn met behulp van bijvoorbeeld RPC en de vragenlijsten en extensies in het systeem dienen te kunnen worden uitgebreid door een admin, die geen kennis heeft van informatica. Er wordt in teamverband gewerkt binnen de projectgroep van 3 studenten en de webmaster van Soffos, die de front-end ontwikkelt.

A.2.6 Deliverables

Voor de TU Delft moet er een tweewekelijks voortgangsrapport, een eindverslag en een eindpresentatie worden gemaakt. Het eindverslag bevat als bijlagen de opdrachtomschrijving, oriëntatieverslag, plan van aanpak en RAD-document met UML.
Vanuit Soffos wordt verwacht dat wij de volgende documenten leveren:

- Software Requirements Specification (SRS), IEEE 830
- Software Design Document (SDD), IEEE 1016
- Software and System Test Document (SSTD), IEEE 829
- Software Quality Assurance Plans (SQAP), IEEE 730
- Technical Design Document (TDD)

Daarnaast zal er vanaf week 3 elke vrijdag een werkende versie uitgebracht worden.

A.2.7 Randvoorwaarden

De volgende voorwaarden worden gesteld aan dit project en onze uitvoering daarvan:

- Het projectteam moet binnen 12 weken een prototype opleveren.
- Het project moet voldoende gedocumenteerd en gestructureerd zijn om uitbreiding in de toekomst mogelijk te maken.

Wij gaan er van uit dat Soffos het volgende voor ons beschikbaar stelt:

- Een server waarop de applicatie gedurende het project meermaals uitgevoerd en getest kan worden.
- Een SVN repository die we kunnen gebruiken voor versiebeheer.
- Een bugtracker of een featuretracker voor ons gehost.
- Soffos zal bereikbaar zijn voor het maken van beslissingen over cruciale keuzes binnen het project.

A.2.8 Risicofactoren

Het gebrek aan ervaring met Python en REST in ons team levert het risico dat het extra tijd kost om bepaalde oplossingen te vinden. De third-party libraries die we in dit project gebruiken kunnen verouderd zijn of bugs bevatten, waardoor het extra tijd kan kosten om een oplossing te vinden.

A.3 Aanpak

Dit hoofdstuk zal ingaan op de methodiek en technieken die gebruikt zullen worden. Ook worden de werkzaamheden gespecificeerd en wordt er een planning gepresenteerd. Aan de hand van de beschreven aanpak zullen wij voldoen aan de in het vorige hoofdstuk beschreven opdracht en gestelde eisen.

A.3.1 Methodiek

We werken volgens de Agile methodiek, met een iteratie van een week. Dit houdt in dat er elke vrijdag wordt een werkend product wordt opgeleverd. Soffos kan dan het product beoordelen en wijzigingen of nieuwe features voorstellen. Aan het eind van het project is er een grote release waarin alles uitgebreid getest is.

De requirements, features en wijzigingen worden in overleg met Soffos geprioriseerd. Elke week wordt er een selectie gemaakt van de features die voor die week op de planning staan. Aan het einde van de week is het doel dat de geselecteerde features geïmplementeerd zijn. De niet (volledig) geïmplementeerde features schuiven door naar de volgende week.

Elke dag maakt iedereen in het projectteam een SVN update van de code en documentatie. Elk teamlid documenteert één enkele feature, schrijft er tests voor en implementeert en test het dan. Daarna wordt er een SVN update gedaan, wordt de code geïntegreerd en getest, en dan gecommit. De SVN bevat nooit code die niet getest of niet werkend bevonden is.

A.3.2 Techniek

Het systeem wordt geïmplementeerd met de programmeertaal Python, met third-party libraries voor REST en databasetoegang. In het Oriëntatieverslag verdiepen wij ons in de technische details van het project en in de precieze libraries die gebruikt zullen worden.

A.3.3 Werkzaamheden

Onze werkzaamheden zullen bestaan uit het vervaardigen van documenten over de back-end en de implementatie van het back-end systeem. Hierbij wordt regelmatig met Soffos overlegd, om te controleren of ons product nog aan de eisen voldoet. De diverse documenten die vanuit Soffos en de TU Delft vereist worden, zijn al beschreven in sectie A.2.6. Bij het implementeren proberen we om code van een zo hoog mogelijke kwaliteit te creëren, zodat latere uitbreidingen en onderhoudswerkzaamheden zo soepel mogelijk uitgevoerd kunnen worden.

A.3.4 Planning

Om de week wordt er een voorgangsrapport naar de TU begeleider gestuurd. We zullen ongeveer 11 weken over dit project doen. Het project kent de volgende fasen:

Oriëntatiefase (week 1-2)
Schrijven van de Requirements Analysis Document, Object Design Document, Plan van Aanpak en Oriëntatieverslag.

Implementatiefase (week 3-9)
Documenteren, implementeren en testen van een selectie van features, wijzigingen en bugfixes, elke vrijdag wordt een werkende build opgeleverd.

Acceptatietests (week 9-10)
Testen of alle componenten onderling goed samenwerken en of het systeem aan de eisen voldoet die Soffos gesteld heeft.

Afrondingsfase (week 10-11)
Schrijven van het verslag, maken van de presentatie, en het presenteren.

A.4 Projectinrichting

Om de wijze waarop het projectteam van plan is het project in te richten zichtbaar te maken, worden in dit hoofdstuk de organisatie en andere afspraken van het project uiteen gezet.

A.4.1 Betrokkenen

Olaf Schüsler zal het technische aspect van ons project begeleiden. Marij Schüsler is de eigenaar van Soffos, en onze opdrachtgever. Zij is inhoudelijk betrokken bij ons project, bijvoorbeeld bij het opstellen van de requirements.

A.4.2 Informatie

In eerste instantie zullen we minstens een keer per week afspreken met de opdrachtgever in persona . Na week 4 kan dit contact indien nodig vervangen worden door contact via Skype. Daarnaast houden we gedurende het gehele project contact via e-mail en telefoon.

A.4.3 Faciliteiten

Er is afgesproken dat wij niet bij het bedrijf op locatie werken, maar op de TU. Dit is omdat onze begeleider, Olaf Schüsler, ook elke dag in Delft aanwezig zal zijn. We zullen onze eigen computers en eventueel ook faciliteiten van de TU gebruiken. Soffos stelt een bugtracker, server en SVN repository beschikbaar.

A.5 Kwaliteitsborging

De voorwaarden uit hoofdstuk A.2 worden hier verder uitgewerkt.

A.5.1 Documentatie

In sectie A.2.6 staan de deliverables die het bedrijf van ons verwacht. De richtlijnen voor deze documenten worden gegeven door IEEE standaarden. Verder zal er bij elke methode en klasse documentatie geschreven worden.

A.5.2 Versiebeheer

Voor versiebeheer zullen we gebruik maken van SVN.

A.5.3 Evaluatie

Op de laatste werkdag van elke week, vanaf week 3, brengen we een werkende testversie uit. Soffos kan hier vervolgens commentaar op geven.

A.5.4 De pilots

Elke week zetten we een versie op de server en testen we alle onderdelen.

Bijlage B

Functional requirements

We have categorized the functional requirements according to the MoSCoW model.

Must have

1. All actors can login with their username and password.
2. All actors have zero, one or more of three roles: Admin, user and client.
3. Each person in database has a unique identifier: their username.
4. The system shall be modular and extensible.
5. The system shall have support for SSL connections.
6. The system shall use UTF-8.
7. The system shall check the input (is it really an e-mail address, etc).
8. Projects have extensions.
9. Extensions have questions from the database.
10. Extension questions can be added to the database.
11. Clients can fill out questionnaires.
12. Clients are not required to fill out personal data for each questionnaire. Personal data is provided once for all questionnaires.
13. Personal data is tied to a client-account.
14. Clients can have an arbitrary amount of delay between filling out questions.
15. Users can fill out extensions.
16. Users can create a questionnaire instance and link it with one of his/her client, also creation an extension instance for him/herself.
17. Admins can add an extension to a project.
18. Admins can add projects.

19. Admins can publish projects.
20. Admins can add users to a project.
21. Admins can create a questionnaire or extension.
22. Admins can create a question.
23. Admins can add questions to questionnaire or extension.
24. Admins can publish questionnaire.
25. Users can create their own accounts.
26. Actors can confirm their email address.

Should have

1. Users can add clients to a group.
2. Users can add groups.
3. Users can remove groups.
4. Users can remove clients from a group.
5. Admins can remove an users from a project.
6. Users can impersonate a specific client account of one of their own clients.
7. A questionnaire instance has a name, eg T0 or T1.
8. Time slots can be changed.
9. Questionnaires have a configuration flag indicating whether it is a private questionnaire.
10. Private questionnaires do not appear in the general questionnaire overview on the website.
11. All actors can change their password.
12. All actors can change their username.
13. Actors can retrieve their username if and only if their e-mailaddress is unique.
14. Categories can be calculated from the subquestions, or provided by the user.
15. Extensions can have 0..* (zero, one or many) categories.
16. Questions can have a category.
17. Categories can have a result calculated from subquestions.
18. A flag on the client account indicates whether the client has changed his/her username for first time use.
19. Clients can be in 0..* (zero, one or many) groups.
20. Admins can remove a questionnaire from a project.

21. Clients can change their personal data.
22. Admins can impersonate any specific user.
23. Admins can remove an extension from a project.
24. Admins can delete projects.
25. Actors can add roles lower than their highest role.
26. Actors can remove roles lower than their lowest role.
27. New languages can be added to the system.
28. The text of a question can be editted.

Could have

1. Users can rename groups.
2. Users can opt-in on reminders. Renminders saying “you need to open a new questionnaire for a clientäre settable per client, per project, per questionaire.
3. Reports can be arbitrarily complex.
4. Reports can be generated automatically when a client filled out a questionaire (e.g. T0, T1 and T1-T0 report).
5. Reports can be generated automatically when a selection of clients have filled out their questionnaires (group report).
6. Reports can have diagrams based on data and calculated data.
7. Questionnaires can be duplicated, edited before use.
8. Questionnaires cannot be deleted, they can be hidden.
9. All actors can reset their password (in case they forgot their password).
10. The system shall send notifications at a specific time of day or week (chosen by user or client).
11. The system shall combine notifications into one e-mail if possible.
12. The system shall be able to reproduce al outgoing e-mails.
13. Per project per extension can be set whether categories can be set at once or must be calculated from subquestions.
14. Clients can opt-in on a questionnaire reminder by e-mail: IJou have *x amount of time* left to fill out the questionnaire.”
15. A user can answer a category at once, or answer each question inside it.
16. Admins can edit projects.
17. Categories have simple calculations based on their subquestions.
18. Category calculations may differ for each category.

Would like

1. Reports can cover a single questionaire.
2. Reports can differ per questionaire.
3. Reports can have fields with a (arbitrarily complex) calculated value.
4. Results of the extensions are not in the same report as the questionnaire results.
5. Reports can be asociated with an extension, or a questionnaire, but never both.
6. Reports can cover more than one result (a group, or a selection) and they can cover multiple (same or different) questionnaires for each client.
7. There is a configuration option for each project specifying whether reports are generated automaticcaly.
8. All automatically generated reports are sent to the user by e-mail.
9. Reports shall be retievable from the system as a PDF download.
10. The system shall be able to export a selection of data in CSV format.
11. The system shall be able to export data in CSV format, formatted to match a predefined format.
12. Bills can be generated.

Bijlage C

Requirements implementation

Deze bijlage is verwijderd in verband met vertrouwelijkheid.

Bijlage D

Orientationreport

D.1 Introduction

In this project the back-end of a portal for medical questoinnaires is created. This back-end will also be called the server application. The server on which the application will run, imposes some restrictions on the server application. For example, it must be able to use MySQL as database. Last but not least, we require certain libraries or frameworks for specific functionality, such as dependency injection, unit testing and documentation generation. In this orientation report we shall explore options for the libraries which can perform these functions, by specifying our requirements, discussing each of the options and drawing a conclusion from that.

In this report we shall explore the conditions under which the project will take place. First, in chapter D.2, the SOLID principles are applied to the Python programming language. SOLID consists of a group of five programming principles. Each principle will be discussed. In chapter D.3 the back-to-front communication is discussed. Several approaches are discussed, after which a conclusion is made that states which approach will be used throughout the project. Chapter D.4 discusses the possible libraries that can be used for the back-to-front communication. In the following chapters, Python libraries for several purposes are discussed. These libraries are needed for MySQL, unit testing, documentation and inversion of control.

D.2 SOLID programming in Python

In 2002, Robert C. Martin (colloquially known as *Uncle Bob*) defined the principles of Object Oriented Design in his book *Agile Software Development, Principles, Patterns, and Practices*[7], and later on his personal in an article[8]. These principles are generally referred to as the SOLID principles, an acronym which stands for:

1. Single responsibility principle
2. Open/closed principle
3. Liskov substitution principle
4. Interface segregation principle
5. Dependency inversion principle

These SOLID principles have been discussed in courses about software engineering, therefore we have not extensively researched other options. Because these principles are part of a strategy to develop better software more effectively, we would like to apply these principles to our server-application project. Applying the SOLID principles will lead to easily extensible and maintainable software. In this chapter, we will investigate the implications of each of the principles for our software, which will be written in Python. We will compare the possible solutions

D.2.1 Single responsibility principle

Martin defines the Single responsibility principle as follows: *there should never be more than one reason for a class to change.* The responsibilities of a class are the requirements that will make the class chamge. The idea here is that when a class has more than one responsibility, there are several possible changes in requirements which could cause the class to change. These responsibilities in one class may then become entangled, which could make it difficult for a class to meet the other requirements if one requirement changes.[9]

To get this principle right in Python (or any other object-oriented programming language), for any two possible reasons for change which may or may not be combined, we have to ask the following question: *is there a reason which could warrant a change at different times?* When such a reason can be found, the two reasons for change should be separated as they are different responsibilities. When such a reason cannot be found, the two reasons for change are assumed to always change together, so they are a single resposibility and they need not be separated.

When reasons such as the details of the hardware or operating system force us to combine responsibilities in one class which should not be combined, we can separate the responsibilities in separate interfaces, both implemented by that one class. As seen from the rest of the application, the responsibilities are separated.[9]

D.2.2 Open/closed principle

The open/closed principle is defined by Martin as follows: *software entities (such as classes, modules, functions, etc.) should be open for extension but closed for modification.* The open/closed principle deals with abstractions: interfaces and abstract methods are abstractions available in object-oriented languages, which can be used by the code. When the code deals only with abstractions, it is possible to provide an alternate implementation by extension (a class implementing an interface, or a subclass implementing an abstract method), without modifying the code which uses these abstractions.[9]

Python is a strongly typed, object-oriented language, but it does not support the concept of interfaces or abstract methods. Whereas static languages rely on inheritance to support polymorphism, dynamic languages like Python depend on duck-typing (*if it quacks like a duck, it is a duck, even if you don't explicitly say it is*). In Python, by default all classes can be subclassed, and all methods can be overridden. Python's duck-typing removes the need for interfaces and base classes as contracts describing the methods you expect an implementation to have.

In a static language, you would provide an extension hook by providing an abstract or virtual method for subclasses to override. For Python we should put this static language mentality aside and start thinking the dynamic way: provide extension hooks by expecting a function to be provided, and use that function. For example, Python's `sorter` expects a function (which, in dynamic languages, can be a lambda function) to provide the comparison, like this: `sorted(numbers, lambda n1, n2 : n2 - n1)`. The `sorter` function is open for extension, but not modified.[10]

It is often the case that future changes cannot be properly anticipated, so that prematurely creating extension hooks (and the appropriate abstractions) may prove to be wasted work. The extension hooks are not used, and the added abstractions make the code unnecessarily complex. Instead, Martin argues, it is much better to leave the extension hooks out until a change is required. To make the change possible, an extension hook should be implemented. These changes should occur as often as possible, as early as possible, because changes later in the process are cumbersome. Late changes may require the adaptation of many parts of the project. Early discovering of changes can be achieved by writing unit tests, having short development cycles, developing the most important features first, and releasing the software early and often. These actions are also part of the Agile development methodology we use.[9]

D.2.3 Liskov substitution principle

The Liskov substitution principle can be very elaborately described, but is often summarized for static languages as follows: *subtypes must be substitutable for their base types.* Completely in line with Barbara Liskov's original definition, where an object of type T is expected, an object of type S may be substituted for it when S is a subtype of T. The methods of the base class or interface become part of the type's explicit contract: it is guaranteed by the language that type S has a method m when type T has a method m, with the same signatures.[9]

However, the Liskov substitution principle is generally taken to mean: *let $q(x)$ be a property provable about objects x of type T. Then $q(y)$ should be true for objects y of type S where S is a subtype of T*, where a property is anything from the accepted input arguments to the exceptions it may throw. This broader definition applies to the contract a class and its methods have to fulfill. For example, when a method has the property that it will not throw an exception, the overriding method in a subclass must also adhere to that, otherwise it violates the Liskov substitution principle. This distinction shows that static type checking with inheritance, along with checked exceptions and code contracts, are merely tools to assist the programmer in writing code which follows the Liskov substitution principle. The fact that Python does not have static type checking, or code contracts or checked exceptions for that matter, just means that we would have to be extra careful to adhere to the Liskov substitution principle.[12]

In static languages, the class in which a method is defined tells us something about the usage of the method in that particular class context. For example, the `draw` method of a `Shape` is expected to draw the shape, while the `draw` method of a `CardDeck` is expected to draw a card from the deck. But the class context is not important in dynamic languages: the expected methods can literally be defined in *any* class, even classes which are not even related. To get any protection against Liskov substitution principble violations in dynamic languages, it was suggested by Michael Feathers on [3] that *every message passed in a system should mean the same thing to all of its callers.* This means for our Python application that all `draw` methods ever implemented on any class are expected to always draw an object (or, equivalently, to always draw a card form a deck), and that no two methods with the same name but different behaviors may exist.

So, for static languages a *class* should substitutable for any other class which has this class as a super type in its inhertiance hierarchy. For dynamic languages, a *method* should be substitutable for any other method which has the same name and number of arguments.[13]

D.2.4 Interface segregation principle

The interface seregation principle states that *clients should not be forced to depend on methods that they do not use.* Abiding to this principle avoids the situation where a client (user of a class)

forces a change on a class, the other clients of that class are affected.[9]

It is still debated whether the interface segregation principle is still valid in Python. The reason is that Python has duck-typing and therefore the code only deals with those methods it wants to deal with. This implicit contract which is imposed on the arguments of functions and the values of fields is therefore as small as possible, containing only those methods which are used. The interface segregation principle is therefore irrelevant in dynamic languages.[10, 1]

D.2.5 Dependency inversion principle

According to the dependency inversion principle, *both high-level and low-level modules should not depend on each other, but on abstractions.* Additionally, those abstractions should not depend on the details (the implementation), but the details should depend on the abstraction. Intuitively, this principle feels correct, and it also applies to Python applications. Together with the Liskov Substitution Principle, these two principles form a strong team, where any object can be substituted by another in only a few places, and it all works.[9]

There is nothing preventing the use of dependency inversion in Python. With some help of third-party libraries, the programmer's life can be made easier and the program will be a lot more maintainable.

D.3 Back-to-front communication

The Medical Questionaires server-application is a back-end application which will be communicating with a front-end application, for example a website. To facilitate this communication, we have to choose a protocol and architecture. There are several web service styles we can choose from, including:

- Remote Procedure Call/Method Invocation (RPC/RMI)
- Simple Object Access Protocol (SOAP)
- Representational State Transfer (REST)

We will briefly discuss each of these technologies.

D.3.1 Remote Procedure Call/Method Invocation

Remote Procedure Call (RPC) was first defined in 1976 in RFC707[14] and uses operations as the basic unit of communication. It is the oldest widely adopted technology for inter-process communication available, and has been reimplemented in various ways. These include the Common Object Request Broker Architecture (CORBA), the Distributed Component Object Model (DCOM), and the object-oriented variant Remote Method +Invocation, popularized by Java. The main criticism on RPC is the tight coupling between the required functionality and the language-specific functions and methods.

D.3.2 Simple Object Access Protocol

The Simple Object Access Protocol (SOAP) is the successor of XML-RPC (Remote Procedure Call over XML). It communicates using XML messages, and there is a tight coupling of operations. These XML messages usually have a large overhead. According to [15], this tight coupling of operations can be tested and debugged before the application is deployed. Opponents of

SOAP argue that it does not behave like it is part of the web, whereas proponents counter that argument by stating that SOAP is made for higher-level languages.[15]

When working with Python, we found that a big disadvantage of SOAP is its strongly typedlyness. Python uses duck-typing for everything but this does not play well with SOAP. Another disadvantage is that seems to be hard to separate the various responsibilities in multiple classes. Usually a single service class is used as an interface to the system. There is also the problem of communicating the Python objects over SOAP to the client who uses a different language.

D.3.3 Representational State Transfer

A much more recent development is Representational State Transfer (REST). It works with resources (identified by URLs) and restricts the operations on the resources to the well-known HTTP operations such as GET, POST and DELETE. Contrary to SOAP, the responses of a REST web service can be anything: from XML, HTML, PDF to JSON. The overhead can be low, especially when using JSON. REST is easily scalable and provides lightweight access to its operations due to the limited number of operations.[15]

The responsibilities for the resources can be divided into multiple classes, one for each type of resource. A disadvantage of using REST is that it is mainly suitable for simple atomic operations on resources. This means that complex, multi-step or transactional web services are much harder to write using REST. Another disadvantage is that the data is not clearly defined, so that the consumer has to interpret the web server's response, and this may lead to ambiguities and errors. [5]

D.3.4 Conclusion

RPC is very outdated and not suitable for the big application we are designing. Both RPC and SOAP have the disadvantage that it would force us to put all service methods in one big class, which increases complexity, reduces modularity and results in bad separation of responsibilities. This is not a problem with REST, because we can separate the responsibilities for each resource in separate classes. REST is more scalable and lightweight than SOAP. When we use REST in combination with JSON, the overhead of the data will be small and almost all languages have (third-party) functionality to create and process JSON. The disadvantage that the REST service response is not well defined and open to interpretation can be mitigated by providing a detailed unambiguous specification of the request and response data. The point that a REST service is mainly suitable for simple atomic operations on resources is not really a disadvantage, because we expect that our application has to perform simple atomic operations on resources anyway.

D.4 REST framework for Python

As we have chosen to use REST, we need to decide which framework to use. We have the following requirements:

1. Active development to ensure that the framework has fewer bugs and works with the latest versions of other software,.
2. Easy to setup and use for non-expert Python programmers.
3. Low cost, or preferably free.
4. Good support through wiki, recepies, help, forum.

5. Easy to create RESTful applications.

6. Not bloated with features we will never use.

When an application is RESTful, it is created and designed properly in the spirit of REST: it uses the existing HTTP features as they should be used. In the discussion on [1] several possbile REST frameworks for Python were named (also found in [2]), together with their advantages and disadvantages. The following are the most popular frameworks, which would most likely conform to our 'active development' requirement:

- CherryPy
  http://www.cherrypy.org/
- Django
  http://www.djangoproject.com/
- web.py
  http://webpy.org/

D.4.1 CherryPy

The latest CherryPy release was in february 2011, so it is very recent. It is free, open source, and actively developed, judging from the lastest changes in the source files. It has integrated support for builing RESTful applications, as can be seen in [3]. It also has build-in support for the *Secure Socket Layer* (SSL) protocol([4]), which is most likely the protcol which we will have to use to securely transfer login credentials and other condifential data to and from the server application.

D.4.2 Django

Django is also a free open-source project, and the latest release was in march 2011. It has an active community which answers questions posted on various forums ([5], [6], [7]). However, it does not have native support for REST, and needs third-party libraries to work RESTful ([8]).

Django is a very complete web framework, but also more of a modular plug-and-play type of framework. It has numerous build-in components which we will not use in our server application, and instead of making the back-end creation easier, is more focussed on creating websites by *clicking* (custom-made) components together easily.

D.4.3 web.py

Web.py is an older open-source REST framework, whose latest release dates back to march 2010. It was recommended on several places [**?**, **?**] but those recommendations were made several years ago. It is currently an inactive, poorly supported project with a dormant community. While web.py works intuitively, the lack of development would make this library a dangerous choice

[1] http://stackoverflow.com/questions/713847 SOF713847
[2] http://wiki.python.org/moin/WebFrameworks
[3] http://docs.cherrypy.org/dev/progguide/REST.html
[4] http://webpy.org/cookbook/ssl
[5] http://stackoverflow.com/questions/tagged/django
[6] http://old.nabble.com/Django-f16208.html
[7] http://thedjangoforum.com/
[8] http://code.google.com/p/django-rest-interface/

for this project. A single change in the Python base class library or any of its dependencies can cause web.py to malfunction, which would then have to be solved without any support from the developers or community. Therefore is web.py not the framework we will use.

D.4.4 Conclusion

The choice is between Djando and CherryPy, and both have active communities and very much the same features. However, due to the fact that we will make a RESTful server application, CherryPy is the library which we will use because of its native REST support and the fact that Django has all sorts of features which we do not need.

D.5 MySQL library for Python

One of the requirements of the project was the use of MySQL for accessing a database, because the server runs a MySQL database. The library we want to use should contain all vital functions that can be used on a MySQL database, for example:

- Execute a query
- Set character set
- Fetching a single or multiple rows
- Execute several INSERT statements simultaneously
- Ability to search
- Retrieve the number of rows or fields
- Getting the last inserted id

These functions are the requirements for the library. There is a standard mysql-python package available to use, but it doesn't meet all the requirements stated above. There is no ability to, for example, execute a query and fetch the results at once. To meet the requirements, we found a thin Python wrapper around the mysql-python library named *MySQLdb*. The advantage of this library is that it enables the extra functionality to meet the requirements. The library also meets the database API specifications stated in PEP-249 [6]. There were no other available packages so the application will use MySQLdb to communicate with the provided database.

D.6 Unit test framework for Python

Because we want to test our application intensively we need a solid unit testing framework for python. After a search on the internet, we found a good unittest framework named pyUnit. Since python version 2.1 this unit testing package was build in the standard library.[4] Since pyUnit is based on JUnit for java and we have used JUnit in previous projects, we decided to use pyUnit as testing framework for our application.

D.7 Documentation library for Python

Because we are working in a team and creating a application which is extensible, it is important to create documentation for classes and methods we create. After a quick search for documentation library we found the built-in python library pyDoc. PyDoc is easy to use and it is not needed to make special modifications in the code to use it. PyDoc is able to create a local server to view webpages on, and can store HTML file to a specific directory. This way other developers can easily check documentation for implemented features.

[BRON]

D.8 Inversion of Control framework for Python

Because we have divided our application into multiple modules, we need an *Inversion of Control* library for Python. This library will help is in obeying to the dependency inversion principle discussed in D.2.5. Our requirements are similar to the requirements for the back-to-front communication. Our requirements are:

1. Active development to ensure that the library has fewer bugs and works with the latest versions of other software,.
2. Easy to setup and use for non-expert Python programmers.
3. Low cost, or preferably free.
4. Good support through wiki, recepies, help, forum.
5. Not bloated with features we will never use.

In a discussion on [11] the respondents name a number of dependency injection solutions for Python:

- snake-guice
  http://code.google.com/p/snake-guice/
- pyContainer
  http://pypi.python.org/pypi/PyContainer/
- python-inject
  http://code.google.com/p/python-inject/
- Spring Python
  http://springpython.webfactional.com/

D.8.1 snake-guice

Google designed the award-winning open-source Guice dependendy injection framework for Java, and snake-guice is a Python derivative of Google's framework. Where traditional dependency injection frameworks use XML for configuration, Guice uses the much more logical annotations on language structures for this. It is free, but is not actively developed and not suitable for production use. The last release of snake-guice was in 2009, and is classified as a development release. Therefore, this library is not suitable for our application.

D.8.2 pyContainer

pyContainer is a free lightweight Python dependency injection container. It could have been a plausible alternative if the library where not in the alpha stage with the last active development occuring in july 2009. Therefore, on the same grounds as snake-guice, this library is not suitable for us.

D.8.3 python-inject

python-inject claims to provide a *pythonic* way of dependency injection. The project is not very active, according to Google Project Hosting. It is free, and the second and last release was version 1.0.1 in july 2010. The code examples shown on the site show a reasonable syntax and usage of dependency injection. However, because of the lack of activity and the fact that the 1.0.1 release appears to be just the second release ever, it seems that python-inject can only be considered when there is no better alternative.

D.8.4 Spring Python

Spring Python is a python version of Java's Spring framework, and this heritage is a point of criticism from some Python developers.[2] The last build of most parts of the framework occured less than 12 hours ago at the time of writing. The framework is free, and not difficult to set up. We did a quick test and it took a few minutes to wrap our heads around how this framework exposes dependency injection, but once we got that straight, it seemed logical. A quick search on the internet reveals that there are several forums where users ask questions about Spring Python and the developer's website features a Spring Python book. Apart from Inversion of Control, Spring Python has additional features such as Aspect Oriented Programming, security interceptors and JMS messaging. These are features that we will not use.

D.8.5 Conclusion

snake-guice, python-inject and pyContainer all have the problem that their latest versions are more than nine months old, and there appear to be no active developers for these projects. Spring Python conforms to all our requirements, except one: it has many more features than we want to use. However, since Spring Python is the only feasible choice for a dependency injection framework, and the fact that the features that we will not use do not have to be imported, this should not have to be an issue.

Bibliografie

[1] Julian Birch. Dynamic languages and solid principles, 11 2009.

[2] Salim Fadhley. What's the best online tutorial for starting with spring python, 3 2009.

[3] Michael Feathers. Liskov substitution in dynamic languages, 8 2006.

[4] Python Software Foundation. 25.3. unittest unit testing framework, 4 2001.

[5] Ryan Heaton. Leave the rest to soap, 1 2007.

[6] Marc-André Lemburg. Python database api specification v2.0, 3 2008.

[7] Robert C. Martin. *Agile Software Development, Principles, Patterns, and Practices.* Prentice-Hall, Inc, 2002.

[8] Robert C. Martin. The principles of ood, 5 2005.

[9] Robert C. Martin and Micah Martin. *Agile Principles, Patterns, and Practices in C#.* Prentice Hall PTR, 1 edition, 7 2006.

[10] Dhananjay Nene. How duck typing influences class design and design principles, 9 2008.

[11] Mark Roddy. Python dependency injection framework, 10 2008.

[12] Stefan Roock. The liskov substitution principle (lsp) in duck typed programming languages, 11 2010.

[13] Dean Wampler. The liskov substitution principle for "duck-typed"languages, 9 2008.

[14] James E. White. A high-level framework for network-based resource sharing, 1 1976.

[15] Michael zur Muehlen, Jeffrey V. Nickerson, and Keith D. Swenson. Developing web services choreography standards–the case of rest vs. soap. *Decision Support Systems*, 40(1):9 – 29, 2005.

Bijlage E

Communication specification

Deze bijlage is verwijderd in verband met vertrouwelijkheid.

Bijlage F

Mission spectrum

The following document was written by Soffos at the start of the project.

F.1 About the company

The mandate of the company is to serve projects that use questionnaires. A project is requested by a company, after which is determined which questionnaires are needed. Users can then register for this project and register their clients. This will enable the clients to fill in a questionnaire, which will be stored in the database. With this data, reports are generated for the individual user, or an article is written which can then be published.

F.2 Current Situation

In the current situation, the website www.soffos.nl handles both the information about the different researches and operates as a portal for the different questionnaires. This site, however, was build a couple of years ago and cannot handle the large number of different requests anymore. As the field of research has expanded, a new system is required which will function as the portal for the questionnaires.

F.3 Problem

In the new situation, the system has to be accessible from different platforms. Not only the site will need to have access to it, but also an administrative program and other programs that will be designed in the future. Furthermore, it has to be possible to easily extend the system with multiple new questionnaires and extensions for the clients.

From this point of view, the system has two main requirements:

- The system has to be easily accessible, for instance with RPC or RMI and
- The questionnaires and extensions in the system need to be extendible by an admin with no knowledge of Computer Science.

The project team working on this assignment will work together with a project team working on the initial website on engineering the system.