Internal Report 2012-15 August 2012

Universiteit Leiden

Opleiding Informatica

Using Software Design Principles
For Software Systems

Fee Yun Wong

BACHELOR THESIS

Leiden Institute of Advanced Computer Science (LIACS)
Leiden University
Niels Bohrweg 1
2333 CA Leiden
The Netherlands

Using Software Design Principles For Software Systems

Fee Yun Wong

31 August 2012

Supervisors: Michel Chaudron
Dave Stikkolorum

Hoofdstuk 1 Inleiding

De bedoeling van dit Bachelorproject is het gebruiken van cassussen en daarvan wordt een analyse gedaan van het probleem. Een aantal verschillende software modellen wordt gerepresenteerd voor eenzelfde casus en door het vergelijken van deze modellen worden vragen bedacht. Deze vragenlijst is bedoeld voor het testen van de kennis van studenten die geen of weinig ervaring hebben met software design. Verder sluit dit project aan op het afstudeeronderzoek van Oswald en promotieonderzoek van Dave Stikkolorum.

In hoofdstuk 2 wordt uitgelegd wat software design is, waarom het belangrijk is en wat de rol is van design principles in het software design proces. Hoofdzakelijk wordt er aandacht besteed aan de design principles coupling, cohesion en information hiding.

In hoofdstuk 3 staan de casussen beschreven, de software modellen en de betreffende vragen worden gerepresenteerd. De eerste casus bevat een fictieve bedrijfssituatie van een kartbaan genaamd "Race-Planet-Delft" gemaakt door Dave Stikkolorum. De tweede casus gaat over een robotisch zeilboot "Sailbot" en de derde casus bevat de zorgvraag.
Verder staan mijn resultaten en bevindingen.

In hoofdstuk 4 staat mijn reflectie en wat er veranderd en verbeterd kan worden. Wat mijn leertraject is geweest.

Hoofdstuk 2 Software Design

Software design is de kern van het software engineering proces. Voor software engineers is het belangrijk om te weten en te begrijpen welke basis design principles nodig zijn voor het bereiken van software met hoge kwaliteit en welke design methodes stellen de engineers in staat deze principles te implementeren.
Een goed design hoort aan de volgende eisen te voldoen: het is in overeenstemming met de klantenwensen; het is foutloos en betrouwbaar; relatief makkelijk te veranderen; het wordt op een manier in modules gezet waardoor reuse mogelijk is.
Om tot een goed design te komen van een computer programma is door het bekijken van resultaten van slecht ontworpen programma.
Toch concentreren software professionals teveel op één element van de design, terwijl andere gebieden die mogelijk zelfs belangrijker zijn, worden verwaarloosd.

Design kan als volgt gedefinieerd worden: het proces van het toepassen van verscheidene technieken en principles voor het doel van definiëren van een device, een proces of een systeem in voldoende detail zodat het fysieke realisatie toe te staan.

Aanpak van het proces voor het ontwikkelen van een model is een combinatie van: intuïtie en inzicht dat gebaseerd is op ervaring met het bouwen zelfde entiteiten; een verzameling van principles en/of heuristieken die leiden naar de manier hoe het model zich ontwikkelt; een verzameling van criteria waarmee de kwaliteit beoordeeld kan worden en een herhalingsproces is nodig dat uiteindelijk leidt tot een definitieve representatie.

Computer software design verandert constant, omdat er nieuwe methodes, betere analyse en breder begrip worden ontwikkeld.

Start met het analyseren en specificeren van software requirements, daarna komt de technische activiteiten- *software design, programmeren en testen*- die vereist zijn voor het bouwen en controleren van software. Elke activiteit verandert de informatie op een manier dat resulteert in een goedgekeurde computer software.

De stroom van informatie tijdens de technische fase van de software proces begint met software requirements, gemanifesteerd door informatie [data model], en gevolgd door de functional en behavioral models.
Maak gebruik van een van de design methodes, de design stap produceert een data design, een architectural design en een proceduraal design. De data design verandert de informatie domein model gecreëerd tijdens analyse naar datastructuren dat zal vereist worden om de software te implementeren. De architectural design definieert de relatie tussen belangrijke structucturele elementen in een procedural beschrijving van de software. Source code wordt gegenereerd, en testen leidt tot integratie en goedkeuring van software.

Design, programmeren en testen absorbeert 75 procent of meer van de kosten van software engineering (onderhoud niet inbegrepen). Hier maken we beslissingen die uiteindelijk het succes van software implementatie beïnvloed, en net zo belangrijk, het gemak dat een software onderhouden kan worden. Deze beslissingen worden tijdens de software design gedaan, dat maakt deze stap cruciaal in ontwikkelingsfase. Maar waarom is design zo belangrijk?

Het belangrijkste van software design is kwaliteit. Design is de plek waar kwaliteit wordt opgenomen in software development, voorziet ons van representaties van software dat de kwaliteit kan bepalen, het kan de klantenwensen vertalen naar een uiteindelijk software product of systeem. software design dient als een fundament voor alle software engineering en software onderhoud stappen die daarop volgen. Zonder design riskeren we een instabiele systeem – dat bij kleine veranderingen al faalt;

moeilijk te testen is; diens kwaliteit kan moeilijk beoordeeld worden tot in de late fase van het software engineering proces, als er weinig tijd is en veel geld al besteed is.[1]

The design process

Software design is een proces waardoor requirements worden omgezet in een representatie van een siftware. Aanvankelijk beeldt de representatie een holostisch voorstelling/zienswijze van een software af. Het verfijnen ervan leidt tot een design representatie dat erg lijkt op de source code.

Vanuit het perspectief van project management, wordt software design ledit tot twee stappen. Preliminary design en detail design. Binnen de context van preliminary en detail design vindt een aantal verschillende design activiteiten plaats. Bovenop de data, architectural en procedural design, hebben veel moderne applicaties een duidelijk interface design activiteit. Interface design stelt de layout en interactie mechanisme voor human-machine interactie vast.

Hieronder volgen de richtlijnen voor het evalueren van de kwaliteit vaneen design representatie:
1) een design hoort een hierarchisch organisatie die intelligent gebruik maakt van de controle tussen elementen van software, te tonen
2) een design moet modulair zijn; dat betekent dat de software logisch ingedeeld is in elementen dat specifieke functies en subfuncties uitvoert
3) een design moet een duidelijk en aparte representatie van data en procedures bevatten
4) een design zal moeten leiden tot modules dat onafhankelijke functionele karakteristieken vertoont.
5) Een design moet leiden tot interfaces dat de complexiteit van connecties tussen modules en met de externe omgeving verminderen.
6) Een desing meot afgeleid worden door het gebruik maken herhaalbare methodes bestuurd door verkregen informatie tijdens de software requirements analyse.

Deze richtlijnen zijn verkregen door de applicatie van fundamentele design principles, systematisch methodologie en nauwkeurig herziening van de software engineering desing process.

Software design heeft invloed op zowel de interface als de functie van een computer programma.

Een verzameling van fundamentele software design concepten heeft zich ontwikkeld in dde afgeloepn drie decennia. Elk concept voorziet een software designer van een fundament waarvan nog meer geavanceerde design methodes kunnen worden toegepast. Elk concept helpt de software engineer de volgende vragen te beantwoorden:
1) Welke criteria kunnen gebruikt worden voor het verdelen van de software in individuele componenten?
2) Hoe wordt een functie of data structuur detail gescheiden van een conceptueel reperesentatie van de software?
3) Zijn er constante criterai die de technische kwaliteit van een software design definiëren?

M. A. Jackson once said: “The beginning of the wisdom for a computer programmer[software engineer] is to recognize the difference between getting a program to work , and getting it *right.*” Fundamentele software design concepten voorziet the noodzakelijke kader voor “getting it right.”[1]

Abstraction

Wanneer we een modulair oplossing voor elk probleem overwegen, kan veel niveaus van abstractie gesteld worden. Op de hoogste niveau van abstractie wordt een oplossing aangegeven in brede termen gebruik makend van de taal van de probleem omgeving. Op lagere niveaus van abstractie is er meer procedural oriëntatie.

Probleemgeoriënteerd terminologie is gekoppeld met implementatiegeoriënteerd terminologie met poging tot een oplossing te komen. Tot slot, op de laagste niveau van abstractie, de oplossing wordt op een manier vermeld dat direct geïmplementeerd kan worden.

Elke stap in de software engineering proces is een verbetering/verfijning op het niveau van abstractie van de software oplossing. Tijdens system engineering is het software toegekend als een elemnt van een computer-based systeem. Als we van preliminary naar detail design gaan, wordt het niveau van abstractie verminderd. Tot slot wordt het laagste niveau van abstractie bereikt wanneer de source code wordt gegenereerd.
Zoals we door verschillende niveaus van abstractie gaan, werken we om procedural en data abstracties te creëren. Een procedural abstractie is een genoemde volgorde van instructies dat een specifieke en beperkte functie heeft. Een data abstractie is een genoemde collectie van data dat een data object beschrijft.

De concepten van *stepwise refinement* en *modularity* zijn nauw verbonden aan abstractie. Als de software design zich ontwikkelt, representeert elk niveaau van module in een programma structuur een verbetering/verfijning in het niveau van abstractie van de software.
Data abstractie, en ook procedural abstractie, maakt het voor een designer mogelijk een data object op verschillende niveaus van detail te representeren en nog belangrijker is het specificeren van een data object in de context van die operaties(procedures) dat toegepast kan worden.

Zodra een data abstractie is gedefinieerd dan is een verzameling van operaties die mogelijk kan worden toegepast ook gedefinieerd.
Control abstractie is de derde vorm van abstractie die gebruikt wordt in software design. Net zoals procedural en data abstractie, impliceert control abstractie een program control mechanisme zonder specificatie van interne details.

Refinement

Stepwise refinement is een vroegere top-down design strategie voorgesteld door Niklaus Wirth. De architectuur van een programma is ontwikkeld door het achtereenvolgens verfijnen van niveaus van procedural detail. Een hiërarchie wordt ontwikkeld door stapsgewijs ontleden van een macroscopische verklaring van een functie (een procedural abstractie) totdat programmeertaal verklaringen zijn bereikt.

Het proces van program verfijning voorgesteld door Wirth komt overeen met het proces van verfijning en verdeling die wordt gebruikt tijdens requirements analyse. Het verschil zit in het niveau van detail die wordt overwogen en niet de aanpak. Refinement is eigenlijk een proces van *uitwerking*.
Refinement veroorzaakt dat de designer de originele verklaring meer uitwerkt, voorzien van meer en meer details zoals elke opeenvolgende uitwerking zich voordoet.

Modularity

Het concept van modularity in computer software heeft al voor bijna vier decennia aanhang. Software achitectuur drukt modularity uit; dat betekent dat software verdeeld is in aparte genoemde en adresseerbare componenten, modules genoemd, die geïntegreerd zijn om aan probleem requirements te voldoen.
Er wordt verklaard dat "modularity is de enige attribuut van software die een programma toestaat intellectueel handelbaar te zijn. Monolithic software (i.e., een groot programma dat een enkele module omvat) kan niet makkelijk begrepen worden door een lezer.

Het is belangrijk om op te merken dat een systeem in modules ontworpen is, zelfs als een implementatie monolothic moet zijn.[1]

Software architecture

Software architecture duidt twee belangrijke kenmerken van een computer programma aan: (1) de hierarchische structuur van procedural componenten (modules) en (2) de structuur van data. Software architectuur is afgeleid door middel van een verdeelproces dat elementen van een software oplossing relateert met delen van een echte wereld probleem impliciet gedefinieerd tijdens requirement analysis. De ontwikkeling van software en data structuur begint met een probleem definitie. Oplossing komt voor als elk deel van het probleem wordt opgelost door één of meer *software* elementen.

Control hierarchy

Control hierarchy, ook wel *program structure* genoemd, representeert de organisatie (vaak hiërarchisch) van programma componenten (modules) en impliceert een hiërarchie van control. Het representeert niet de procedural aspecten van software zoals volgorde van processen, gebeurtenis/volgorde van beslissingen of herhaling van operaties.
De control hiërarchie representeert ook twee subtiel verschillende karakeristieken van de software architectuur: *visibility* en *connectivity*.
Visibility geeft de verzameling van programma componenten aan die opgeroepen of gebruikt kan worden als data door een gegeven component, zelfs wanneer indirect wordt bereikt.
Connectivity geeft de verzameling van componenten aan die indirect worden opgeroepen of gebruikt als data door een gegeven component.

Data structure

Data structuur is een representatie van de logische relatie tussen individuele elementen van data. Datastructuur is net zo belangrijk als een programma structuur voor de representatie van software architectuur, omdat de structuur van informatie zal onveranderlijk invloed hebben op de uiteindelijke procedural design.
De organisatie en complexiteit van een datatstructuur zijn alleen beperkt de vindingrijkheid van de designer. Hoewel er nog klassieke datastructuren aanwezig zijn.

Software procedure

Programma structuuur definieert control hiërarchie zonder rekening te houden met de volgorde van verwerking en beslissingen.
Software procedure concentreert zich individueel op de verwerking van details van elke module. Procedure moet voorzien zijn van een precieze specificatie van verwerking, inclusief volgorde van gebeurtenissen, exacte beslissingspunten, repetitieve operaties en zelfs data organisatie/structuur.

2.1 Wat is information hiding?

Het concept van modulariteit leidt voor elke software designer naar een fundamentele vraag: “Hoe ontleden we een software oplossing om de beste verzameling van modules te verkrijgen?”
Het principe van *information hiding*[5] duidt aan dat modules worden “gekarakteriseerd/gekenmerkt door design beslissingen die (elk) zich verborgen houdt voor alle andere design beslissingen. M.a.w. modules worden gespecificeerd en ontworpen zo dat informatie(procedure en data) die een module bevat, ontoegankelijk zijn voor andere modules die deze informatie niet nodig hebben.
Hiding impliceert dat effectieve modulariteit bereikt kan worden door een verzameling van onafhankelijke modules te definiëren die met elkaar communiceren alleen voor de informatie die nodig is om tot een software functie te komen. Abstraction helpt hierbij het definiëren van de procdural (of informational) entiteiten die de software omvat. Hiding definieert en dwingt toegangsbeperkingen af bij procedural detail in een module en ieer lokale data structuur gebruikt door de module.
Het nut van information hiding als een design criteria voor modular systemen verschaft grootste voordelen wanneer wijzigingen vereist zijn tijdens testen en later tijdens software onderhoud. Omdat

de meeste data en procedure verborgen zijn voor andere delen van de software, onbedoelde errors geintroduceerd tijdens wijziging zijn minder geneigd zich te verspreiden naar andere locaties binnen de software.

Effective modular design

De design grondbeginselen beschreven in de voorgaande sectie dienen allemaal voor het aanzetten tot modular design. In werkelijkheid is modularity een geaccepteerde aanpak geworden in alle engineering disciplines. Een modular design vermindert complexiteit; vergemakkelijkt verandering (een kritieke aspect van software onderhoud), en resulteert in makkelijkere implementatie door aanmoediging van parallelle ontwikkeling van verschillende delen van een systeem.

Functional independence

Het concept functional independence is een directe resultaat van modularity en de concepten van abstraction en information hiding. Functional independence wordt bereikt door ontwikkeling van modules met een doelbewuste functie en een weerzin van buitengewone interactie met andere modules. Op een andere manier verklaard, betekent het dat we een design software willen zo dat elke module een specifieke subfunctie van requirements verwijst en heeft een simpel interface bekeken vanuit andere delen van de programma structuur.
Software met effective modularity, i.e. onafhankelijke modules, is belangrijk omdat het makkelijker te ontwikkelen is omdat functie kan worden gecompartimenteerd en interfaces zijn vereenvoudigd. (overweeg vertakkingen wanneer ontwikkeling wordt uitgevoerd door een team.) onafhankelijke modules zijn makkelijker te onderhouden (en te testen) omdat: secondaire effecten veroorzaakt door design/code wijziging zijn beperkt; error verspreiding wordt verminderd; reusable modules zijn mogelijk. Samenvattend, functional independence is de sleutel voor goed design, en design is de sleutel voor software van de relatieve kwaliteit.
Onafhankelijkheid wordt gemeten door het gebruik van twee kwalitatieve criteria; cohesion en coupling. Cohesion is een maatstaf van relatieve sterkte van een module. Coupling is een maatstaf voor onderlinge afhankelijkheid tussen modules.

2.2 Wat is coupling?

Coupling is een maatstaf van verbinding tussen modules in een software structuur. Coupling is afhankelijk van de interface complexiteit tussen modules, het punt bij welke ingang of verwijzing is gemaakt naar een module en welke data wordt doorgegeven aan de interface.
In software design streven we naar laagst mogelijke coupling. Simpel connectiviteit tussen modules resulteert in een software dat makkelijker te begrijpen is en minder gevoelig voor een “ripple effect”[6] veroorzaakt wanneer errors optreden bij een locatie en zich door een systeem verspreiden. Op gematigde levels is coupling gekenmerkt door passage van control tussen modules. Control coupling is erg gebruikelijk in meeste software designs. In zijn simpelste vorm, control wordt doorgegeven via een “flag” op welke beslissingen zijn gemaakt in een subordinate of superordinate module.
Relatief high levels of coupling komt voor wanneer modules zijn verbonden aan een omgeving oppervlakkig aan software. External coupling is essentieel, maar het moet beperkt worden tot een klein aantal modules binnen een structuur. High coupling komt ook voor wanneer een aantal modules refereert een globale data area. De hoogste graad van coupling, content coupling, komt voor wanneer één module gebruik maakt van data of control informatie onderhouden binnen de grenzen van een andere module. Ten tweede, content coupling komt voor wanneer vertakkingen zijn gemaakt in het midden van een module. Deze mode van coupling kan en zal worden vermeden.
De coupling modes die hierboven zijn behandeld omdat design beslissingen gemaakt wanneer structuur wordt ontwikkeld. Varianten van external coupling, hoewel, kan mogelijk geïntroduceerd worden tijdens programmeren.[1]

2.3 Wat is cohesion?

Cohesion is een natuurlijke extensie van information hiding concept. Een cohesive module voert een enkele taak uit binnen een software procedure, waarbij vereist is dat een beetje interactie met procedures worden uitgevoerd in andere delen van een programma. Een cohesive module moet alleen één ding doen (in het ideale geval).
Er bestaan verschillende soorten cohesion, coincidentally cohesion treedt op bij een module die een verzameling van taken uitvoert die los van elkaar zijn gerelateerd.
Logically cohesion is een module die taken die logisch gerelateerd zijn, uitvoert.
Temporal cohesion treedt op wanneer een module taken bevat die gerelateerd zijn door het feit dat ze worden uitgevoerd met dezelfde tijdspanne.
De combinatie van functies in één module verhoogt de risico op error verspreiding wanneer een wijziging wordt gemaakt bij één van de procestaken, dit gebeurt wanneer er sprake is van low cohesion.
Moderate levels of cohesion are relatively close to one another in the degree of module independence.
Procedural cohesion bestaat wanneer verwerkingselementen van een module zijn gerelateerd en noeten worden uitgevoerd in een specifieke volgorde. Communicational cohesion is aanwezig wanneer alle proceselementen zich concentreren op één gebied van een data structuur.
Hoge cohesion wordt gekenmerkt door een module die een apart procedural taak uitvoert.
Het is onnodig het precieze niveau van cohesion te bepalen. Het is eerder belangrijk te streven naar high cohesion en het herkennen van low cohesion, zodat software design veranderd kan worden om meer funtioneel onafhankelijkheid bereikt kan worden. [1]

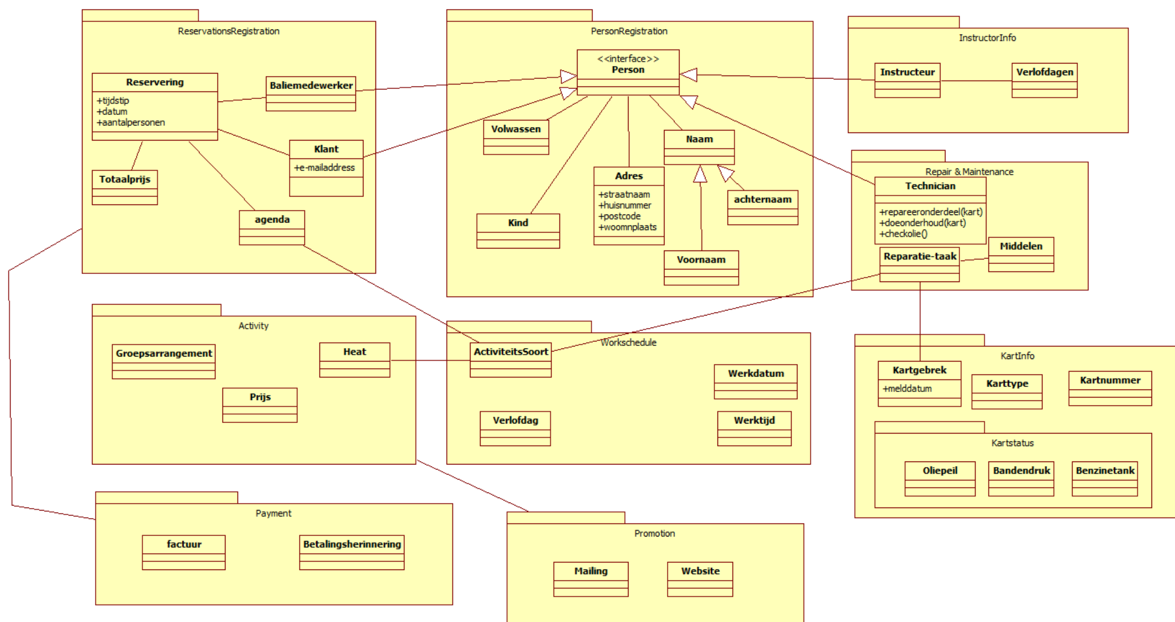
Hoofdstuk 3 Casussen

In dit hoofdstuk beschrijven we drie casusses die bestaan uit de volgende systemen: de kartbaansysteem, zeilbootsysteem en zorgvraagsysteem.

3.1 Casus 1: Kartbaan

De klant kan bij de kartbaan reserveringen maken voor een bepaalde datum, tijdstip en aantal personen. Hierbij gebruikt het personeel van Raceplanet het kartbaansyteem dat ontworpen is voor het personeel die de reserveringen kunnen registreren voor klanten en in het syteem nodige informatie zoeken voor de reservering. De gegevens van klanten, het personeel en de eigendommen van een kartbaan worden eveneens geregistreerd, ook deze kunnen verwijderd, gewijzigd en toegevoegd worden.
Om het bedrijf draaiende te houden is het daarnaast noodzakelijk dat de benodigdheden bruikbaar en beschikbaar zijn. De reparatie en onderhoud van karts moeten gepland worden.

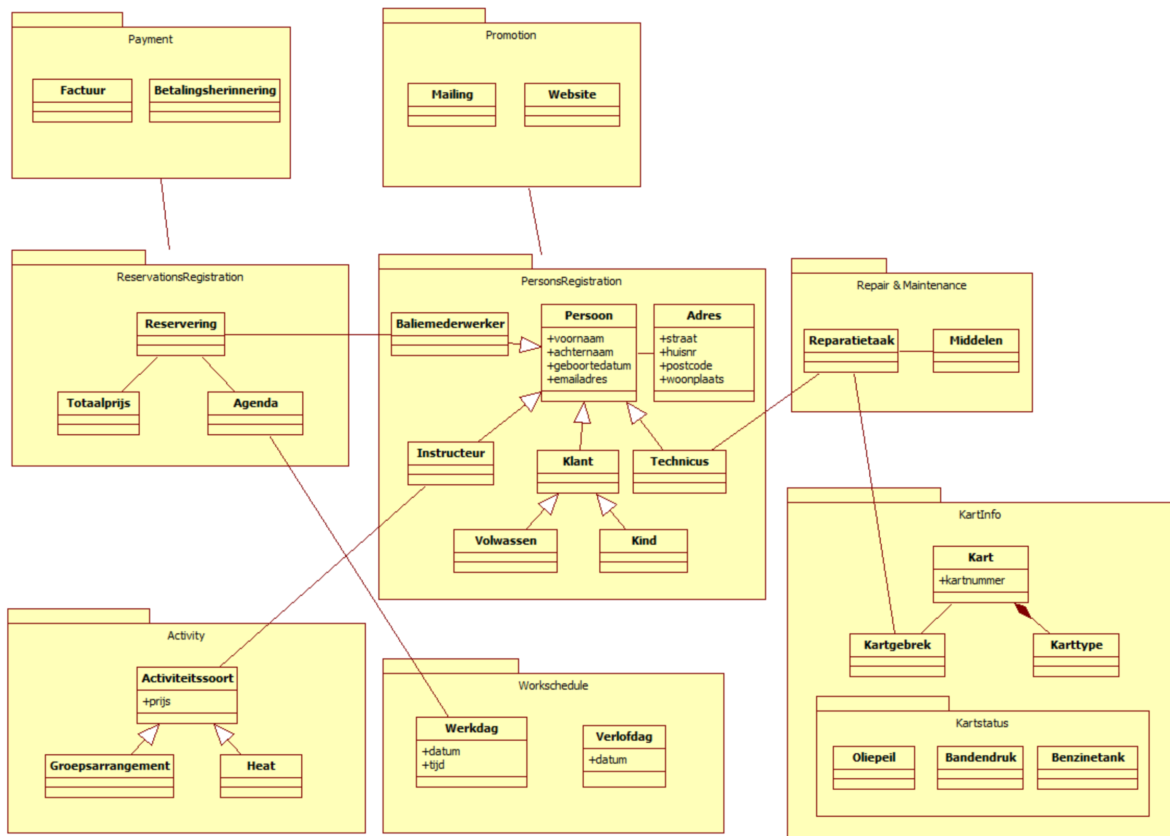
Belangrijk is het werven van en contact houden met klanten, reclame maken en goede service bieden aan klanten is eveneens van belang om regelmatig te doen. Zo is hiervoor o.a. een website opgezet voor het bieden van informatie van activiteiten die verwacht worden en deze worden ook via post of email verstuurd.[2]



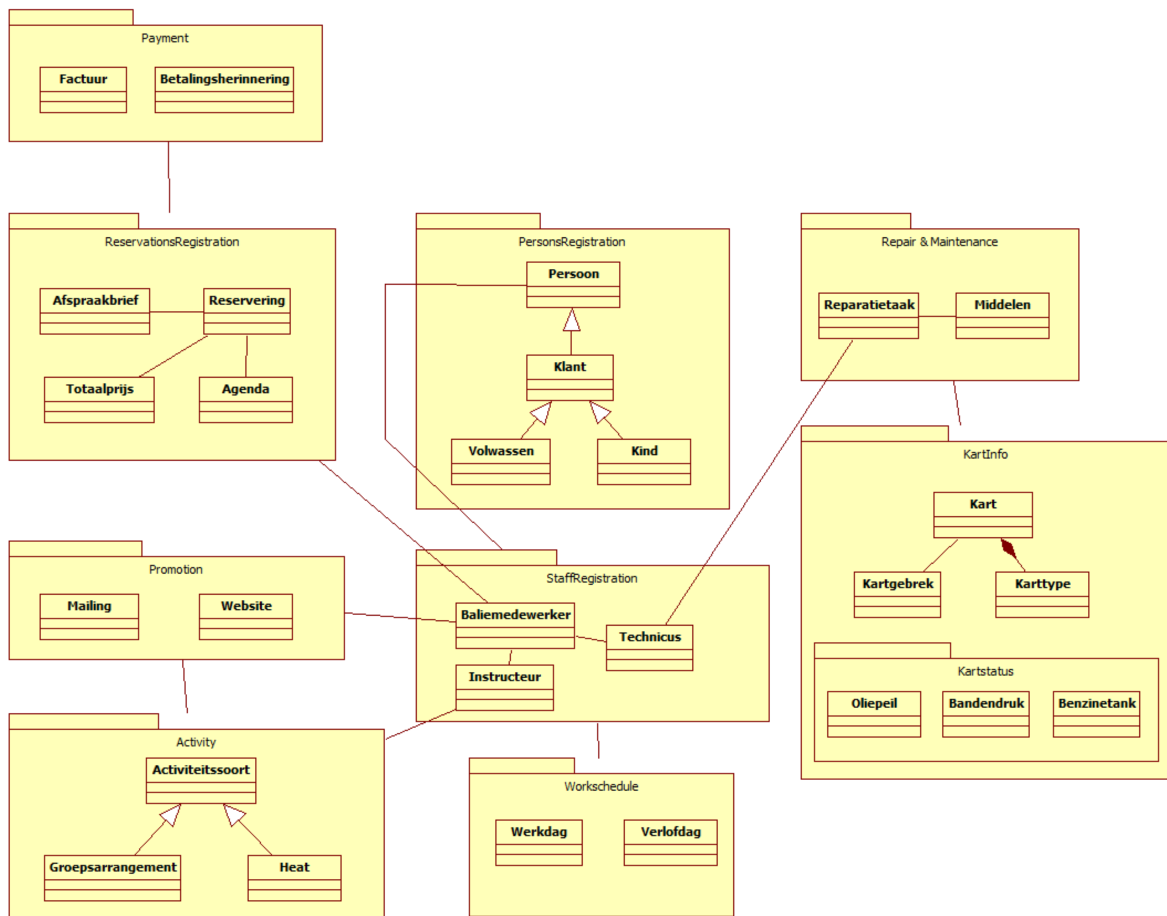
Figuur 1.1 Kartbaan model 1

Verantwoordelijkheden van modules
Module *ReservationsRegistration:* registratie van informatie en berekenen van de totale prijs.
Module *Personsregistration:* het registreren persoonlijke gegevens van klanten en personeel.
Module *InstructorInfo:* werkgegevens van instructeuren.
Module *Repair & maintenance:* planning van de reparatie en onderhoud.
Module *KartInfo:* informatie van de aanwezige karts in het bedrijf.
Module *WorkSchedule:* rooster voor personeel.
Module *Payment:* factuur voor klant.
Module *Activity:* informatie van activiteiten.

Module *Promotion:* acties en korting voor klanten.

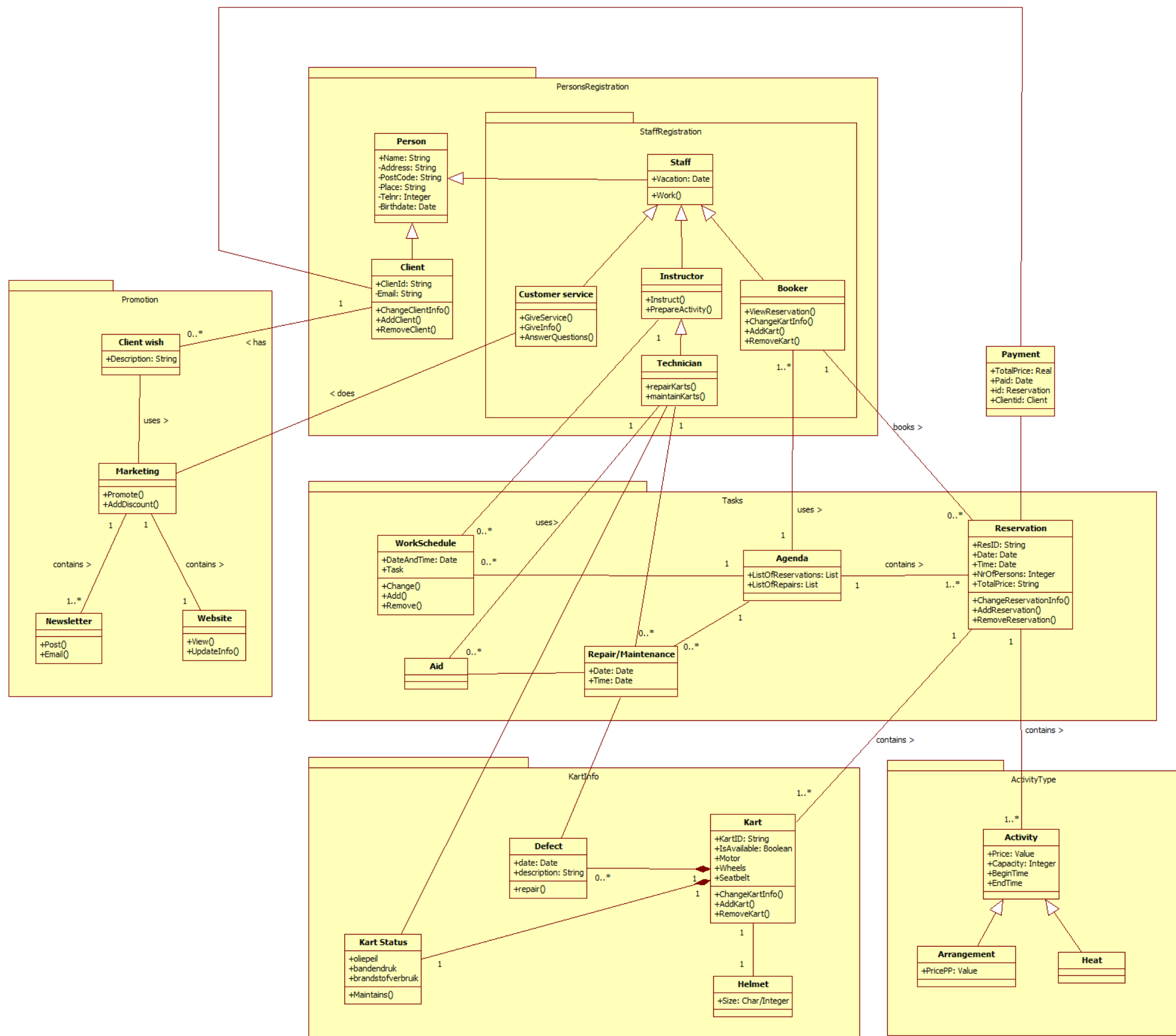


Figuur 1.2 Kartbaan model 2



Figuur 1.3 Kartbaan model 3

Module *StaffRegistration:* registratie van het personeel



Figuur 1.4 Kartbaan model 4

1) Welke ontwerp is beter: figuur 1.1 of figuur 1.4?

a) Figuur 1.1, want het gebruik van modules geeft toegang tot een overzicht van klassen die directe relatie met elkaar hebben de informatie die nodig is tussen verschillende modules.
b) Figuur 1.4, want de klassediagram geeft direct aan welke klassen directe invloed op elkaar hebben.
c) Figuur 1.1 zorgt voor dat de klassen die nauw met elkaar samenwerken in één module horen met een bepaalde verantwoordelijkheid.
d) Figuur 1.1, omdat er meer cohesion en minder coupling is.

Antwoord: *c*

2) In welke ontwerp staat de klasse activiteitssoort op de juiste plaats?

a) In figuur 1.1 waar de activiteitssoort in de package *Workschedule* staat, omdat het nodig voor de rooster welke activiteiten verzorgt/ voorbereid moeten worden door het personeel.
b) In figuur 1.2/1.3 waar de activiteitssoort in de module *Activity* staat, want het hoort een onderdeel te zijn van activiteiten.
c) Activiteitssoort is een abstracte klasse van *Activity* en is een generalisatie van andere klassen.
d) Zowel antwoord b en c is juist.

Antwoord: *d*

3) Welk ontwerp is het meest handig, als het gaat over verschillende personeel?
a) Elk soort personeel hoort in een package waar ook zijn/haar taken staan, zoals in figuur 1.1.
b) Er hoort een package *StaffRegistration* te bestaan verbonden met de packages met de verantwoordelijkheden van elk soort personeel, zoals in figuur 1.3.
c) De package *StaffRegistration* verplaatsen in de package *PersonsRegistration*, zodat er minder coupling ontstaat, zie figuur 1.4.
d) Geen van de antwoorden is juist.

Antwoord: *c*

4) In figuur 1.4 kan één klasse in een package met meerdere klassen verbonden zijn uit andere packages, terwijl sommige informatie niet nodig is voor één bepaalde functie. Hoe kan dit voorkomen worden?

a) Door het toepassen van information hiding is er sprake van minder coupling.

b) Alleen via ontworpen interface van de module kan er toegang verleend worden tot de nodige implementatie details voor een andere module.

c) Bij veranderingen van één module heeft geen effect op andere modules door information hiding.

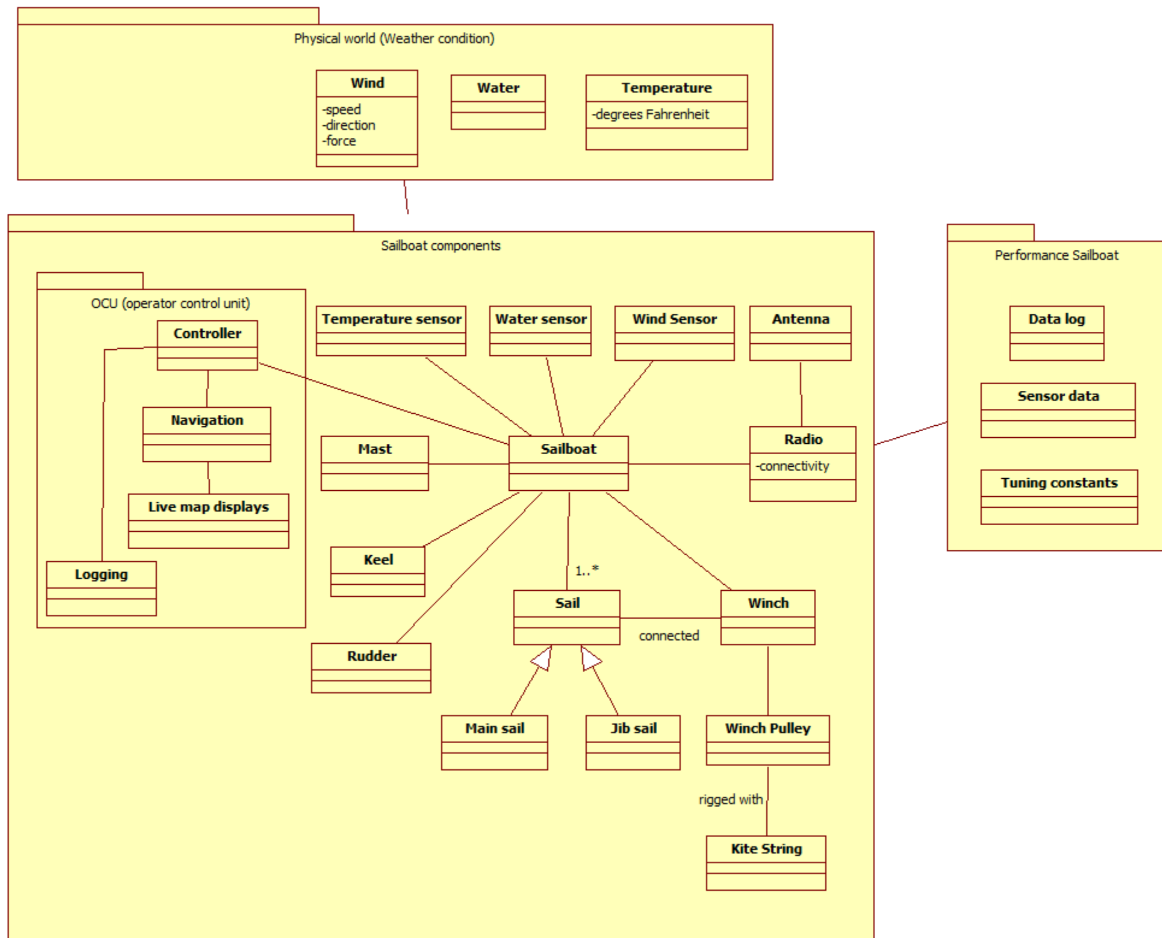
d) Alle bovenstaande antwoorden zijn juist.

Antwoord: *d*

3.2 Casus 2: Zeilboot

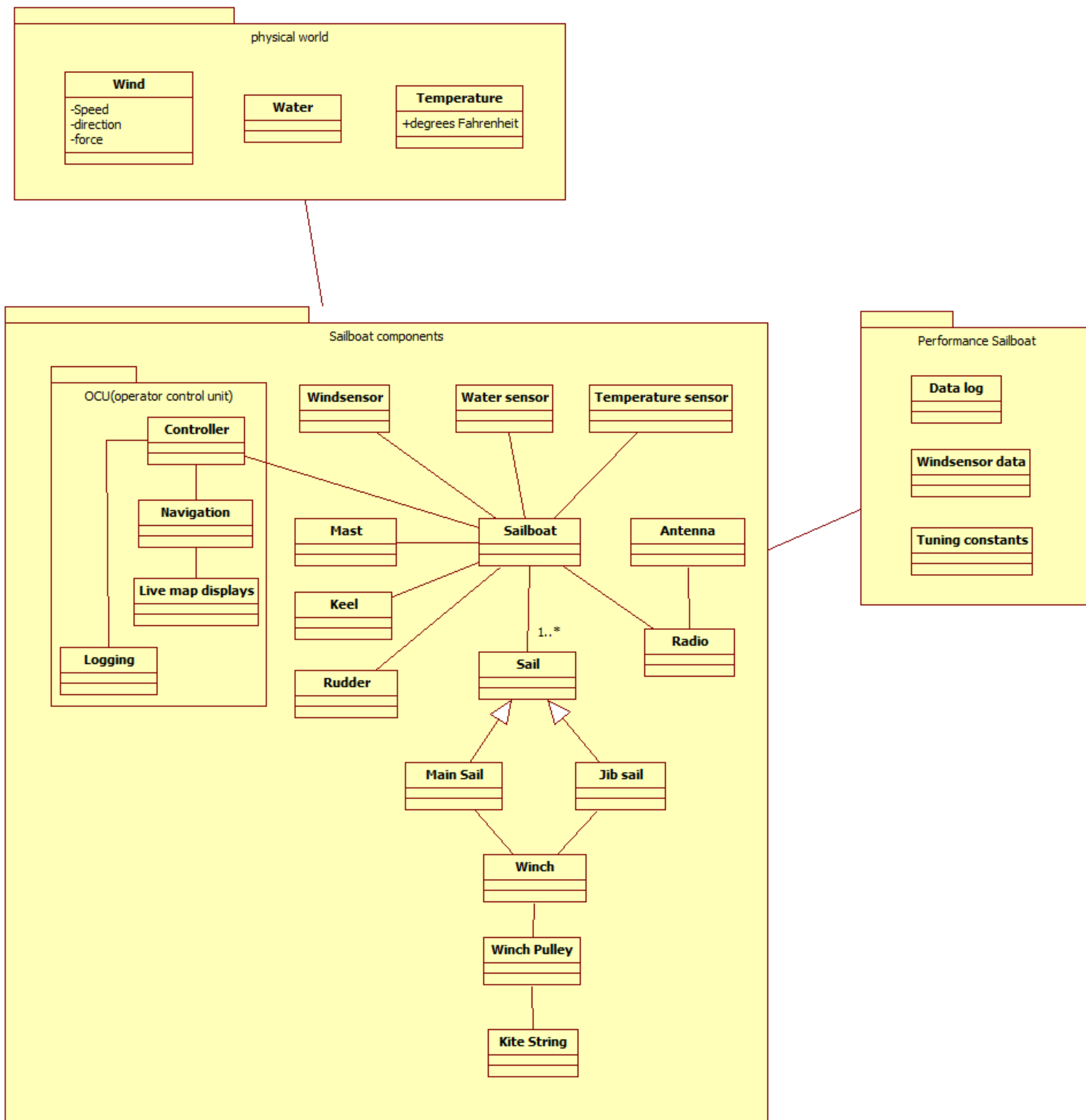
Voor het bouwen van een zeilboot is het noodzakelijk te bestuderen en te onderzoeken welke externe invloeden en middelen nodig zijn, zodat het besturen van een zeilboot in het water geoptimaliseerd kan worden. De onderdelen van de zeilboot horen zoveel mogelijk weerstand te kunnen bieden tegen zware weeromstandigheden. Data verzamelen van de omgeving zorgen ervoor dat de zeilboot op koers blijft en traceerbaar is.

Op de website www.olinsailbot.com staat informatie over de Olin's Robotic Sailing Team die een robotic zeilboot bouwt met robot technologie. Hun doel is het bouwen van een twee meter volledig autonoom zeilboot die in een competitie van 6^{e} jaarlijkse International Robotics Sailing Competition deelneemt en met het oog op de toekomst de allereerste robotic zeilboot bouwen die het Atlantisch Oceaan oversteekt.[3]

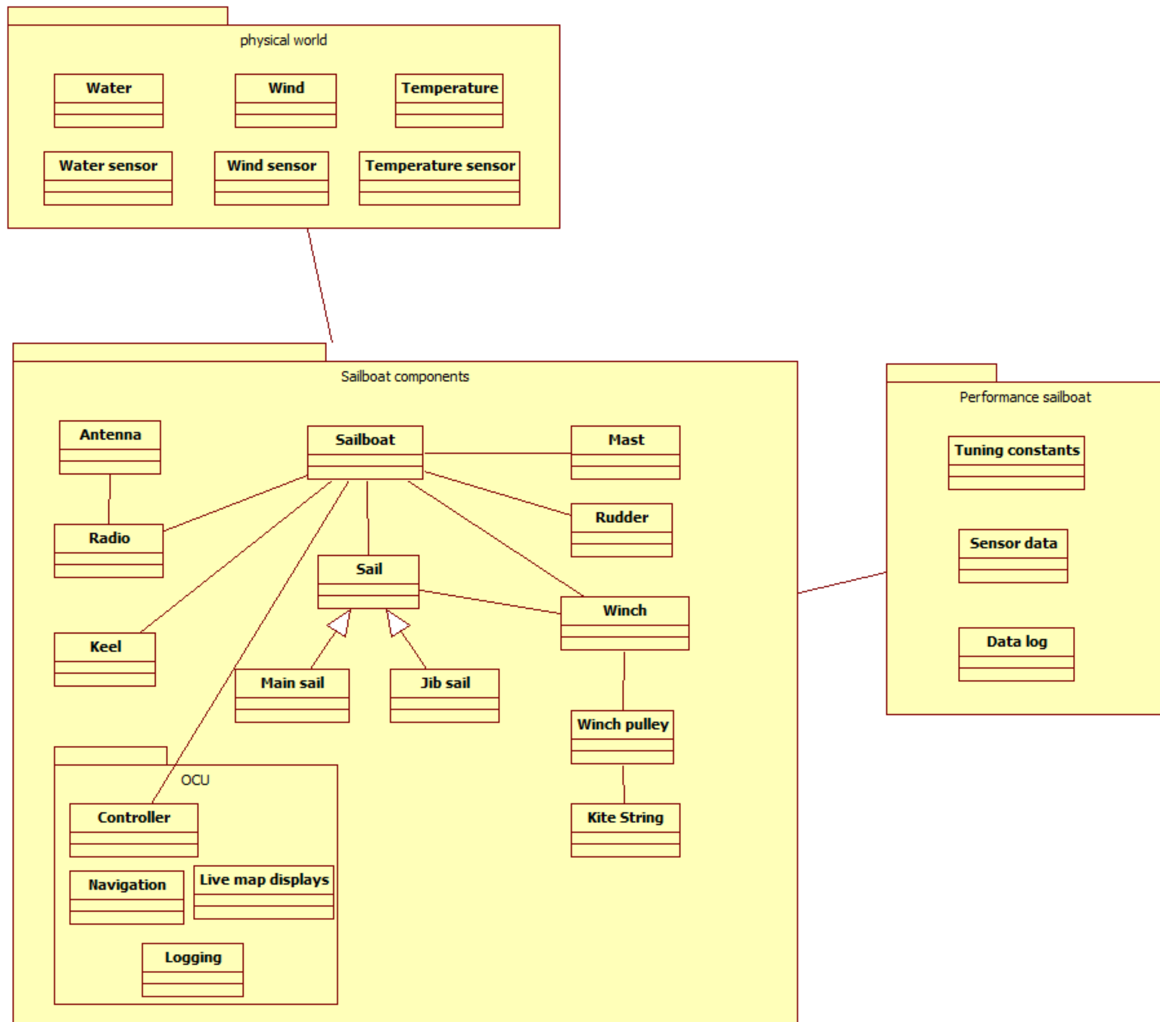


Figuur 2.1 Zeilboot model 1

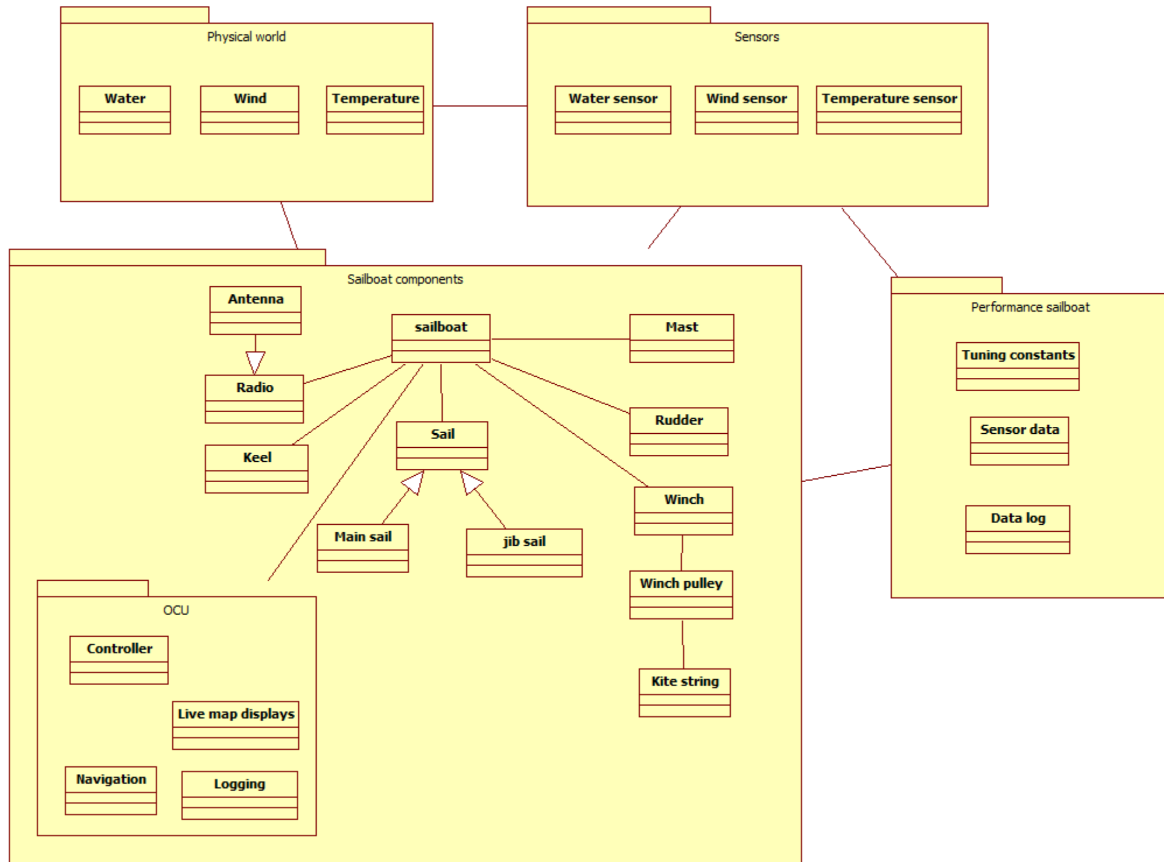
Module *Physical world*: het verzamelt gegevens over het fysieke omgeving(weersomstandigheden).
Module *Sailboat components*: bevat de onderdelen van een zeilboot.
Module *Performance sailboat:* verzamelt gegevens van de uitvoeringen van de zeilboot.
Module *OCU*: de zeilboot wordt hiermee verstuurd, waarbij de gegevens/data worden veranderd.



Figuur 2.2 Zeilboot model 2



Figuur 2.3 Zeilboot model 3



Figuur 2.4 Zeilboot model 4

5) Stel dat dit ontwerp uitgebreid moet worden met functionaliteit voor bepalen van de positie van de boot. Wat is dan de beste manier om het ontwerp uit te breiden?

a) met een nieuw package.

b) een nieuwe klasse in package *Physical world.*

c) een nieuwe klasse in package *Sailboat components.*

d) er moeten klassen toegevoegd worden bij zowel *Physical world* als *Sailboat components.*

Antwoord: *a*

6) Welk ontwerp heeft de voorkeur als het gaat om de relatie tussen de winch en de zeilen?

a) sail is verbonden met de winch, omdat ze beiden op hetzelfde niveau horen.

b) winch is verbonden met de gespecialiseerde klassen main sail en jib sail.

c) winch is alleen verbonden met de zeilboot.

d) de winch is zowel verbonden aan de klasse sail en de gespecialiseerde klassen main en jib sail.

Antwoord: *b*

7) Waar horen de 3 sensoren in het ontwerp?

a) In de physical world package, omdat de sensoren moeten worden verbonden met de fysieke omgeving.

b) In sailboat components package, omdat het onderdelen van een zeilboot zijn.

c) In een aparte package sensors, omdat ze een eigen functionaliteit hebben.

d) De drie sensoren zijn gespecialiseerde klassen van de klasse sensor in de package sailboat components.

Antwoord: *c*

8) Welke ontwerp zorgt ervoor dat information hiding beter toegepast kan worden als er veranderingen optreedt?

a) Figuur 2.4, omdat het ontwerp van het systeem uit meerdere modules bestaat waardoor er het eenvoudiger is het juiste module te vinden waarbij veranderingen gedaan moet worden.

b) Figuur 2.2, hier is direct zichtbaar wat erin een zeilboot zit(module *Sailboat components*) en welke module er invloed op heeft.

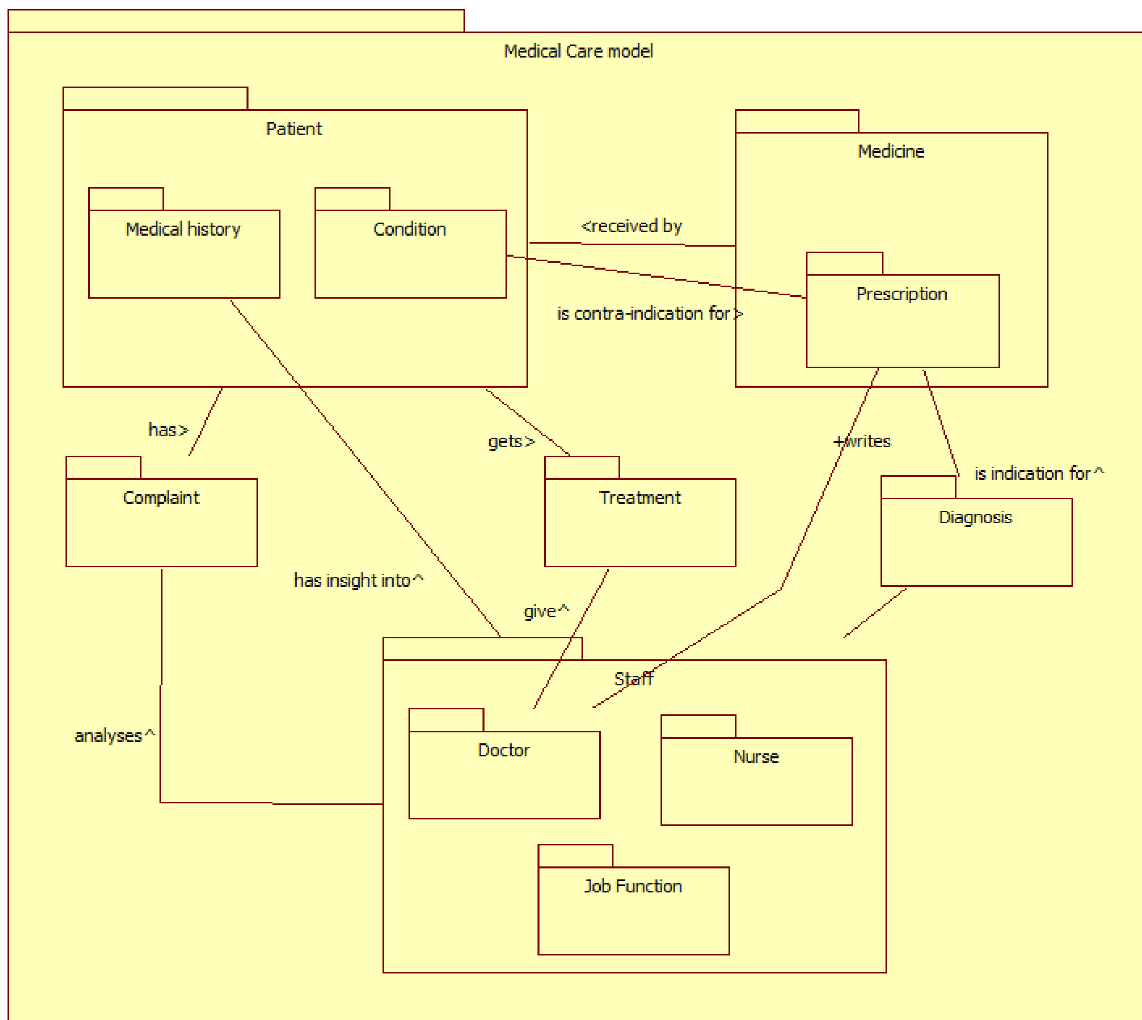
c) Er moeten nog zgn. poorten ontworpen worden in de modules.

d) Geen van de antwoorden.

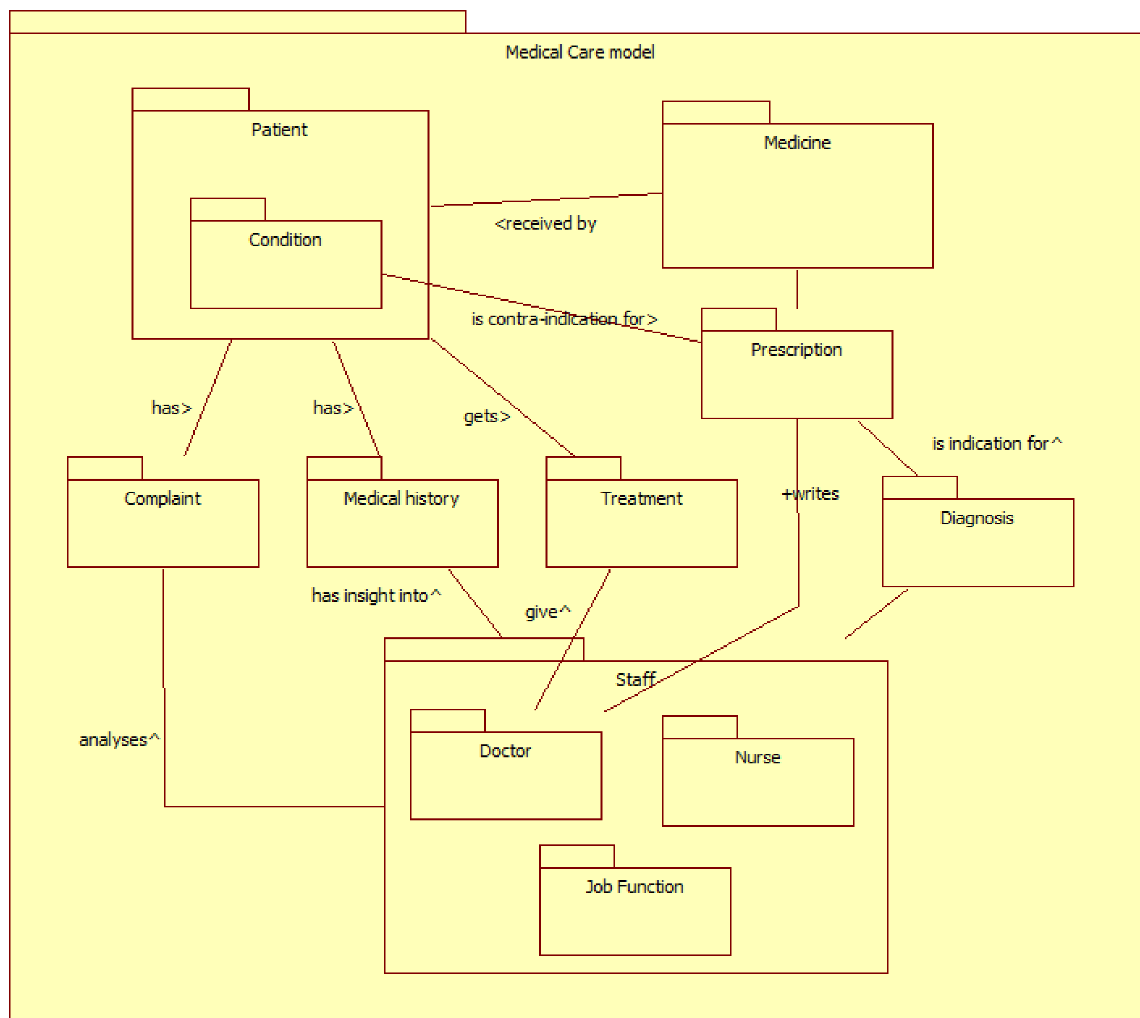
Antwoord: *c*

3.3 Casus 3: Zorgvraag

De patiënt komt met een klacht bij de dokter en de dokter analyseert de zorgvraag. De dokter bekijkt daarnaast de medicatiedossier van de patiënt voor achtergrondinformatie en nadere onderzoek van de conditie van de patiënt volgt. De arts stelt een diagnose en schrijft een voorschrift voor een geneesmiddel. Daarmee kan de patiënt het geneesmiddel afhalen.



Figuur 3.1 Zorgvraag model 1



Figuur 3.2 Zorgvraag model 2

9) Welk ontwerp is beter?
a) Figuur 3.1, omdat er sprake is van minder coupling en meer cohesion.
b) Figuur 3.1, zodra er veranderingen optreedt in één module heeft het geen invloed op andere modules.
c) Figuur 3.2, de package *Prescription* is geen onderdeel van de package *Medicine*.
d) Zowel a en b is juist.

Antwoord: *d*

10) Wat kan er aan het ontwerp verbeterd worden?
a) De packages *Complaint, Medical History* en *Diagnosis* hoort in de package *Treatment*, omdat het allemaal hoort bij een behandeling van een patiënt.
b) De packages *Complaint, Medical History* en *Diagnosis* hoort in de package *Treatment*, omdat er dan minder coupling optreedt tussen andere packages.
c) De package *Job Function* is niet relevant in dit model.
d) Alle bovenstaande antwoorden zijn juist.

Antwoord: *d*

Hoofdstuk 4 Conclusies & Reflectie

Bij het ontwerpen van systemen, hoor ik continu te analyseren wat beter kan en kom ik tot verschillende inzichten van hoe een model eruit kan zien. Mijn intuïtie en kennis gebruiken hoe een design het beste in modules ingedeeld kan worden, waardoor er beter ontwerp naar voren kan komen en het makkelijk te veranderen is. Dus zorgt het verder ook voor hergebruik van componenten en maakt het onderhoud ervan ook vele malen eenvoudiger en minder kostbaar.
Bij het toepassen van de design principles is het ideale geval dat door het gebruik van information hiding, zorgt voor minder coupling tussen modules en meer cohesion binnen modules. Maar door het indelen van een systeem in verschillende componenten op basis van hun verantwoordelijkheden, moet ik me vaak afvragen of welke objecten bij elkaar horen en is het wel een optimale ontwerp.
Als ik aan de hand van de design principles ontwerp, heb ik de kans dat ik andere design principles niet goed genoeg hanteer/overtreed.

Wat ik verder heb geleerd is dat het toepassen van design principles tijdens het ontwerpen van een systeem een enorm grijs gebied is. Het continu zoeken naar een optimaalste ontwerp voor een systeem is een proces waarbij veel vaardigheden en ervaring vereist, technisch inzicht en kennis. Welke design principles pas je toe, welke diagrammen gebruik je, welke technieken. Je begint op een bepaald niveau en gaat steeds naar een hogere level van ontwerpen. Continu feedback is vereist bij het gebruiken van het systeem. het beoordelen of het nodig is ontwerp te veranderen, want de wensen van mensen veranderen en is tijdsgebonden. Voor het bouwen van een systeem heb je mankrachten nodig die verschillende vaardigheden en inzicht hebben.

Referenties

[1] R.S. Pressman & Associates, Inc. ESE Module 3-4 Design Principles. 1995
[2] Indoor Karting "Race-Planet-Delft", Dave Stikkolorum
[3] www.olinsailbot.com, Olin's Robotic sailing Team