

WERKBOEK OOSE-OOAD VT_

Marco Engelbart
17 januari 2020

INHOUD

0	OOPD REVISITED.....	5
1	USE CASE MODELING	6
1.1	Instapopdracht.....	6
1.2	Zelfstudie en lesvoorbereiding	6
1.3	Opdrachten Use Case Modeling	7
1.3.1	Mastermind	7
1.3.2	De parkeergarage	7
1.3.3	De drankautomaat.....	8
1.3.4	Arnhem Bicycle Couriers	9
1.3.5	Use Case Process Sale.....	10
2	REQUIREMENTS.....	11
2.1	Instapopdracht.....	11
2.2	Zelfstudie en lesvoorbereiding	12
2.3	Opdrachten Requirements	12
2.3.1	Wat is FURPS?.....	12
2.3.2	Requirements hotelovernachtingssysteem.....	13
3	THEMA DOMAIN MODELING.....	14
3.1	Instapopdracht.....	14
3.2	Zelfstudie en lesvoorbereiding	15
3.3	Opdrachten Domain Modeling	16
3.3.1	Mastermind	16
3.3.2	Zonnestelsel.....	16
3.3.3	Postbedrijf	16
3.3.4	Informatica-opleiding	17
3.3.5	Stedentrip.....	17
3.3.6	Dynamiek in domeinmodel	18
3.3.7	A2B	21
4	THEMA SEQUENCE DIAGRAMS.....	22
4.1	Instapopdracht.....	22
4.1.1	Van diagrammen naar code	22
4.1.2	Adventure Quest - class diagram en sequence diagram bij code	23
4.1.3	Cohesion, coupling en delegation	24
4.2	Zelfstudie en lesvoorbereiding	25
4.2.1	System Sequence Diagrams.....	25
4.2.2	Operation contracts	25
4.2.3	Sequence Diagrams	26
4.3	Opdrachten (System) Sequence Diagrams	26
4.3.1	AdventureQuest – opstellen sequence diagrams	26
4.3.2	Mastermind	28
4.3.3	Bibliotheek.....	29

5	DESIGN CLASS DIAGRAMS	31
5.1	Zelfstudie en lesvoorbereiding	31
5.1.1	Leeswijzer Design Class Diagrams	31
5.2	Opdrachten Design Class Diagrams	31
5.2.1	AdventureQuest – opstellen design class diagram	31
5.2.2	A2B (vervolg)	31
6	MAPPING DESIGN TO CODE	32
6.1	Zelfstudie en lesvoorbereiding	32
6.1.1	Leeswijzer Mapping Design to Code	32
6.2	Opdrachten Mapping Design To Code	32
6.2.1	POS – van class diagram naar code	32
6.2.2	BankAccount – van class diagram naar code	32
7	ACTIVITY DIAGRAMS.....	33
7.1	Zelfstudie en lesvoorbereiding	33
7.2	Opdrachten Activity Diagrams	33
7.2.1	Naar school.....	33
7.2.2	De website	33
7.2.3	Webwinkel.....	33
8	OO DESIGN PRINCIPLES	34
8.1	Instapopdracht.....	34
8.1.1	Chuck-a-luck	34
8.1.2	Yahtzee	37
8.2	Zelfstudie en lesvoorbereiding	39
8.2.1	Kijkwijzer SOLID	39
8.2.2	Leeswijzer GRASP	39
8.3	Opdrachten OO Ontwerpprincipes	40
8.3.1	SOLID – Single Responsibility Principle.....	40
8.3.2	SOLID – Open/Closed Principle	40
8.3.3	AdventureQuest	41
8.3.4	SOLID – Dependency Inversion Principle.....	43
8.3.5	AdventureQuest vervolg	43
8.3.6	Hardloper.....	44
8.3.7	Boter Kaas en Eieren	45
9	GOF DESIGN PATTERNS: OBSERVER, STRATEGY EN STATE	46
9.1	Instapopdracht.....	46
9.2	Zelfstudie en lesvoorbereiding	47
9.3	Opdrachten GoF Design Patterns	48
9.3.1	PluralSight examples	48
9.3.2	AdventureQuest	49
9.3.3	Slimme deur in fabriek	49
10	GOF DESIGN PATTERNS: FACTORY METHOD, ADAPTER	50
10.1	Zelfstudie en lesvoorbereiding	50
10.2	Opdrachten	52

10.2.1 AdventureQuest	52
10.2.2 Koenen en Kramers	52

0 OOPD REVISITED

Opdracht:

Bouw Mastermind als een console-applicatie in java volgens de objectgeoriënteerde principes die je hebt geleerd in de propedeusecourse OOPD.

Maak ook een class diagram voor de gerealiseerde applicatie.

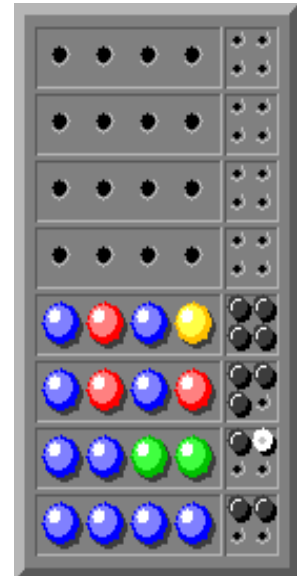
Hint: Maak voor het inlezen van de poging van de speler gebruik van de class `java.util.Scanner`.

Hoe werkt Mastermind?

De spelleider (dat is bij ons de computer) kiest een combinatie van vier gekleurde pionnetjes; een kleur kan vaker dan een keer voorkomen in dit rijtje. Daarbij kan gekozen worden uit de kleuren blauw, groen, rood en geel. De speler dient probeert vervolgens deze combinatie van kleuren te raden. De speler vult de eerste rij met gekleurde pionnen. Als alle vier de gaatjes zijn gevuld, dan bevestigt de speler zijn keuze. De spelleider meldt dan hoeveel kleuren goed zijn geraden. Voor elke kleur op de juiste positie plaatst de spelleider een zwarte pin. Voor een kleur die wel voorkomt, maar op een andere positie, wordt een witte pin gezet. Er wordt niet aangegeven welke kleuren (bijna) goed zijn, alleen het aantal wordt gemeld.

In de tweede beurt vult de speler de tweede rij. De spelleider geeft vervolgens wederom een hint over het aantal goede en bijna goede (d.w.z. kleur is goed, maar de positie niet) kleuren.

Als de speler na acht beurten niet de juiste kleurencombinatie heeft geraden, dan maakt de spelleider de kleurencombinatie bekend en is het spel afgelopen.



De communicatie tussen speler en applicatie loopt bijvoorbeeld als volgt. Daarbij dient de speler een kleur d.m.v. een letter (B=blue, G=green, R=red, Y=yellow) in te voeren.

```
Welcome to mastermind. Try to find the secret code.
```

```
Guess please: BBBB
```

```
right colour right position = 2
```

```
right colour wrong position = 0
```

```
Guess please: BBGG
```

```
right colour right position = 1
```

```
right colour wrong position = 1
```

```
Guess please: BRBR
```

```
right colour right position = 3
```

```
right colour wrong position = 0
```

```
Guess please: BRBY
```

```
right colour right position = 4
```

```
right colour wrong position = 0
```

```
You've found the secret code. Congrats!
```

1 USE CASE MODELING

1.1 Instapopdracht

Maak bij deze opdracht gebruik van de bij de propedeusecourse SAQ opgedane kennis.

Casus Bookplaza

Het bedrijf *BookPlaza* wil een systeem laten ontwikkelen waarmee klanten boeken kunnen bestellen. Voor je bestellingen kunt verrichten, dien je je als klant te registreren. Je moet dan de volgende gegevens invoeren: emailadres, wachtwoord, naam en adres.

Als een klant is ingelogd, dan kan hij bestellingen plaatsen. De klant zoekt een boek op basis van titel en/of auteur. Het systeem toont vervolgens een lijst met alle boeken die aan de zoekcriteria voldoen. De klant selecteert het juiste boek en geeft aan dat deze aan het winkelmandje moeten worden toegevoegd. Als er geen enkel boek aan de zoekcriteria voldoet, dan wordt dit door het systeem gemeld.

Als de klant nog meer boeken wil bestellen, dan zoekt hij opnieuw en het gevonden boek wordt in het winkelmandje gestopt. Als hij alle gewenste boeken in het mandje heeft zitten, dan geeft hij aan te willen betalen. Het systeem geeft dan eerst een overzicht van de bestelde boeken, de totaalprijs, alsmede het reeds bekende (aflever)adres. De klant krijgt de mogelijkheid om dit aan te passen. Na de bevestiging van bestelling en afleveradres, dient de klant aan te geven hoe hij wil betalen. Dat kan via een acceptgiro of via een online betaalservice. Als de klant voor de laatste optie kiest, dan wordt hij door het systeem doorgestuurd naar de betaalservice waar hij de betaling afhandelt. Als de betaling compleet is, dan wordt de klant bedankt voor zijn bestelling en stuurt het systeem een email ter bevestiging.

Verder dient de applicatie aan de klant de mogelijkheid te bieden om de status van de door hem verrichte bestelling(en) te bekijken.

Verder hebben medewerkers van *BookPlaza* de mogelijkheid om (gegevens van) de beschikbare boeken bij te werken en nieuwe boeken toe te voegen.

Opdrachten:

1. Identificeer de actor(en) en de use-case(s) en geef ze weer in een use-case diagram.
2. Beschrijf de gevonden use case(s) in brief format.
3. Beschrijf de gevonden use case(s) in fully-dressed format.

Opmerking: je kunt gebruik maken van het use case format dat op #OnderwijsOnline beschikbaar is als Word-document.

1.2 Zelfstudie en lesvoorbereiding

Lees uit het boek 'Applying UML and Patterns' van Craig Larman (in het vervolg van dit werkboek aangeduid met [LAR]) Chapter 1, 3, 6, overslaan 6.21.

Toelichting

- In chapter 1 wordt een zeer beknopt overzicht gegeven van wat OOAD (Object Oriented Analysis and Design) met behulp van UML zo al kan inhouden. Lees met name aandachtig 1.3, 1.4, en 1.5.
- In chapter 3 worden twee casussen geïntroduceerd, één over het NextGen POS systeem, die ook in de Powerpoint-presentaties gebruikt gaat worden en één over een Monopoly game. Het is van belang dat je met name NextGen POS goed bestudeert. Je ziet ook voor het eerst een voorbeeld van een *ontwerpbeslissing*: de opdeling van een systeem in *lagen*, en een opdeling van lagen in

objecten (zie fig. 3.1). Op dergelijke ontwerpbeslissingen zal later nader ingegaan worden. Verder wordt iets verteld over de *iteraties* die gebruikt gaan worden bij de ontwikkeling van de casus NextGen en waarin telkens nieuwe OOAD-concepten zullen worden uitgelegd. Wij zullen die leerlijn van het boek niet helemaal volgen: hier en daar maken we een sprong in het boek zoals je merkt, maar later komen we op de meeste overgeslagen hoofdstukken of paragrafen nog wel terug.

- In chapter 6 worden de belangrijkste zaken rond use cases behandeld. Dit is een erg belangrijk hoofdstuk, bestudeer het zorgvuldig. De afkorting UP staat voor *Unified Process*, een zogenaamd *procesmodel* of *softwareontwikkelmethode*. (R)UP komt in het semester OOSE niet expliciet aan bod.
- Besteed extra veel aandacht aan 6.6 t/m 6.16 in [LAR].
- In 6.8 [LAR] staat een uitgewerkt voorbeeld van een zeer belangrijke use case van NextGen POS: Process Sale. In het semester OOSE ga je het twee-koloms format voor use cases gebruiken.

1.3 Opdrachten Use Case Modeling

1.3.1 Mastermind

Stel een fully dressed use case op voor het spelen van het spel Mastermind.

Opmerking: De werking van Mastermind is beschreven bij de opdracht van Thema 0 OOPD revisited.

1.3.2 De parkeergarage

Om je auto te parkeren in een parkeergarage moet je het volgende doen. Je rijdt de garage binnen richting de slagboom en drukt op een knop van een automaat, zodat daar een ticket met de datum en de tijd uit komt. Dat lukt alleen als de garage niet vol is. Je pakt dat ticket, de slagboom gaat open en je zoekt een vrije plek en parkeert de auto. Iets in de parkeergarage registreert dat er een parkeerplek minder is en maakt dat bekend via de bekende displays. Als je weer weg wilt ga je eerst betalen door het ticket in een betaalautomaat te stoppen. Op een display verschijnt het verschuldigde bedrag, dat je kunt voldoen door munten in een gleuf te stoppen of door een chipknip te gebruiken. Na betaling krijg je het ticket terug waarop nu aangegeven staat hoe lang je geparkeerd hebt en dat je betaald hebt. Je stapt weer in de auto, rijdt tot vlak voor de slagboom en stopt het ticket in een gleuf van een automaat. De slagboom gaat nu open en je rijdt weg; iedereen moet nu kunnen zien dat er een parkeerplaats is vrijgekomen. Achter je gaat de slagboom meteen weer omlaag.

Opdrachten:

1. Identificeer de actor(en) en de use-case(s) en geef ze weer in een use-case diagram.
2. Beschrijf de gevonden use-case(s) in brief format.
3. Beschrijf de gevonden use-case(s) in fully-dressed format.

1.3.3 De drankautomaat

Op een dorstige dag komen Alex en Max elkaar tegen bij de koffie-automaat.

Alex: Goeiemorgen, ook een bakkie halen?

Max: Zeker, niets zo lekker wakker worden als met een kopje koffie.

Alex: Weet jij dan al hoe die nieuwe automaat werkt?

Max: Ja hoor, het is heel eenvoudig. Eerst pak je een bekertje en dat zet je in de bekerhouder. Je kunt kiezen tussen koffie en heet water, maar ik kies altijd voor koffie. Daarna krijg ik de keuze of ik melk in de koffie wil, en zo ja hoeveel. Ik neem nooit melk, dus dan druk ik op OK en vult de automaat mijn bekertje.

Alex: En suiker dan?

Max: Oh, dat ligt los in een doos, dat moet je er zelf in doen.

Alex: Ik kan altijd moeilijk wakker worden. Ik zou geheid het bekertje vergeten.

Max: Geeft niets, als je toch probeert koffie of heet water te kiezen geeft de automaat aan dat er geen bekertje in staat. Als er een bekertje in staat kan je opnieuw een keuze maken en loopt je bekertje vol.

Alex: Ik neem maar heet water. Koffie heb ik nooit lekker gevonden.

Max: Nou, bekertje pakken, heet water kiezen en je bekertje loopt vol! Daarna kan je zelf uit een doos een theezakje pakken en suiker als je wilt.

Alex: Ha, lekker. Wie zorgt trouwens voor nieuwe koffie?

Max: Dat doet Ria van de onderhoudsdienst. Er is een aparte knop op de automaat om de voorraadruimte te openen. Als je die indrukt, gaat er op de automaat een rood lampje branden en kun je de koffie bijvullen. Als je klaar bent, wordt het lampje weer groen; zit het nieuwe pak koffie echter niet goed dan blijft het lampje rood, de automaat is dan buiten werking, totdat het pak koffie goed gezet is. Meestal neemt Ria daarna zelf even een bakkie om te kijken of ze het goed heeft gedaan.

Alex: Ik snap geloof ik helemaal hoe dat ding werkt! Werk ze vandaag!

Max: Ja, jij ook!

Opdrachten:

1. Identificeer de actor(en) en de use-case(s) en geef ze weer in een use-case diagram.
2. Beschrijf de gevonden use-case(s) in brief format.
3. Beschrijf de gevonden use-case(s) in fully-dressed format.

1.3.4 Arnhem Bicycle Couriers

Casus “Arnhem Bicycle Couriers”

ABC (Arnhem Bicycle Couriers) is een bedrijf dat fietskoeriers in dienst heeft om pakketjes binnen Arnhem van de ene naar de andere locatie te brengen. Hier volgt een beschrijving van de gewenste situatie.

Een klant neemt contact op (meestal telefonisch) met het kantoor van ABC om een opdracht te plaatsen voor het ophalen en bezorgen van een pakket. Op het kantoor van ABC is een medewerker aanwezig die nieuwe opdrachten aanneemt. Deze medewerker wordt de planner genoemd. De planner maakt een nieuwe opdracht aan in het systeem en voert de klantgegevens, het ophaaladres en het bezorgadres van het pakket in en koppelt een koerier aan de opdracht.

De planner is ook verantwoordelijk voor het inplannen van koeriers, d.w.z. wie op welk dagdeel werkt.

Elke koerier heeft een smartphone met daarop een speciale ABC-app. *Als de koerier onderweg is, dan kan hij met deze app (nadat hij zich bij het systeem heeft aangemeld) de nog uit te voeren opdrachten opvragen die aan hem zijn toegewezen voor dat dagdeel. Het systeem toont dan een lijst met opdrachten. Vervolgens kan de koerier het systeem de opdracht laten selecteren waarvan het ophaaladres het dichtst bij zijn huidige locatie (wordt m.b.v. GPS bepaald) ligt. Meestal wordt voor deze optie gekozen, maar hij kan ook zelf een van de opdrachten uit de lijst selecteren.*

Van de geselecteerde opdracht worden het ophaal- en het afleveradres op een kaartje getoond met daarbij de verwachte reistijd. De koerier bevestigt de keuze en daarna zet het systeem de status van de gekozen opdracht op “in uitvoering”.

Dan begeeft de koerier zich naar het ophaaladres. Daar neemt hij het pakket mee en brengt het naar het bezorgadres. Vervolgens kan de koerier met de smartphone app invoeren dat de opdracht is afgerond.

Opdrachten:

1. Identificeer de actor(en) en de use-case(s) en geef ze weer in een use-case diagram.
2. Beschrijf de gevonden use-case(s) in brief format.
3. Beschrijf de gevonden use-case(s) in fully-dressed format.

1.3.5 Use Case Process Sale

Stel een activity diagram op dat correspondeert met de onderstaande fully dressed description van de use case Process Sale van het POS systeem,

<i>Primary actor:</i> Cashier	
<i>Stakeholders and interests:</i> Cashier, Customer, Company, etc.	
<i>Preconditions:</i> Cashier is identified and authenticated	
<i>Postconditions (Success Guarantee):</i> Sale is saved. Tax is correctly calculated. Etc.	
<i>Main Success Scenario</i>	
<i>Actor action</i>	<i>System responsibility</i>
1. Customer arrives at a POS checkout with goods to purchase.	
2. Cashier starts a new sale.	
3. Cashier enters item identifier	4. System records each sale line item and presents item description and running total. Price calculated from a set of price rules.
Cashier repeats steps 3-4 until indicates done.	5. System presents total with taxes calculated.
6. Cashier tells Customer the total, and asks for payment.	
7. Customer pays.	8. System handles payment.
	9. System logs the completed sale and sends information to external accounting and inventory systems. System presents receipt.
<i>Extensions (Alternative flows):</i>	
	3A. [Invalid identifier] 1. System signals error and rejects entry.

Opmerking: Bij thema 7 van deze course komen Activity Diagrams uitgebreider aan bod.

2 REQUIREMENTS

2.1 Instapopdracht

Lees het artikel “Requirements: An introduction” op <http://www.ibm.com/developerworks/rational/library/4166.html> en beantwoord de volgende vragen.

1. Welke van de begrippen *needs*, *features* en *requirements* horen bij het *solution domain*?
 - A. *needs*, *features* en *requirements*
 - B. *needs* en *requirements*
 - C. *features* en *requirements*
 - D. alleen *requirements*
2. Wat wordt bedoeld met ‘*elicit needs from stakeholders*’ ?
 - A. Het boven tafel krijgen van de needs van belanghebbenden.
 - B. Het documenteren van de needs van belanghebbenden.
 - C. Het valideren van de needs van belanghebbenden.
 - D. Het vertalen van de needs van belanghebbenden naar software features.
3. Wat leg je vast m.b.v. use cases?
 - A. *needs*
 - B. *features*
 - C. functional requirements
 - D. non-functional requirements
4. Welke term wordt gebruikt voor ‘*the ability to describe and follow the life of a requirement, in both forwards and backwards direction*’?
5. In een project wordt gewerkt volgens een iteratieve en incrementele aanpak. Aan het begin van het project heb je de requirements opgesteld. Op welke momenten in het project kunnen de requirements gewijzigd worden?
 - A. Op elk gewenst moment.
 - B. Na elke iteratie.
 - C. Niet meer, de requirements liggen vast.
6. Geef voor elk van de volgende eisen aan in welke categorie (volgens FURPS) het valt?
 - R 1. Het systeem moet 75% van de webpagina's binnen 2 seconden tonen. Voor de overige pagina's mag het nooit langer dan 5 seconden duren.
 - R 2. Een ervaren internetgebruiker moet binnen 2 minuten op basis van zijn selectiecriteria een hotelkamer kunnen boeken, ook als het de eerste keer is dat hij de hotelreserveringssite bezoekt.
 - R 3. Het systeem moet in hooguit 2 dagen met een extra Europese taal uitgebreid kunnen worden.
 - R 4. Het systeem moet 1.200 gebruikers tegelijkertijd aan kunnen met een piek van 2.500 gebruikers op de eerste werkdag van iedere maand. Tijdens de piek mag de snelheid met maximaal 20% dalen.
 - R 5. Het systeem moet laten zien dat het bezig is als het meer dan 2 seconden nodig heeft om zichtbaar op een actie van de gebruiker te reageren.

Lees ook: <https://www.cio.com/article/3174516/project-management/it-project-success-rates-finally-improving.html>

2.2 Zelfstudie en lesvoorbereiding

De volgende onderwerpen komen aan bod:

- Software process en RUP: [LAR] Ch 2, overslaan: 2.6 t/m 2.8;
- Inception: [LAR] Ch 4;
- Requirements: [LAR] Ch 5; Ch 7: 7.4 en 7.5
- Vision en Glossary: [LAR] Ch 7: 7.6 t.m 7.9.

Toelichting

- In het boek wordt (Rational) Unified Process (UP of RUP) gebruikt. RUP speelt geen centrale rol binnen het semester OOSE, maar het is handig om de terminologie te kunnen volgen. In chapter 2 staat een inleiding.
- In chapter 4 staat een kort overzicht van wat de Inception fase van RUP inhoudt.
- Chapter 5 beschrijft wat requirements of eisen zijn in de context van systeemontwikkeling. De eisen worden ingedeeld volgens het FURPS+ model, een handige en eenvoudige classificatie van de soorten eisen die je tegen kunt komen. Een andere, veel grovere maar wel belangrijke onderverdeling van eisen is die tussen functionele en niet-functionele eisen.
- Chapter 7 is weer een typisch software engineering hoofdstuk. In 7.4 worden goede voorbeelden gegeven van niet-functionele eisen. Het is goed om kennis te nemen van de overige paragrafen van dit hoofdstuk. Zo is de Vision (zie 7.6 en 7.7) een veelgebruikt document dat aan het begin van een project wordt geschreven met als doel de focus van het project helder te communiceren. Daarnaast moet je het doel van een Glossary (zie 7.8 en 7.9) kennen. Waarom? Het blijkt telkens weer dat het onduidelijk is wat een ontwerper precies bedoelt met bijvoorbeeld de attributen van een class in een model. Als goed omschreven wordt wat de betekenis is van elke class en van elk attribuut in een class model voorkom je problemen met de opdrachtgever en andere ontwikkelaars.

2.3 Opdrachten Requirements

2.3.1 Wat is FURPS?

De volgende websites definiëren het acroniem FURPS(+):

1. <http://agileinaflash.blogspot.nl/2009/04/furps.html>
2. <http://www.ibm.com/developerworks/rational/library/4706.html>

Opdrachten:

- A. Wat zijn de verschillen in inhoud tussen deze twee en denk je dat het uitmaakt voor de volledigheid van requirements?
- B. Wat zou jouw uitleg van het acroniem FURPS+ zijn? Leg dit per letter uit in een Nederlandstalige zin.

M.b.t. kwaliteitskenmerken van software is ook de ISO-norm 25010 van belang.

Voor meer informatie zie: https://nl.wikipedia.org/wiki/ISO_25010.

2.3.2 Requirements hotelovernachtingssysteem

De onderstaande casus is afkomstig uit het boek “Handboek Requirements” van Nicole de Swart.

Casus Sweet Dreams

De internationale hotelketen ‘Sweet Dreams’ wil haar hotelkamers gaan aanbieden via de bekende hotelreserveringssites. Ze moet daartoe eerst haar interne automatisering op orde brengen. Sweet Dreams heeft namelijk problemen met het systeem dat kamers toewijst aan hotelgasten. De directie heeft daarom een inventarisatie laten uitvoeren naar de problemen die de hotels ondervinden bij het onderbrengen van haar gasten. De directie is onaangenaam verrast door de resultaten van die inventarisatie en is onmiddellijk een project gestart dat het huidige systeem moet gaan vervangen door het nieuw te ontwikkelen systeem HOS (HotelOvernachtingsSysteem).

Voor het systeem HOS zijn de volgende eisen opgesteld:

1. In ieder hotel moet het systeem in de voertaal van het land ingesteld kunnen worden. Het systeem moet daarom de volgende talen aankunnen: Nederlands, Engels, Spaans, Frans, Duits, Portugees, Chinees, Russisch, Mexicaans en Japans. De HOS-reserveringssite moet dezelfde talen aankunnen.
2. Het systeem moet het aantal beschikbare kamers per kamertype juist aangeven. Het systeem mag er hooguit één op de tienduizend keer één of twee kamers naast zitten.
3. Het systeem moet bij het verzenden van de creditkaartgegevens onderschepping door derden onmogelijk maken. Bovendien mogen medewerkers deze gegevens niet kunnen raadplegen.
4. 80% van de baliemedewerkers moeten het systeem na maximaal twintig minuten uitleg zelfstandig kunnen bedienen.
5. Een ervaren internetgebruiker moet binnen twee minuten een hotelkamer kunnen reserveren, ook als het de eerste keer is dat hij de reserveringssite bezoekt.
6. Het systeem mag tussen 7.00 en 22.00 uur lokale tijd niet meer dan één keer per kwartaal uitvallen.
7. Het systeem moet in hooguit twee dagen met een extra taal uitgebreid kunnen worden.

Opdracht: Geef voor elk van deze requirements aan in welke categorie (FURPS) die valt.

3 THEMA DOMAIN MODELING

3.1 Instapopdracht

Maak bij onderstaande opdracht gebruik van de bij de propedeusecourse SAQ opgedane kennis en vaardigheden m.b.t. het opstellen van een Business Class Diagram.

Casus Langeafstandswandelroutes

In Nederland zijn er diverse langeafstandswandelroutes uitgezet. Hieronder zie je een overzicht van de etappes van de bekende langeafstandswandelroute het Pieterpad.

Traject 1 Pieterburen – Vorden

Etappe	Start- en Finishplaats	Afstand (km)
1	Pieterburen – Winsum	11
2	Winsum – Groningen	18
3	Groningen – Zuidlaren	21
4	Zuidlaren – Rolde	16
5	Rolde – Schoonlo	19
6	Schoonlo – Sleen	22
7	Sleen – Coevorden	19
8	Coevorden – Hardenberg	20
9	Hardenberg – Ommen	22
10	Ommen – Hellendoorn	20
11	Hellendoorn – Holten	15
12	Holten – Laren	15
13	Laren – Vorden	14

Traject 2 Vorden – Sint Pietersberg

Etappe	Start- en Finishplaats	Afstand (km)
1	Vorden – Doetinchem	26
2	Doetinchem - Hoog Elten	19
3	Hoog Elten – Millingen	12
4	Millingen – Groesbeek	20
5	Groesbeek – Gennip	15
6	Gennip – Vierlingsbeek	17
7	Vierlingsbeek – Swolgen	21
8	Swolgen – Venlo	22
9	Venlo – Roermond	31
10	Roermond – Pey	21
11	Pey - Sittard	18
12	Sittard – Strabeek	21
13	Strabeek – Maastricht	11
14	Maastricht – Sint Pietersberg	4

Elke langeafstandswandelroute bestaat uit een serie etappes verdeeld over meerdere trajecten. Voor elke etappe zijn de start- en finishplaats en de afstand in kilometers van belang. Hieronder zie je als voorbeeld het etappeschema van het Pieterpad (wellicht de bekendste wandelroute in Nederland). Elke langeafstandswandelroute heeft een unieke code. Voor het Pieterpad is dat bijvoorbeeld LAW9.

Stel een domeinmodel op:

- A. Identificeer mogelijke concepten en attributen in dit systeem. Gebruik hierbij “noun phrase identification”. Teken de voorlopige versie van het domeinmodel (ook wel: Businessclass Diagram).
- B. Voeg mogelijke associaties toe na analyse van verbanden die bestaan tussen de in onderdeel A gevonden zelfstandig naamwoorden. Label de associaties en geef de multipliciteiten aan.

3.2 Zelfstudie en lesvoorbereiding

Lees [LAR] Chapter 9 t/m 9.17.

Toelichting

- En dan begint in [LAR] chapter 9 het grote werk: het maken van een *domain model* of *conceptueel model*. Hierin toon je de *conceptuele classes*, hun *associaties*, en *attributen*. Het is typisch weer een product uit de analyse: ontwerpbeslissingen laat je niet zien. En dat is lastig; software ontwikkelaars hebben altijd de neiging er stiekem al keuzes in te zetten: een ‘handig’ nummertje als attribuut, een ‘handige’ class die je toch wel nodig zult hebben als je gaat bouwen, enzovoorts. Onderdruk die neiging. Het domain model is als analyse-product doorgewoond een weergave van de werkelijkheid, niets meer en niets minder. Houd je dus ook aan de ‘strategie van de kaartenmaker’: verzin geen nieuwe namen die je ‘beter’ vindt. Dat mag alleen in overleg met de opdrachtgever.
Omdat er alleen classes, associaties, en attributen in voorkomen lijkt een domain model erg veel op een *Entity-Relationship model* dat je misschien kent uit een database-gerichte course of semester. En dat is niet verwonderlijk: de makers van UML hebben zich erg laten inspireren door ER-modeling. De richtlijnen voor *Noun-Phrase Identification* om een domain model te maken komen ook uit de database-wereld.
In [LAR] table 9.1 staat nog een ander hulpmiddel om de conceptuele classes op te sporen: de *Conceptual Class Category List*. Loop die ook eens langs.
- Het begrip *domain model* komt niet voor in de officiële UML-documentatie. Door de methode van Larman te volgen werken we vanuit een domain model naar een design model. In een domain model komen geen operaties voor, alleen classes, attributen met simple data types en associaties. In een design model staan wél de operaties aangegeven. We moeten dus later op een of andere manier zien te bepalen welke class *welke* methode mag krijgen, en welke class in de implementatie een *referentie* krijgt naar een andere class waar hij een associatie mee heeft (aangegeven door de pijlen bij de associaties, die de zogenaamde *navigability* van het ene object naar het andere object weergeven).
Een design model kun je best nog ingewikkelder maken (zodat het werk voor de programmeur makkelijker wordt), maar veel verder dan het aangeven van methodes en navigability in een design class diagram zullen we niet gaan.
- Vervolgens gaan we nader in op associaties. Als hulpmiddel bij het vinden van die associaties kun je verder gaan met de NPI-richtlijnen en/of de criteria toepassen op pag. 151 [LAR]. Besteed veel aandacht aan de naamgeving: veel te vaak gebruikt men weinig inzichtelijke namen als ‘*heeft*’.

Verwar associaties niet met *rollen*. Een rol is het ‘eindpunt’ van een associatie, en elke associatie heeft dus twee rollen. Elke rol heeft een bepaalde *multipliciteit* en heeft al of niet een bepaalde *navigeerbaarheid*. Het is niet verplicht om elke rol ook een naam te geven, maar een goede rolnaam geeft soms wel meer inzicht in de rol (!) die een class speelt in een associatie. Rolnamen worden wel belangrijk in het design model, want de meeste tools maken gebruik van die namen bij het genereren van code. Hierover later meer. Navigeerbaarheid van een rol moet je pas in het design model aangeven. In UML zie je of een rol navigeerbaar is door een pijl; sommige tools geven die pijl default aan, wat je niet wilt zien in het domeinmodel. Bedenk dat navigeerbaarheid een *ontwerpsbeslissing* is!

Een model met 20 classes kan in principe $20 \times 19 / 2 = 190$ associaties bevatten! Te veel waarschijnlijk, dus we moeten ons een beetje in proberen te houden. In de analysefase kan het nuttig zijn je niet te beperken in het aantal associaties, omdat elke associatie er toe bijdraagt het domein beter te begrijpen. Tijdens design zullen soms associaties wegvallen omdat ze niet nodig zijn (niet ‘need-to-know’) of kunnen soms twee of meer associaties tussen twee classes samengevoegd worden tot één associatie.

3.3 Opdrachten Domain Modeling

Maak de onderstaande opdrachten m.b.v. een tekstverwerker (het NPI-gedeelte) en een UML-tool (Astar Professional of PlantUML).

3.3.1 Mastermind

Stel een domeinmodel op voor het spel Mastermind.

*Opmerking: Zie de beschrijving van het spel bij de opdracht van hoofdstuk 0 **OOPD revisited**.*

3.3.2 Zonnestelsel

Gegeven is de volgende domeinbeschrijving:

Ons zonnestelsel bestaat uit een zon met daaromheen draaiende hemellichamen zoals planeten en hun manen. Elk hemellichaam heeft een naam, een massa en een straal. Elke maan draait om één van de planeten.

De tijd die een planeet nodig heeft om eenmaal rond de Zon te draaien heet de omlooptijd. De tijd die een planeet nodig heeft om eenmaal rond haar eigen as te draaien heet de omwentelingstijd. Voor elke planeet dient het volgende te worden vastgelegd: de afstand tot de zon, de omlooptijd en de omwentelingstijd.

Stel een domeinmodel op.

3.3.3 Postbedrijf

Een postbedrijf dat pakketten bezorgt (zeg van UPS) bouwt een systeem waarmee pakketten on-line getraceerd kunnen worden. Voor een deel van dit systeem is de volgende domeinbeschrijving opgesteld.

Het bedrijf heeft klanten die pakketten kunnen versturen of ontvangen. Van een klant is in elk geval naam en adres bekend. Van pakketten weten we de afzender, de ontvanger, het gewicht en de huidige locatie. Een locatie kan een sorteercentrale zijn, maar ook een rit van een vrachtwagen. De sorteercentrales hebben onder andere een code en een adres. Van de ritten weten we zeker de vertrekdatum en -tijdstip en de sorteercentrale waar de rit begon.

Stel een domeinmodel op.

3.3.4 Informatica-opleiding

Gegeven is de volgende domeinbeschrijving:

Op een Informatica-opleiding worden gegevens van studenten en de vakken die ze volgen bijgehouden. Elke student heeft een uniek studentnummer en zit in een klas. De school wil natuurlijk ook de NAW-gegevens van elke student bijhouden.

Elke klas heeft een mentor, dat is een docent van de opleiding. Een docent heeft een unieke docentcode en ook de naam is van belang.

Een vak heeft een vakcode, een korte beschrijving en er wordt vastgelegd hoeveel studiepunten het vak oplevert.

Een eventueel cijfer voor een module van een student wordt ook opgeslagen.

- A. Stel een domeinmodel op.
- B. Stel ook een glossary op met daarin een beschrijving van de belangrijkste domeinconcepten.

3.3.5 Stedentrip

Casus 'Stedentrip'

Als je op stedentrip gaat, dan is het van belang om op de hoogte te zijn van de hotels en bezienswaardigheden in de betreffende stad. Elke gelegenheid (hotel of bezienswaardigheid) heeft een naam, lengtegraad en breedtegraad (de coördinaten op de wereldbol) en de stad waar het zich bevindt. Voor elke stad wordt vastgelegd in welk land het ligt.

Voor zowel hotels als bezienswaardigheden bestaan er recensies van bezoekers. Een recensie heeft een cijfer (op de schaal van 1 tot 10), een omschrijving en de naam en woonplaats van de recensent.

Elk hotel heeft een adres (straatnaam en huisnummer). Op één adres kan maar één hotel zijn gevestigd.

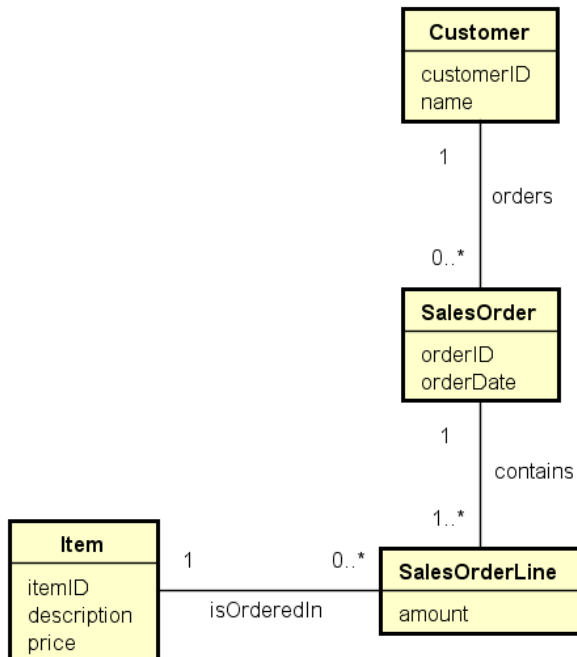
Elke bezienswaardigheid heeft een omschrijving. Elk hotel heeft een aantal sterren en er is bekend welke faciliteiten (zwembad, sauna, restaurant, wifi, etc.) het heeft.

Daarnaast is het interessant om te weten wat de loopafstand in minuten is van een hotel naar de diverse bezienswaardigheden.

- A. Maak een domeinmodel op basis van de bovenstaande beschrijving. Maak daar waar mogelijk en zinvol gebruik van generalisatie en/of specialisatie.
- B. Stel ook een glossary op met daarin een beschrijving van de belangrijkste domeinconcepten.

3.3.6 Dynamiek in domeinmodel

In het domeinmodel worden de classificaties weergegeven zoals ze in de real world bestaan. In deze real world zijn dingen aan verandering onderhevig. Kijk bijvoorbeeld naar onderstaand diagram:



Dit is een vereenvoudigde weergave van een klant die een of meerdere orders kan plaatsen. Elke order kan een of meerdere orderregels bevatten. Een orderregel bevat altijd 1 artikel en in de orderregel wordt het aantal exemplaren van dat artikel bijgehouden.

Wanneer je nu wil weten hoeveel de klant moet betalen voor een order lees je alle orderregels door bij die order. Bij elke orderregel neem je het bijbehorende artikel (item), zoekt de prijs op, vermenigvuldigt deze met het aantal uit de orderregel en telt dit bij de totaalprijs op voor de order. Dit lijkt dus helemaal in orde.

Wanneer we echter goed nadenken weten we eigenlijk wel dat er zaken zijn die aan verandering onderhevig zijn. De prijs van een artikel blijft niet altijd hetzelfde.

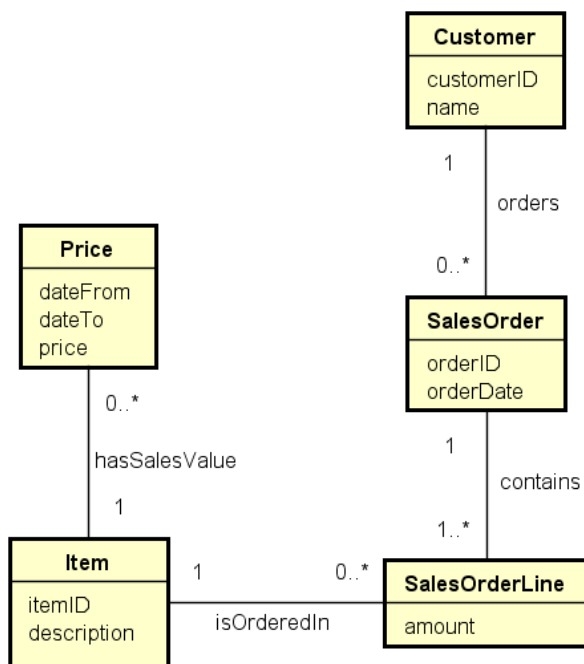
Probleem:

Klant heeft van een Item 10 stuks besteld tegen een prijs van 5 euro. Gedurende de wettelijke bedenktijd is de prijs van het artikel verhoogd naar 6 euro. Hierna stuurt de klant de 10 artikelen terug. Hoeveel geld krijgt de klant teruggestuurd volgens dit model?

Het antwoord is natuurlijk 60 euro terwijl hij er maar 50 betaald had. De prijs is kennelijk toch minder star verbonden aan het artikel als het voorbeeldmodel suggereert. De logische vervolgstap is dus om van de prijs een aparte class te maken.

Vraag: Hoeveel prijzen heeft 1 artikel? Welke attributen heeft deze class nodig om het eerder genoemd probleem op te lossen?

Antwoord: Op een zeker moment heeft een artikel maar één prijs maar zolang als het artikel bestaat heeft een artikel 0 of meerdere prijzen. Prijzen hebben de gewoonte periodiek te stijgen maar soms zijn er ook tijdelijke acties omdat een product in de aanbieding is. Om nu te weten welk bedrag een klant moet betalen voor het artikel moet je dus weten welke prijs op het moment van de order geldt. Bij het retourneren van het artikel krijgt hij ook het orderbedrag terug zoals gold op de orderdatum. Een verbeterd model ziet er dan als volgt uit:



Wanneer de klant nu zijn 10 artikelen retourneert, kan het systeem de orderregel opzoeken met deze artikelen en opzoeken welke prijs voor deze artikelen gold op de datum die genoemd is in de verkooporder. Jammer voor de klant. Hij krijgt maar 50 Euro terug. Maar dat is in deze situatie wel terecht.

Deze aanpak voegt dynamiek toe aan je domeinmodel. Hierdoor wordt het gemakkelijker uitbreidbaar voor het geval dat er bijvoorbeeld ook tijdelijke prijsafspraken voor een specifieke klant in de toekomst noodzakelijk zijn.

De uitleg die hier is gegeven aan de hand van een voorbeeld kun je ook veralgemeniseren. Dat gaat als volgt:

Uitgangspunt hierbij is dat je een eerste versie van je domeinmodel hebt gemaakt op basis van de informatie zoals je die tot dan toe hebt verkregen en geïnterpreteerd. In die interpretatie zitten vaak fouten. Daar kun je weinig aan doen omdat de kennis over het probleemdomein niet jouw specialiteit is maar van je opdrachtgever of de door hem aangestelde domeinspecialist.

Wanneer je met de domeinspecialist dit model gaat bespreken loop je eerst per geformuleerde conceptuele class de attributen langs. Per attribuut stel je de vraag of deze in de normale procesgang onveranderlijk verbonden is aan deze class. Indien dat het geval is, staat het er goed. Zo niet, dan pas je je model aan en maak je van dit attribuut een aparte conceptclass met een associatielijn naar de class waar je het attribuut zojuist uit verwijderd hebt.

Let op: In bovenstaande tekst staat de term 'in de normale procesgang'. Deze toevoeging is belangrijk om te vermijden dat je extreem ver gaat zodat je het doel van dynamiek voorbijschiet. (Analysis Paralysis) Voorbeeld: Een omschrijving van een artikel kan best incidenteel eens wijzigen. Dit maakt echter geen deel uit van de normale procesgang. Daarom zou je je doel voorbijschieten wanneer je van deze omschrijving een aparte class zou maken met daarin de periode waarbinnen het artikel zo genoemd wordt.

Wanneer je al je conceptuele classes op deze manier doorgewerkt hebt, ga je met de domeinspecialist de multipliciteiten in de associaties doornemen. Hierbij is er één vraag essentieel: Met hoeveel voorkomens van B heeft A te maken **gedurende de gehele levenscyclus** van A? Wanneer dat er meerdere zijn moet er de afhankelijkheid aan worden toegevoegd om vast te stellen welke B op een

gegeven moment actueel is voor A. In bovenstaand voorbeeld is dat de periode waarvoor een prijs geldt.

Als je nu het aantal B's behorend bij A hebt vastgesteld moet je ook op dezelfde manier het aantal A's behorend bij B vaststellen, ook weer gedurende de gehele levenscyclus van een B.

Mocht nu blijken dat A met meerdere B's van doen heeft en B ook met meerdere A's dan moet worden vastgesteld hoe je kunt weten welke A nou precies met welke B te maken heeft in een specifieke situatie. Vaak zal dit ook weer gedurende een periode zijn maar het kan ook een vastliggende volgorde of iets anders zijn.

Wanneer je met een domeinspecialist op deze manier je domeinmodel hebt doorgenomen zul je vaststellen dat het waarschijnlijk fors gegroeid is. Hierdoor lijkt het op het eerste gezicht complexer maar wanneer je de use cases die je op dat moment onderhanden hebt door het model laat lopen, merk je dat dit in het nieuwe model soepeler verloopt. Dat wil zeggen dat je per class minder tekst hebt voordat je via de associatie naar de volgende class gaat.

Doordat je de wereld waarin je use cases zich afspelen nu met alle bestaande dynamiek gemodelleerd hebt, is het gemakkelijker om het model uit te breiden wanneer je nieuwe use cases gaat toevoegen.

Opdrachten:

1. De overheid houdt de gegevens bij van alle geregistreerde motorvoertuigen en hun eigenaar. Elk voertuig wordt geïdentificeerd middels een kenteken. Daarnaast heeft elk motorvoertuig een uniek Voertuigidentificatienummer. Van het voertuig is verder het merk en type bekend alsook de kleur en de brandstof waarop het voertuig zich voortbeweegt. Daarnaast is het leeggewicht in kilo's van belang voor het vaststellen van de motorrijtuigbelasting.
Wanneer een motorvoertuig op naam gesteld wordt, krijgt de nieuwe eigenaar direct een aanslag motorrijtuigbelasting voor het restant van het lopend kwartaal.
Van een eigenaar worden NAW-gegevens en het bankrekeningnummer bijgehouden. Een eigenaar kan meerdere auto's in bezit hebben. Op een zeker moment kan een auto slechts één eigenaar hebben.
Voor het berekenen van de te betalen motorrijtuigen belasting wordt een tabel gehanteerd. Het te betalen bedrag is afhankelijk van de brandstof waarop de auto rijdt en de gewichtsklasse. Een gewichtsklasse bestaat uit een vanaf- en tot- aantal kg. (Bijvoorbeeld van 1251 t/m 1350 kg).

Opdracht: Maak een domeinmodel voor het omschreven probleemdomein.

2. Het Centraal Justitieel Incassobureau (CJIB) maakt ook gebruik van het systeem uit de vorige opgave. In Nederland kennen we de wet Mulder die voor de meeste overtredingen een sanctie oplegt aan de eigenaar van het motorvoertuig, die zelf maar moet zien dat hij dat geld terugkrijgt van de chauffeur die de vertreding beging. Veel van deze overtredingen worden vastgelegd door middel van flitspalen. In het algemeen is het zo dat er tussen het vaststellen van de overtreding en het toekennen van de sanctie enkele dagen verstrijken. Het CJIB heeft tabellen voor verschillende soorten overtredingen en ook de mate waarin deze wordt gepleegd. (Beetje te hard rijden of veel te hard rijden)

Opdracht:

Breid het domeinmodel uit de vorige opgave uit zodat boetes voor snelheidsovertredingen door het CJIB naar de juiste eigenaar kunnen worden verzonden. De juiste eigenaar is de eigenaar op het moment dat de overtreding is geconstateerd.

3.3.7 A2B

Onderstaande casus is bedoeld om de technieken die je hebt geleerd geïntegreerd toe te passen. Daarbij dien je steeds kritisch te kijken naar de kwaliteit en de consistentie van jouw werk.

Casus: A2B

Achtergrond

Het spin-off bedrijf A2B van de Automobilisten Vereniging Nederland (AVN) wil een app voor automobilisten bouwen waarmee verkeersinformatie, zoals files, wegwerkzaamheden en snelheidscontroles, kan worden opgevraagd.

A2B wil een OO analyse en ontwerp laten maken voor deze app.

Gewenste functionaliteit: Verkeersinformatie

Het systeem dient bij te houden uit welke plaatsen en verbindingstukken (bijv. een deel van een snelweg) het wegennet bestaat. Een verbindingstuk loopt van de ene naar de andere locatie. Op zo'n locatie kan zich een plaats bevinden, maar het kan bijvoorbeeld ook een knooppunt zijn.

Wegwerkzaamheden vinden plaats op de verbindingstukken. Ook voor files dient bekend te zijn welk verbindingstuk het betreft. Als een verbindingstuk helemaal niet toegankelijk is tussen bepaalde tijdstippen, dan moet dat aangegeven zijn. Verder wil A2B ook nog informatie over flitspalen en radarcontroles bijhouden. Het is gewenst om naast de plaatsnaam ook de lengte- en breedtegraad van plaatsen op te slaan.

Een automobilist wordt na invoer van het beginpunt en eindpunt van de reis op de hoogte gebracht van vertragingen en belemmeringen ten gevolge van files en wegwerkzaamheden. Voor elk van de bekende routes van beginpunt naar eindpunt moet de verwachte reistijd worden getoond, rekening houdend met de 'normale' reistijd voor de verbindingstukken en mogelijke vertragingen. Ook wil de automobilist weten waar hij eventueel snelheidscontroles kan verwachten.

Daarnaast verwacht A2B een beheersysteem, dat zal worden gebruikt door medewerkers van A2B, voor de basisgegevens die nodig zijn om deze service aan te kunnen bieden. Enerzijds moet het beheerssysteem het toevoegen, wijzigen en verwijderen van de relatieve statische gegevens over het wegennet en de flitspalen ondersteunen. Anderzijds moet het beheerssysteem ook het toevoegen, wijzigen en verwijderen van de relatief dynamische gegevens over files en radarcontroles ondersteunen.

Ook het verwijderen van dynamische verkeersgegevens gebeurt door medewerkers van A2B. Bij verwijdering van de gegevens geeft de medewerker aan dat deze files of radarcontroles wil verwijderen. Vervolgens moet hij aangegeven welke hij wil verwijderen.

Vooraf: Controleer bij deze opdracht steeds de consistentie tussen de opgestelde artefacten!

- Stel een use case model op voor te ontwikkelen A2B-informatiesysteem op basis van bovenstaande beschrijving.
- Maak voor de belangrijkste use case (het opvragen van de mogelijke routes voor een af te leggen reis met bijbehorende verkeersinformatie) een fully dressed beschrijving.
- Stel niet-functionele eisen op die je uit de casusbeschrijving kunt halen.
- Maak een domeinmodel, inclusief toelichting en een beschrijving van de domeinconcepten.

4 THEMA SEQUENCE DIAGRAMS

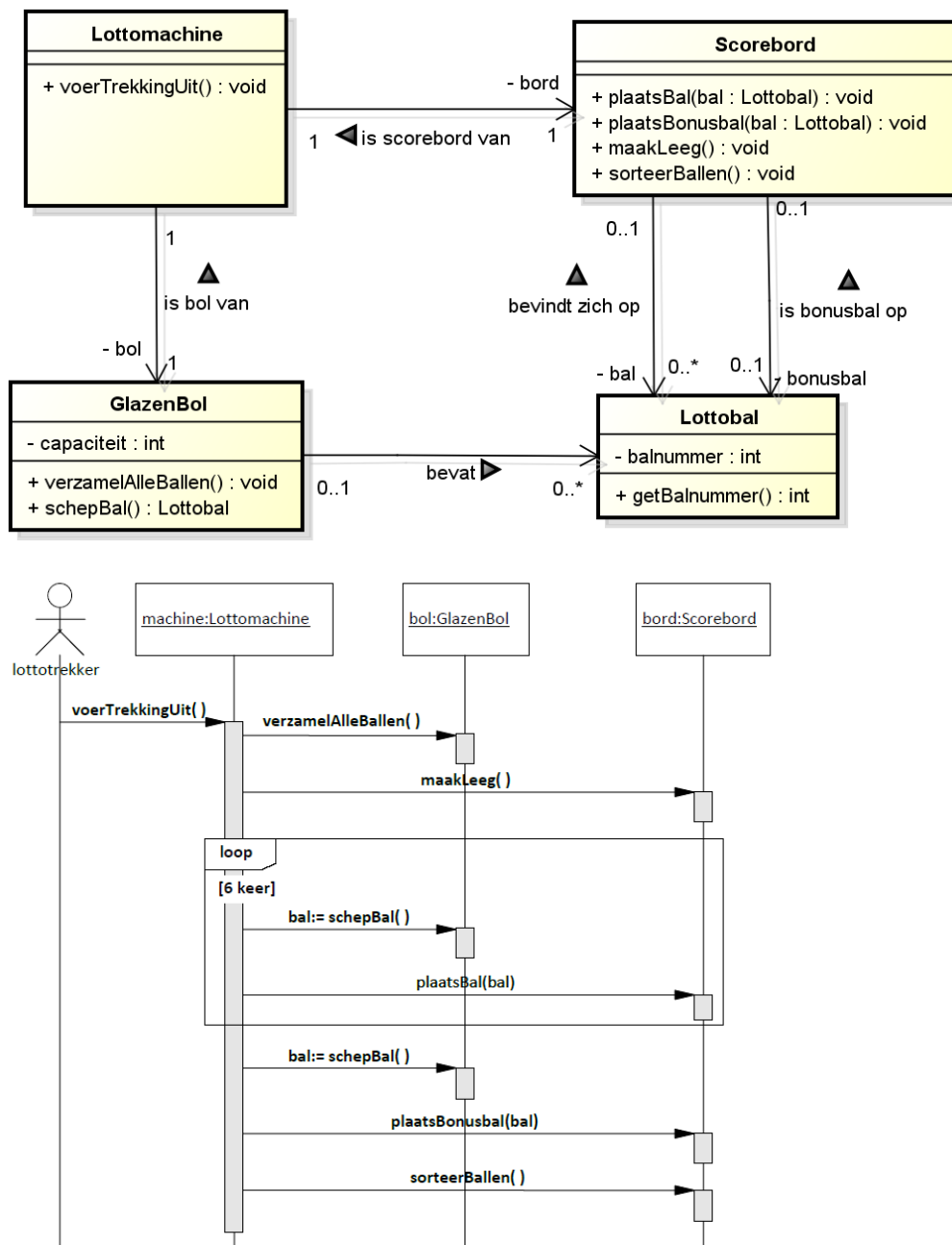
4.1 Instapopdracht

Lees de reader 'UML class en sequence diagrams', die bij de propedeusecourse OOPD is gebruikt, nogmaals door, zie de course OOSE-OOAD op #OnderwijsOnline.

4.1.1 Van diagrammen naar code

Opmerking vooraf: Bij deze opdracht heb je enige kennis van design class diagrams nodig. Dit onderwerp komt bij het volgende thema uitgebreider aan bod.

Bestudeer onderstaand class diagram en sequence diagram bij de lottotrekking:



Geef de code (in Java) van de class Lottomachine. De code dient te corresponderen met het afgebeelde class diagram en het sequence diagram van de operatie voerTrekkingUit().

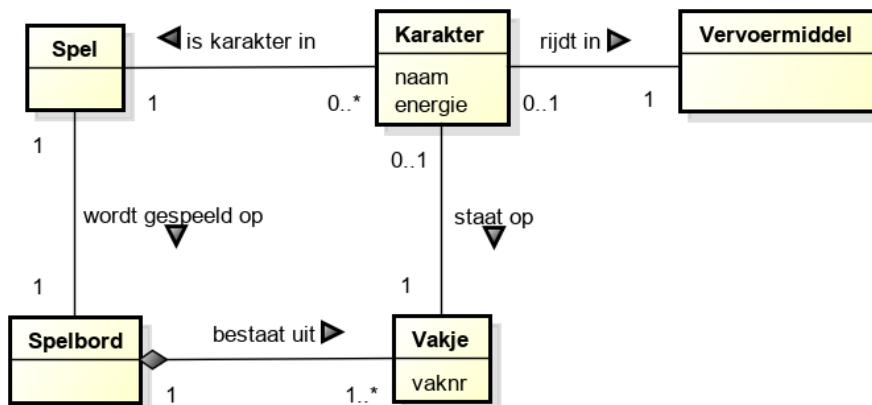
4.1.2 Adventure Quest - class diagram en sequence diagram bij code

Doel: Een class diagram en sequence diagram maken die corresponderen met gegeven code.

Deze opdracht betreft een multi-user computerspel genaamd AdventureQuest waarin karakters zich m.b.v. een vervoermiddel kunnen voortbewegen op een bord verdeeld in vakjes. Voor dit spel geldt het volgende:

- Een speler speelt met een bepaald *Karakter* het spel AdventureQuest.
- Een *Karakter* bezet een bepaald *Vakje* op het *Spelbord*.
- Het *Spelbord* bestaat uit een serie *Vakjes*.
- Een *Karakter* beschikt over een *Vervoermiddel* waarmee hij zich kan verplaatsen.

Domeinmodel:



- A. Hieronder zie je (incomplete) code bij de classes *Spel*, *Karakter* en *Spelbord*. Het betreft functionaliteit voor het verwisselen van de voertuigen van karakters. Neem onderstaande code over. Maak een main waarin het spel en het spelbord worden aangemaakt en waarin vervolgens een speler zich aanmeldt bij het spel.

```

public class Spel {
    private ArrayList<Karakter> karakters;
    private Spelbord bord = new Spelbord();

    public void meldAan (String naamKarakter) {
        Karakter k = new Karakter(naamKarakter);
        karakters.add(k);
        bord.plaatsOpVrijVakje(k);
    }

    public Karakter getKarakter (String naam) {
        for (Karakter k : karakters) {
            if (naam.equals (k.getNaam())) {
                return k;
            }
        };
        return null;
    }
    .....
}

```

```
public class Karakter {

    private String naam;
    private int energie = 1000;
    private Vakje vakje;
    private Vervoermiddel vervoermiddel;

    public Karakter(String naam) {
        this.naam = naam;
        vervoermiddel = new Vervoermiddel();
    }

    public Vakje getVakje () {
        return vakje;
    }

    public String getNaam () {
        return naam;
    }

    public void setVakje(Vakje v) {
        vakje = v;
        v.setKarakter(this);
    }
    .....
}
```

```
public class Spelbord {

    private ArrayList<Vakje> vakjes = new ArrayList<Vakje>();

    public Spelbord() {
        .....
    }

    public void plaatsOpVrijVakje(Karakter k) {
        Vakje v = kiesVrijVakje();
        k.setVakje(v);
    }

    private Vakje kiesVrijVakje() {
        .....
    }

    .....
}
```

- B. Maak een class diagram voor AdventureQuest uitgaande van het gegeven domeinmodel. Het class diagram dient consistent te zijn met bovenstaande code.
- C. Maak op basis van bovenstaande code een sequence diagram van de operatie meldAan (zie class Spel).

4.1.3 Cohesion, coupling en delegation

Bekijk op PluralSight de module **Thinking in Objects** van de course **Java Fundamentals: Object-oriented Design** van Allen Holub. Let bij het bekijken van deze module speciaal op de begrippen *cohesion*, *coupling* en *delegation*.

Beschrijf na het bekijken van deze module in eigen woorden wat de begrippen cohesion, coupling en delegation (Nederlands: cohesie, koppeling en delegatie), in de context van object-oriënted design, inhouden.

4.2 Zelfstudie en lesvoorbereiding

4.2.1 System Sequence Diagrams

In [LAR] chapter 10 komt het *System Sequence Diagram* (SSD) aan bod. In feite is een SSD niet als zodanig gedefinieerd in de UML, wél een *sequence diagram* in het algemeen. Een SSD is onlosmakelijk verbonden met een use case scenario. In een use case worden de events beschreven die een actor genereert op het systeem. Elk event moet een reactie van het systeem ten gevolge hebben, anders is het zinloos. Dus zal er een operatie uitgevoerd moeten worden als antwoord op dat event. In een SSD laat je precies die operaties zien die bij de events van de use case horen.

Het systeem moet die operaties uitvoeren, maar je weet dan nog niet welk onderdeel van het systeem daar de verantwoordelijkheid (*responsibility*) voor krijgt. (In een OO systeem bedoelen we met een onderdeel meestal de objecten). Als je dat zou weten dan zou je al een flink eind in het ontwerp zitten, en we zijn alleen nog maar aan het analyseren. We onderzoeken *wat* het systeem moet doen en niet *hoe* (het hoe houdt immers ook in dat je weet welk onderdeel de operatie uit moet voeren).

De operaties in een SSD noteer je op een ‘programmeertaal’-achtige manier: door middel van een naam gevolgd door ‘()’. Als een operatie argumenten en/of returnwaarde(n) heeft, dan moeten die óók in het SSD weergegeven worden, inclusief hun data types. Het is helemaal van het gebruikte tool afhankelijk of dat op een gebruiksvriendelijke manier kan gebeuren. Bij sommige tools moet je *notes* gebruiken om ze weer te geven. Trap in ieder geval niet in de valkuil om argumenten of returnwaarden te laten zien als onderdeel van de operatiennaam als dat in de tool niet kan!

Tot slot: de door een actor gegenereerde events worden meestal via een user interface gegenereerd. Van dat user interface trek je je voorlopig echter niets aan: doe alsof het niet bestaat. Zo wordt in de NextGen POS het event enterItem later vast wel via een barcode scanner of via een toetsenbord gegenereerd, en via dat user interface wordt eigenlijk de systeem operatie enterItem(itemID, quantity) aangeroepen. Maar in een SSD maak je je daar nog niet druk over: we doen of de aanroep van de operatie rechtstreeks van de actor :Cashier komt.

4.2.2 Operation contracts

Lees [LAR] Chapter 11.

Toelichting

- Het domain model van NextGen POS dat voorlopig gebruikt wordt staat in [LAR] fig.9.27 op pag 167. Raadpleeg dit zo nodig.
- Door een contract (pre- en postcondities) van een operatie te maken specificer je wat die operatie moet bewerkstelligen. Een contract voor een systeemoperatie, die rechtstreeks verband houdt met een use case, geeft in feite verdere detaillering van die use case. In de praktijk kun je dit doen voor de belangrijkste systeemoperaties van ingewikkelde use cases.
- Als je later in het ontwerp een ingewikkelde operatie tegenkomt kun je ook overwegen er eerst een contract voor te maken. Soms ontdek je tijdens het maken van een contract al dat het domeinmodel of de use case niet klopt, en moet je iteratief je modellen verbeteren.

4.2.3 Sequence Diagrams

Lees [LAR] chapter 15, behalve Singleton Objects, Diagram Frames in UML Sequence Diagrams, pagina's 235 t/m 239, pagina's 246 en 247.

Toelichting

- In chapter 15 wagen we de stap van analyse naar ontwerp! Tot nu toe hebben we requirements, use cases, een domeinmodel, system sequence diagrams, en contracten voor operaties uit die SSD's. Maar nergens hebben we bepaald *hoe* het te bouwen systeem iets moest uitvoeren. Dat gaan we nu doen, op een tamelijke grove manier, met behulp van *interaction diagrams*. Zoals de naam al aangeeft, toont zo'n diagram de interactie tussen objecten. Een object heeft een interactie met een ander object als er een boodschap gestuurd wordt van de een naar de ander.
- Welke objecten kunnen elkaar boodschappen sturen? Nodig is natuurlijk dat het zendende object het ontvangende object kent. Hij moet een *referentie* naar de ontvanger hebben, zodat hij naar de ontvanger kan *navigeren*. Maar zoiets is in het domeinmodel nog niet aangegeven.
- Wel zie je in het domeinmodel associaties tussen classes. Als er een associatie tussen twee classes is, is het in principe mogelijk dat een instantie van de ene class een instantie van de andere zou kunnen kennen, dus daar ligt een belangrijk aangrijpingspunt (maar niet het enige, zoals we later zullen zien). Voor het maken van een interaction diagram is dat domeinmodel dus je eerste houvast. Gebruik het!
- Chapter 15 gaat alleen over de notatie van interaction diagrams. Hoe je het maken van een interaction diagram aan zou kunnen pakken komt later aan de orde. Met UML kun je redelijk veel dingen aangeven in een interaction diagram, maar de tools hebben daar lang niet alles van geïmplementeerd zodat het gebruik van *notes* onontbeerlijk is.

4.3 Opdrachten (System) Sequence Diagrams

4.3.1 AdventureQuest – opstellen sequence diagrams

Doel: een sequence diagram maken van systeemoperaties vanuit een domeinmodel. Het design class diagram zoveel mogelijk door het gebruikte tool laten aanpassen.

Ga bij deze opdracht uit van de beschrijving van het spel AdventureQuest en het bijbehorende domeinmodel (zie opgave 4.1.2).

Hieronder staan beschrijvingen van de use cases **Teleporteren** en **Wisselen van vakje** en bijbehorende systeemoperaties. Na deze beschrijvingen volgen de uit te voeren opdrachten.

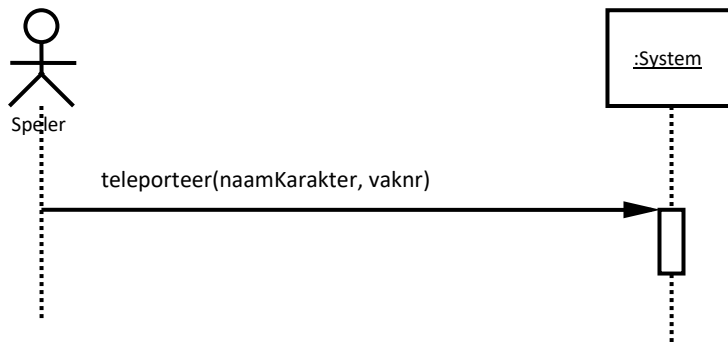
Use case **Teleporteren** en systeemoperatie **teleporteer()**:

Use case **Teleporteren**, brief description.

Een karakter kan zich teleporteren naar een ander vakje in de wereld, d.w.z. hij verdwijnt van het ene vakje en verschijnt (vrijwel tegelijkertijd) op een ander vakje. De energie van het karakter wordt daardoor met 20 verlaagd.

Einde use case.

De use case Teleporteren heeft aanleiding gegeven tot een systeemoperatie **teleporteer()**, met twee parameters en zonder returnwaarden:



Opmerking: De parameter naamKarakter is de naam van het karakter dat wordt verplaatst en vaknr is het nummer van het doelvakje.

Precondities van teleporteer(naamKarakter, vaknr):

Vóórdat deze operatie uitgevoerd wordt bestaat de volgende situatie:

- Er zijn al instanties van Spel en Spelbord aangemaakt, en ook alle instanties van Vakje bestaan al.
- Er is een instantie *k* van Karakter waarvan de naam gelijk is aan naamKarakter.
- Er is een instantie *v1* van Vakje waar het karakter *k* op staat.
- Er is een instantie *v2* van Vakje waarvan het vaknummer gelijk is aan vaknr en dit vakje is leeg.

Postcondities:

Nadat de operatie uitgevoerd is moet het volgende gelden:

- Het vakje *v1* is leeg.
- Het karakter *k* staat op het vakje *v2*.
- De waarde van het attribuut *energie* van het karakter *k* is verminderd met 20.

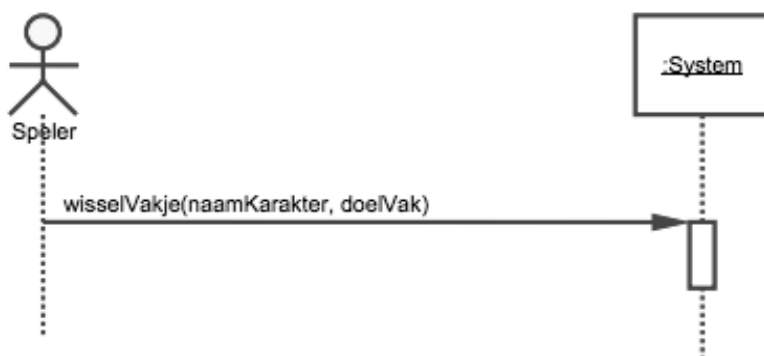
Use case **Wisselen van vakje** en systeemoperatie **wisselVakje()**:

Use case Wisselen van vakje, brief description.

Een karakter dat meer dan 20 aan energie heeft, kan van positie verwisselen met een karakter naar keuze. De energie van het initiatiefnemende karakter wordt met 20 verminderd.

Einde use case.

De use case **Verwisselen van vakje** heeft aanleiding gegeven tot een system operation **wisselVakje()**, met twee parameters en zonder returnwaarden:



Opmerking: De parameter naamKarakter is het initiatiefnemende karakter, doelVak het nummer van het vakje waar het andere karakter zich staat en waar de initiatiefnemer naartoe wil.

Precondities van wisselVakje (naamKarakter, doelVak):

Vóórdat deze operatie uitgevoerd wordt bestaat de volgende situatie:

- Er zijn al instanties van Spel en Spelbord aangemaakt; ook alle instanties van Vakje bestaan al.
- Er is een instantie k1 van Karakter die de naam draagt die als parameter meegegeven wordt (naamKarakter),
- Het karakter k1 bevindt zich op vakje v1.
- Er is een instantie v2 van Vakje waarvan het vaknummer gelijk is aan het meegegeven nummer (doelVak).
- Er is een instantie k2 van Karakter dat zich op vakje v2 bevindt.

Postcondities:

Nadat de operatie uitgevoerd is moet het volgende gelden:

Als de waarde van het attribuut energie van karakter k1 vooraf groter was dan 20, dan geldt:

- Karakter k1 bevindt zich op vakje v2, karakter k2 bevindt zich op vakje v1.
- De waarde van het attribuut energie van karakter k1 is verminderd met 20.

Opdrachten:

- Codeer de systeemoperatie **teleporteer()** in java.
Maak ook een sequence diagram bij deze systeemoperatie.
- Codeer de systeemoperatie **wisselVakje()** in java.
Maak ook een sequence diagram bij deze systeemoperatie.
- Maak het class diagram compleet.
- Beoordeel a.d.h.v. de sequence diagrams van onderdelen A, B en C of de verantwoordelijkheden aan de juiste objecten zijn toegewezen. Zo niet, welke verbetermogelijkheden zijn er?
Maak indien nodig verbeterde versies van de sequence diagram, het class diagram en de code.

4.3.2 Mastermind

Stel een system sequence diagram op op basis van het volgende main success scenario van fully dressed use case voor het spelen van Mastermind.

Main success scenario			
	Actor Action		System Responsibility
1	Speler geeft aan mastermind te willen spelen.	2	Systeem kiest een geheime kleurencombinatie.
3	Speler doet een poging om de geheime combinatie te raden.	4	Systeem vergelijkt de geheime combinatie met de poging van de speler.
		5	Systeem geeft aan hoeveel kleuren goed zijn en hoeveel bijna goed.
Stappen 3 t/m 5 worden herhaald tot geheime combinatie is geraden of beurten op zijn.			
		6	Systeem geeft melding (winst of verlies) en beëindigt spel.

Opdrachten:

- Stel een system sequence diagram (SSD) op op basis van het volgende main success scenario van fully dressed use case voor het spelen van Mastermind.
- Stel voor elk van de systeemoperaties uit het SSD een sequence diagram op

4.3.3 Bibliotheek

Leden kunnen m.b.v. de computers in de bibliotheek boeken lenen. Daarnaast kunnen deze computers worden gebruikt voor het verlengen en reserveren van boeken, maar dat kan ook thuis via de website van de bibliotheek.

Hieronder zie je een beschrijving van de use case **Boek lenen**.

Use case: **Boeken lenen**

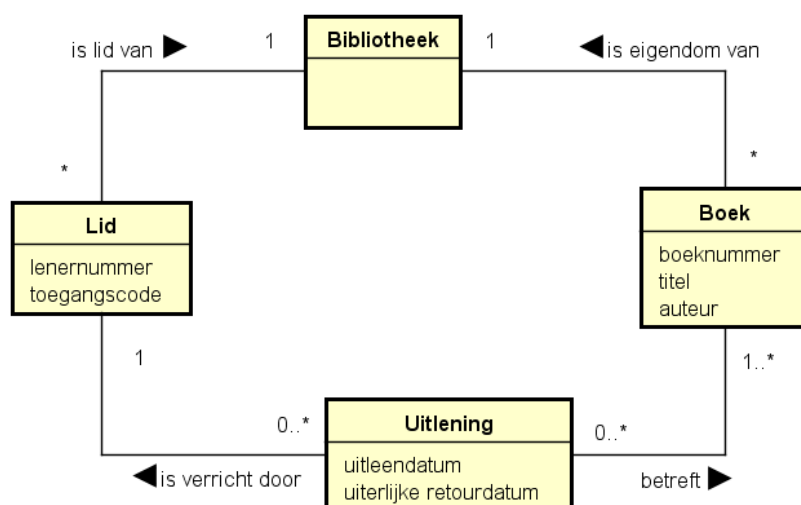
Primary Actor: Lid van bibliotheek

Brief description: Een lid wil een of meerdere boeken lenen. Hij meldt zich aan met linummer en pincode en voert vervolgens de boeknummers van de uit te lenen boeken in. Tot slot wordt er een bon met de uit te lenen boeken afgedrukt.

Main Success Scenario

	Actor Action		System Responsibility
1	Deze use case begint als een lid een of meerdere boeken wil lenen.		
2	Het lid meldt zich aan met lenernummer en toegangscode.	3	Het systeem controleert lenernummer en toegangscode.
4	Het lid geeft het boeknummer van het boek dat hij wil lenen.	5	Het systeem registreert dat dit boek is uitgeleend.
		6	Het systeem toont de titel en auteur die bij dit boek horen.
Stap 4, 5 en 6 worden herhaald tot de klant alle boeken heeft verwerkt.			
7	Het lid vraagt om een bon.	8	Het systeem print een bon met titel, auteur en retourdatum van de te lenen boeken.
		9	Het systeem sluit de uitleentransactie af.

Voor de bibliotheek is het volgende domeinmodel opgesteld.



Opdrachten:

- A. Breid de use case uit met pre- en postcondities en eventuele alternative flows.
- B. Bij de bibliotheek wordt voor een uitgeleend boek dat wordt teruggebracht ook de feitelijke retourdatum vastgelegd. Pas het domeinmodel aan zodat deze informatie ook kan worden vastgelegd.
- C. Geef een system sequence diagram bij de use case **Boeken lenen** (zie hierboven).
- D. Stel voor de systeemoperatie, die correspondeert met actor action 4 uit het main success scenario van de use case **Boeken lenen**, een operation contract op.
- E. Werk de systeemoperatie, die correspondeert met actor action 4 uit het main success scenario van de use case **Boeken lenen**, uit in een sequence diagram.

5 DESIGN CLASS DIAGRAMS

5.1 Zelfstudie en lesvoorbereiding

5.1.1 Leeswijzer Design Class Diagrams

Lees [LAR] 16.2 t/m 16.6, 16.21.

5.2 Opdrachten Design Class Diagrams

5.2.1 AdventureQuest – opstellen design class diagram

Zet het domeinmodel van Adventure Quest (zie opdrachten 4.1.2 en 4.3.1) om naar een design class diagram en zorg dat dit consistent is met de sequence diagrams en code van **meldAan**, **teleporteer** en **wisselVakje**.

Door het maken van het sequence diagram zet jouw UML-tool (als je de tool goed gebruikt) het domeinmodel automatisch om in een design class diagram. Controleer of dat goed gaat. Je dient zelf nog wel een aantal dingen toe te voegen en/of te controleren:

- datatypen van de attributes van classes;
- visibility van attributes en methods;
- navigability van associaties;
- dependencies.

5.2.2 A2B (vervolg)

Deze opdracht is een vervolg op opdracht 3.3.7.

- Op de OOAD-site op #OnderwijsOnline is een voorlopige versie van het design class diagram gegeven voor A2B (zie A2B.asta).
Stel een sequence diagram op bij de systeemoperatie `bepaalRouteInformatie()`. Deze systeemoperatie krijgt twee plaatsnamen als parameter (de start en de bestemming van de reis) en retourneert een `ArrayList` van `Strings` (per mogelijke route de routeinformatie als tekst).
- Maak het design class diagram compleet.
- Op de OOAD-site op #OnderwijsOnline is de geraamte van de java-code die correspondeert met het gegeven design class diagram en de systeemoperatie `bepaalRouteInformatie()` gegeven.
Codeer deze systeemoperatie in java.

6 MAPPING DESIGN TO CODE

6.1 Zelfstudie en lesvoorbereiding

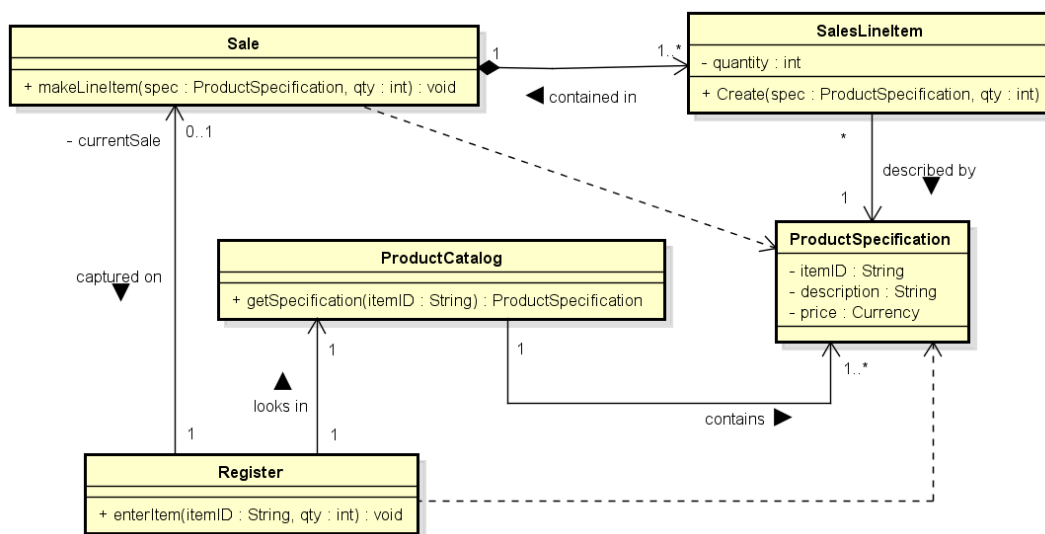
6.1.1 Leeswijzer Mapping Design to Code

Lees [LAR] Chapter 20.

6.2 Opdrachten Mapping Design To Code

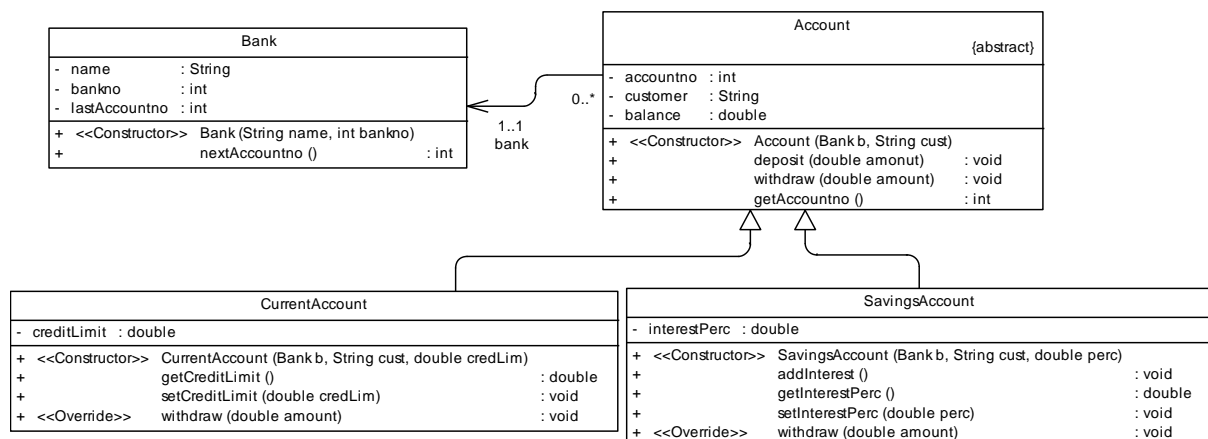
6.2.1 POS – van class diagram naar code

Geef de initiële java-code voor de classes Register en ProductCatalog (zie onderstaand design class diagram).



6.2.2 BankAccount – van class diagram naar code

Geef de initiële java-code voor de classes Account en CurrentAccount.



7 ACTIVITY DIAGRAMS

In het proces van analyse en design loop je soms tegen gedrag aan dat lastig is te beschrijven met de diagrammen die we tot nu toe hebben behandeld.

Neem als voorbeeld een bestelling van een boek bij een grote internet aanbieder. Het lijkt zo simpel: je bestelt een boek, je betaalt met iDeal en de volgende dag gaat de bel en staat de postbode met een pakje met daarin je boek. Als je er even over nadenkt, dan snap je al snel dat er achter de schermen heel veel gebeurt voordat de postbode op de stoep staat. Er zijn ook heel veel verschillende partijen bij betrokken: Er is een systeem waar je kan zoeken en bestellen, de betaling wordt door een ander systeem afgehandeld. Wanneer alles klopt, dan moet de distributie geïnformeerd worden en deze pakken je boek in. Zij dragen de bestelling over aan de bezorgdienst die op hun beurt er weer alles aan doet zodat jij met *track & trace* kan nagaan of de postbode er al aan komt.

Voor zo'n proces kan je prima een *activity diagram* gebruiken om dit te analyseren. Een activity diagram beschrijft welke reeks van activiteiten tegelijk en na elkaar moet worden uitgevoerd om iets voor elkaar te krijgen.

7.1 Zelfstudie en lesvoorbereiding

Leeswijzer Activity Diagrams

Lees [LAR] Chapter 28 over Activity Diagrams

7.2 Opdrachten Activity Diagrams

7.2.1 Naar school

Stel een activity diagram op bij de volgende beschrijving.

's Ochtends sta ik op en ga ik ontbijten. Als ik op tijd ben ga ik op de fiets. Zo niet, dan ga ik met de brommer naar school. Als het regent dan ga ik altijd met de bus naar school. Daar aangekomen begin ik vol enthousiasme met het volgen van de lessen.

7.2.2 De website

Stel een activity diagram op bij de volgende beschrijving. Maak hierbij gebruik van swimlanes

Een browser stuurt een verzoek om informatie naar een server toe. Deze ontvangt het verzoek, leest het en stuurt dan alvast het verzoek door naar de database. Tegelijkertijd wordt er een authenticatie uitgevoerd. Als er een antwoord van de database is gekomen en als de authenticatie is uitgevoerd wordt het antwoord teruggestuurd naar de browser. Deze zal het gestuurde antwoord op het scherm tonen.

7.2.3 Webwinkel

Stel een activity diagram op bij het hieronder beschreven orderproces. Maak hierbij gebruik van swimlanes.

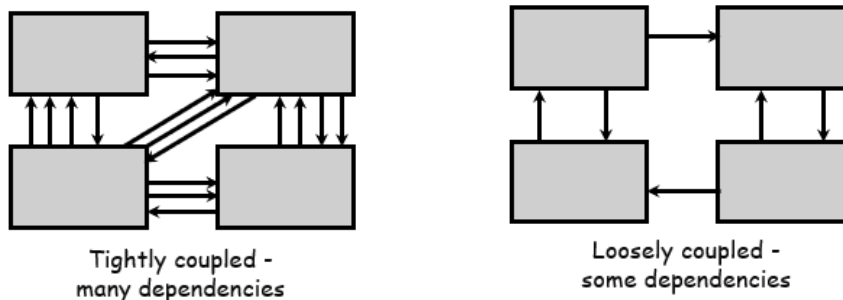
Een klant van een webwinkel vult een orderformulier in. Dit orderformulier wordt gelijktijdig verstuurd naar zowel de afdeling Orderverzending als de afdeling Financiën. De eerste afdeling pakt op basis van het orderformulier het pakket in en verstuurt het pakket. Als het pakket urgent is, dan wordt het met de snelpost verstuurd, anders met de gewone post. De afdeling Financiën print een factuur en stuurt deze naar de klant. Zodra de klant betaald heeft wordt de order afgesloten door de afdeling Financiën. De klant betaalt pas als het pakket ontvangen is.

8 OO DESIGN PRINCIPLES

8.1 Instapopdracht

Belangrijke kernbegrippen en principes bij het ontwerp van objectgeoriënteerde software zijn: *modularization*, *information hiding*, *loose coupling* en *high cohesion*. In het boek *Objects First with Java* van David Barnes en Michael Kölling worden deze begrippen als volgt gedefinieerd:

- **Modularization**
The process of dividing the whole into well-defined parts, which can be built and examined separately, and which interact in well-defined ways.
- **Information hiding**
A principle that states that internal details of class's implementation should be hidden from other classes.
- **Coupling**
The interconnectedness of classes. Loose coupling in a system means: each class is largely independent and communicates with other classes via a small, well-defined interface.



- **Cohesion**
How well a unit of code (method, class, module) maps to a logical task or entity. In a highly cohesive system each unit of code is responsible for a well-defined task or entity.

Deze principes dien je bij de onderstaande dobbelspelletjes toe te passen.

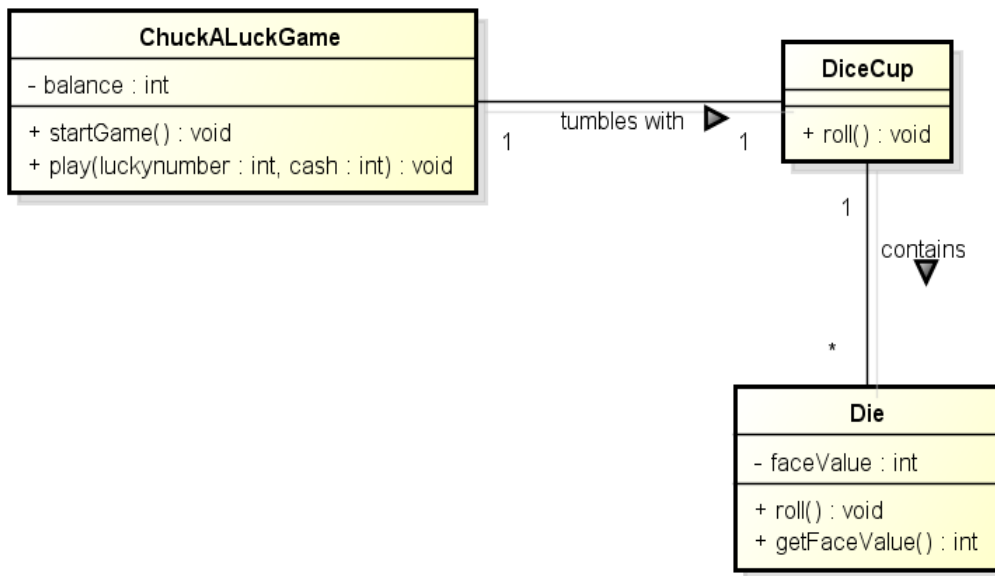
8.1.1 Chuck-a-luck

Beschouw de volgende eenvoudige versie van het spel Chuck-a-luck:

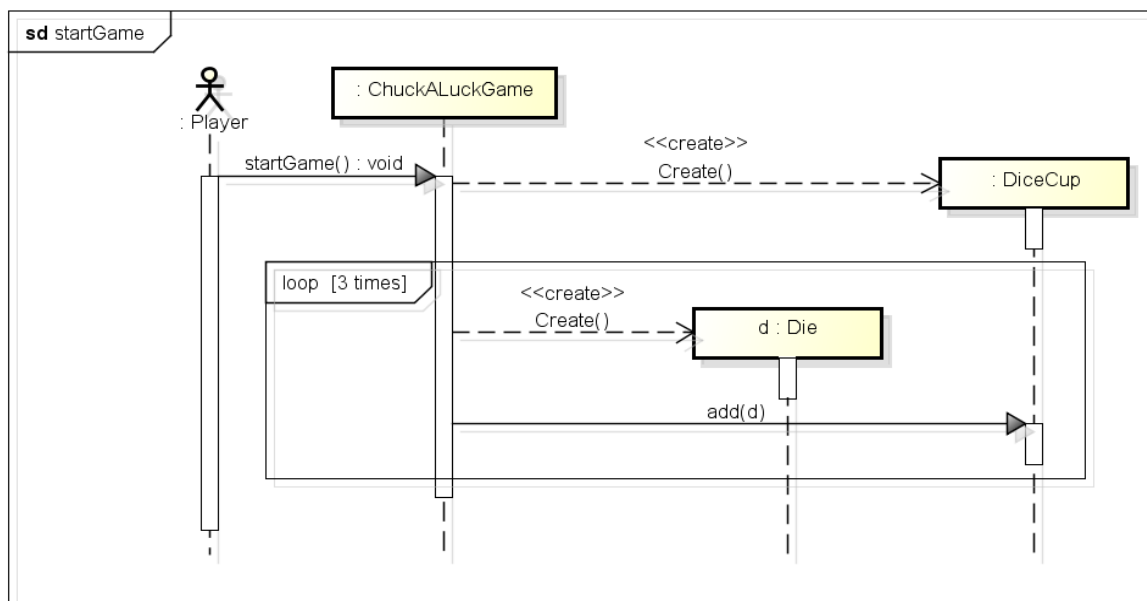
Per ronde zet je een bedrag in op een geluksgetal van 1 tot 6 en vervolgens gooi je drie dobbelstenen met behulp van een dobbelbeker. Als geen van de dobbelstenen dit geluksgetal aangeeft, dan ben je je inzet kwijt. In alle andere gevallen wordt uitbetaald afhankelijk van het aantal dobbelstenen dat het voorspelde aantal ogen weergeeft:

Overeenkomende dobbelstenen	Uitbetaling
1 (een Single)	1:1
2 (een Double)	2:1
3 (een Triple)	10:1

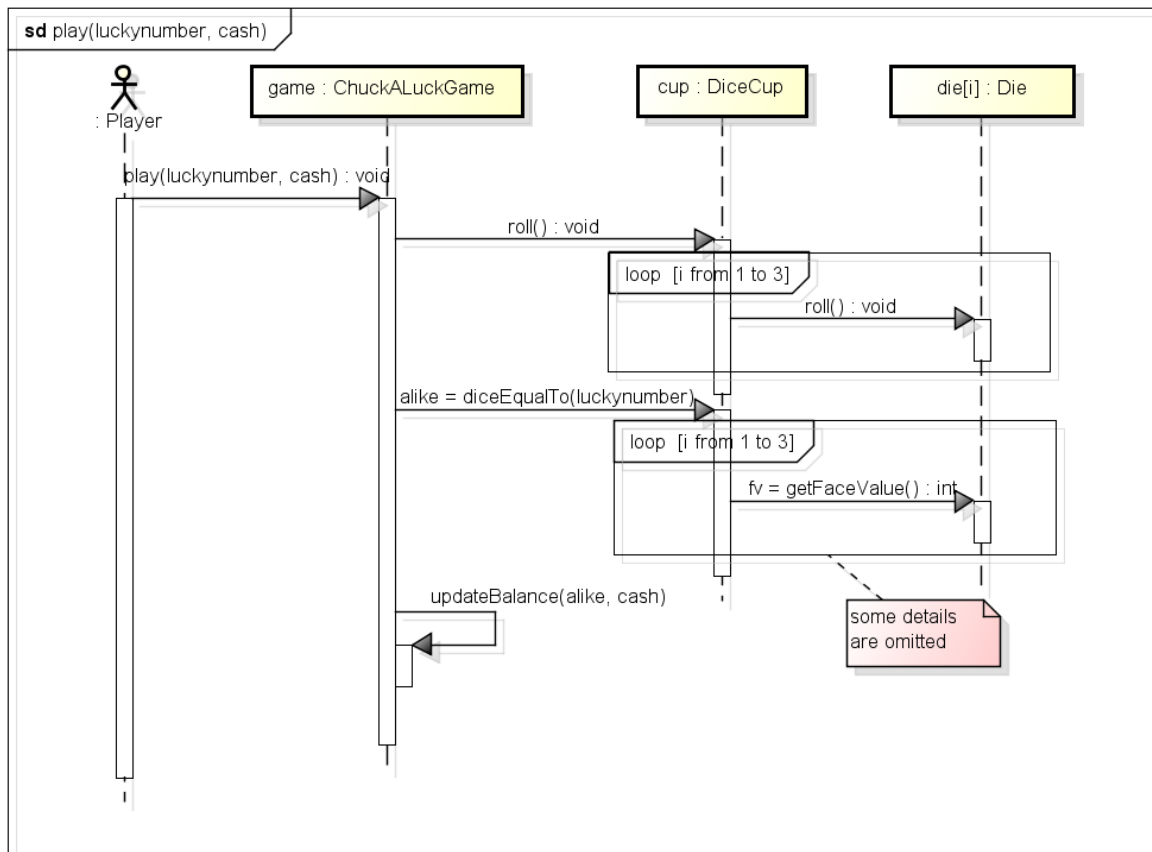
Een gedeelte van het ontwerp van een Chuck-a-luck applicatie is gemaakt. Hieronder zie je achtereenvolgens een nog incompleet design class diagram en sequence diagrams van de systeemoperaties *startGame* en *play*.



Figuur 1: Design class diagram



Figuur 2: Sequence diagram startGame



Figuur 3: Sequence diagram play

Toelichting: In het bovenstaande sequence diagram staat de parameter *luckynumber* voor het geluksgetal en *cash* voor de inzet.

Opdrachten:

- Geef de navigeerbaarheid van de associaties weer in het class diagram op basis van de gegeven sequence diagrams.
- Voeg dependencies, die je kunt afleiden uit de sequence diagrams van *startGame* en *play*, toe aan het class diagram.
- In het sequence diagram van *startGame* zijn verantwoordelijkheden niet altijd aan de juiste objecten/classes toegewezen? Geef aan hoe het beter kan.
- In het sequence diagram van *play* krijgt het *DiceCup*-object de verantwoordelijkheid om te bepalen hoeveel van geworpen dobbelstenen gelijk zijn aan het geluksgetal. Is dat een goed idee? Zo ja, leg uit waarom. Zo nee, aan welk ander object kun je deze verantwoordelijkheid beter toewijzen?
- Op de OOAD-site op #OnderwijsOnline vind je een geraamte voor de code voor het implementeren van Chuck-a-Luck. Maak deze code compleet en zoveel mogelijk in lijn met de opgestelde diagrammen.







8.1.2 Yahtzee

Bij deze opgave bekijken we een versimpelde versie van het spel Yahtzee. Hier volgt de beschrijving:

Het spel wordt door één speler gespeeld en deze maakt gebruik van 5 dobbelstenen. Daarnaast is er een scorekaart met daarop zes vakjes die corresponderen met het aantal ogen van de dobbelsteen. Alleen het bovenste gedeelte van een gebruikelijke Yahtzee-scorekaart (zie hieronder) wordt dus gebruikt.

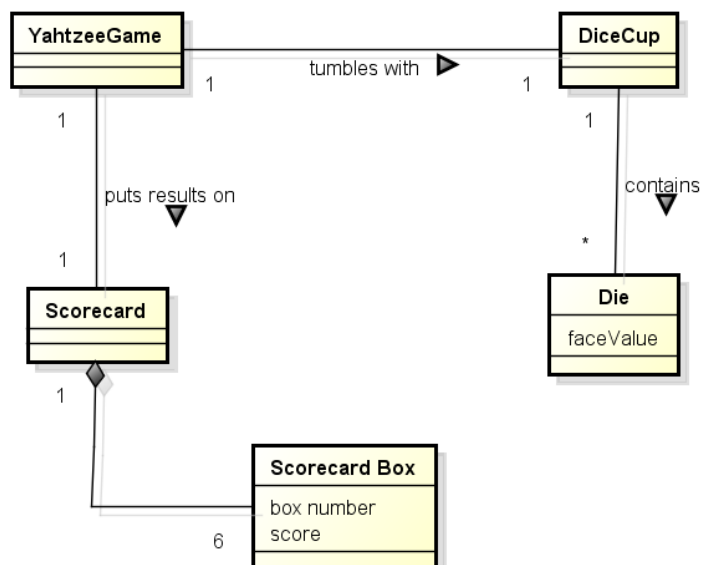
De speler werpt met vijf dobbelstenen. Het resultaat van de worp moet in een van de zes vakjes op de scorekaart worden genoteerd. Bijvoorbeeld, als je 5 3 3 5 3 werpt kun je 9 invullen in het vakje bij 3 of 10 in het vakje bij 5. Zo werp je zes keer waarbij de score steeds in een ander vakje dient te worden ingevuld. Zet je dus na de eerste worp 9 in het vakje bij 3, dan kan dit vakje bij een volgende worp niet meer worden gebruikt.

Bonus: Aan het eind van het spel krijg je een bonus van 35 punten als het totaal van de punten 63 of hoger is.

Yahtzee Player's Name _____ SCORE CARD						
UPPER SECTION	HOW TO SCORE	GAME #1	GAME #2	GAME #3	GAME #4	GAME #5
Aces  = 1	Count and add only Aces					
Twos  = 2	Count and add only Twos					
Threes  = 3	Count and add only Threes					
Fours  = 4	Count and add only Fours					
Fives  = 5	Count and add only Fives					
Sixes  = 6	Count and add only Sixes					
TOTAL SCORE	→					
BONUS If total score is 63 or over	SCORE 35					
TOTAL Of Upper Section	→					



Hieronder zie je een domeinmodel dat is opgesteld voor deze Yahtzee-variant.



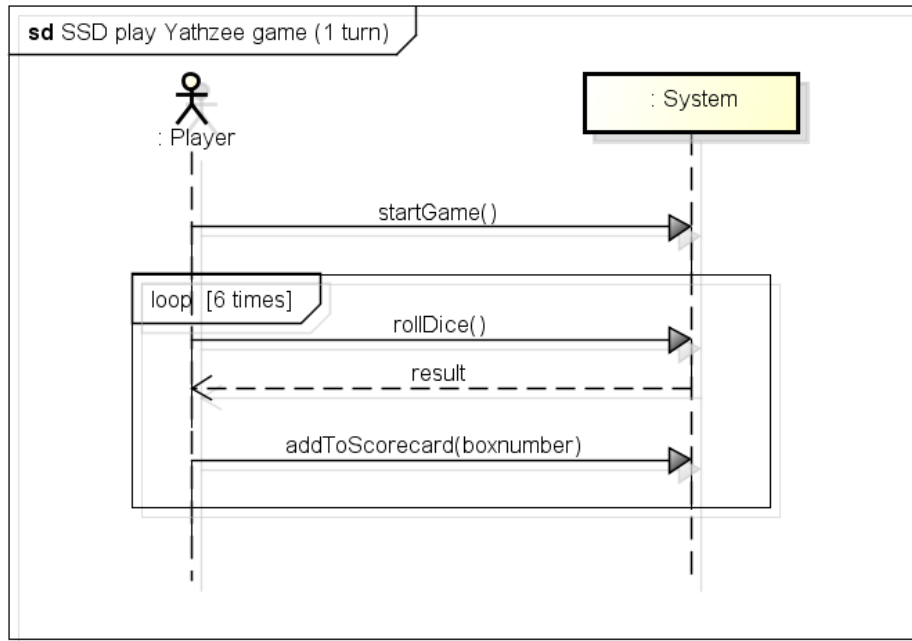
Een analist heeft een use case *Speel Ronde* geïdentificeerd.

Use case *Speel Ronde*, brief description.

Een speler gooit met 5 dobbelstenen en plaatst het resultaat van de worp in een van de vakjes van de scorekaart. Dit doet hij in totaal 6 keer waarna de scorekaart volledig is gevuld. Bij een score van 63 of hoger krijgt de speler een bonus van 35 punten.

Einde use case.

Bij deze use case is een system sequence diagram gemaakt.



Opdrachten:

- Stel sequence diagrams op voor de systeemoperaties *startGame*, *rollDice* en *addToScorecard*. Probeer dit zodanig te doen dat de koppeling tussen classes laag is en de cohesie van classes hoog.
- Welke ontwerpprincipes heb je bij onderdeel 1 gebruikt om verantwoordelijkheden aan objecten/classes toe te wijzen?
- In hoeverre kun je classes van het Chuck-a-luck spel hergebruiken?
- Maak een design class diagram.
- Op de OOAD-site op #OnderwijsOnline vind je een geraamte voor de code voor het implementeren van Yahtzee. Maak deze code compleet en zoveel mogelijk in lijn met de opgestelde diagrammen.

8.2 Zelfstudie en lesvoorbereiding

8.2.1 Kijkwijzer SOLID

SOLID is een acroniem voor de eerste vijf object-oriented design principes van Robert C. Martin. SOLID staat voor:

- Single responsibility principle
- Open closed principle
- Liskov substitution principle
- Interface segregation principle
- Dependency inversion principle

Kijk de course ***SOLID Software Design Principles in Java*** op PluralSight.

8.2.2 Leeswijzer GRASP

- GRASP Information Expert en Creator: [LAR]17.3 t/m 17.11.
- GRASP Low Coupling, High Cohesion en Controller: [LAR]17.12, 17.13 behalve pag 307 vanaf Web UIs and Server-Side Application of Controller tot pag 311 bovenaan, 17.14 behalve Yin and Yang.
- Koppelen van de UI layer aan de Domain Layer: [LAR] pag 345 t/m 347.

Toelichting

- De behandelde Design principles gelden voor ontwerpen in het algemeen, niet alleen voor OO design. Begrippen als information hiding, abstractie, modulariteit, cohesie en koppeling horen tot het vocabulaire van elke informaticus, op welk gebied ook werkzaam. Het is wel zo dat ze in een OO ontwerp makkelijker zijn toe te passen, maar dat neemt niet weg dat het toch best eenvoudig is een OO ontwerp te maken dat tegen deze begrippen zondigt. Zo zie je in de praktijk nog al eens dat er classes zijn met een zwakke cohesie: let op classes met veel responsibilities die weinig of niets met elkaar te maken hebben.
- De in OOAD behandelde patterns, Information Expert, Creator, Low Coupling, High Cohesion en Controller, zijn patterns waarvan de toepassing bevordert dat bovengenoemde design principles worden gevolgd. Lees daarvoor de secties bij Information Expert en Creator onder het kopje Benefits.
- In [LAR] chapter 18 wordt geen nieuwe theorie behandeld maar wordt wel het ontwerp van NextGen POS doorlopen. Het is aan te raden dit goed te bestuderen.
- In [LAR] pag 345 t/m 347 maak je voor het eerst kennis met het denken over modulariteit of architectuur: interface en domein worden op de juiste manier van elkaar gescheiden (low coupling, high cohesion, information hiding). De koppeling tussen UI-laag en domeinlaag wordt laag gehouden door de UI-laag geen domeinkennis te geven (want anders is de UI-laag moeilijk aan te passen). Er zijn andere patterns die dit bevorderen, zoals Observer.

8.3 Opdrachten OO Ontwerpprincipes

8.3.1 SOLID – Single Responsibility Principle

- A. Beschrijf in eigen woorden wat het Single Responsibility Principle inhoudt.
- B. Geef een voorbeeld (code of diagram) waarin dit principe overduidelijk niet wordt toegepast.
Geef daarbij aan tegen welke problemen je aanloopt.

8.3.2 SOLID – Open/Closed Principle

Het onderstaande codefragment is gebaseerd op het *video store* voorbeeld uit het boek *Refactoring* van Martin Fowler.

Je ziet in de onderstaande code dat de prijs (charge) die je betaalt voor de huur van een film afhankelijk is van het aantal dagen dat de film wordt gehuurd en het filmtype (regular, new release, children).

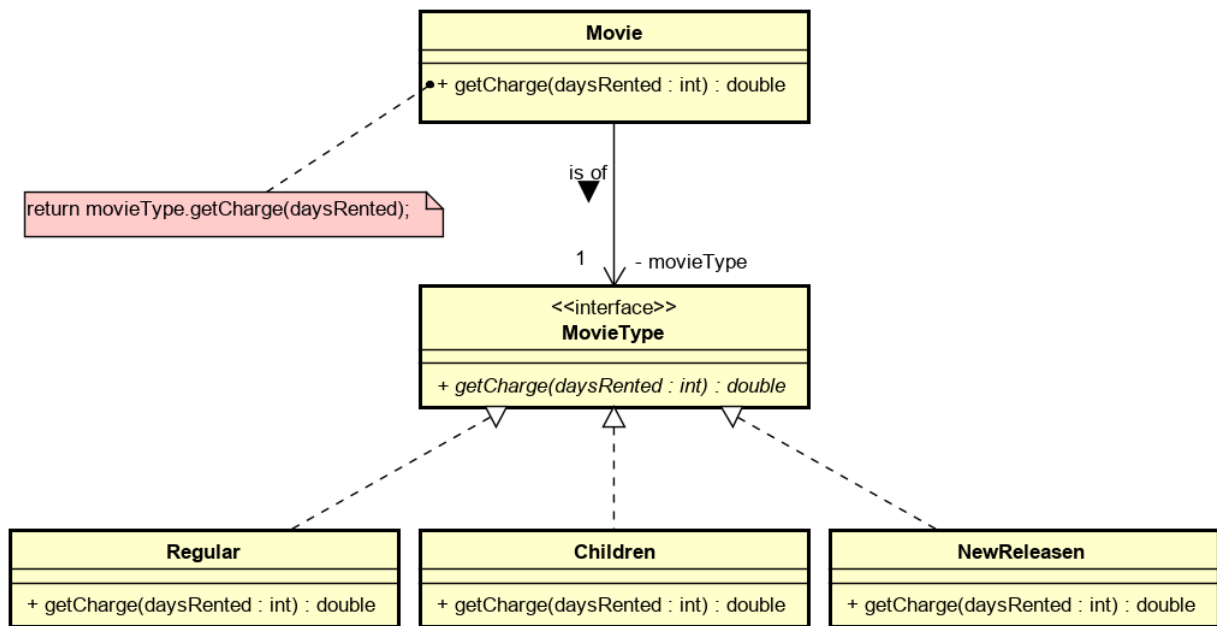
```
public class Movie {
    public static final int REGULAR = 1;
    public static final int NEW_RELEASE = 2;
    public static final int CHILDREN = 3;

    private String title;
    private int releaseyear;
    private int pricecode;

    public Movie(String title, int year, int pricecode) {
        this.title = title;
        this.releaseyear = year;
        this.pricecode = pricecode;
    }

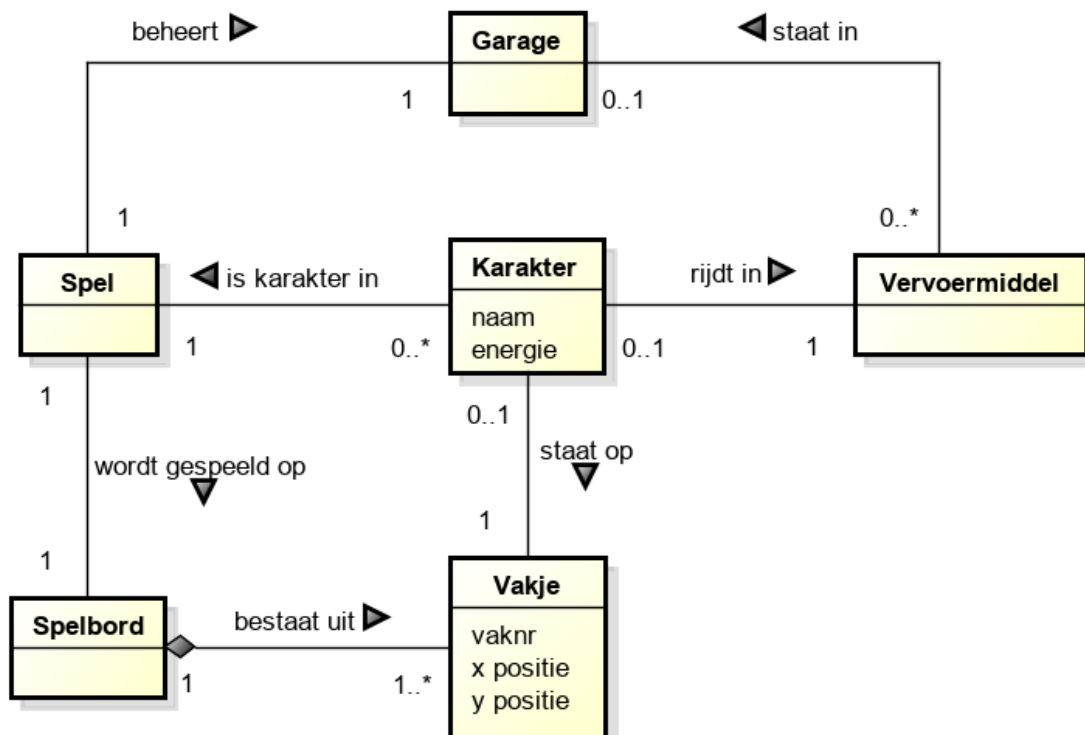
    public double getCharge(int daysRented) {
        double result = 0;
        switch (pricecode) {
            case Movie.REGULAR:
                result += 2;
                if (daysRented > 2)
                    result += (daysRented - 2) * 1.5;
                break;
            case Movie.NEW_RELEASE:
                result += daysRented * 3;
                break;
            case Movie.CHILDREN:
                result += 1.5;
                if (daysRented > 3)
                    result += (daysRented - 3) * 1.5;
                break;
        }
        return result;
    }
}
```

- A. Refactor de code naar de structuur die is weergegeven in het onderstaande class diagram.
Schrijf een main (of een unit test) om te controleren of het correct werkt.
Opmerking: Deze refactoring wordt ook wel *Replace Conditional With Polymorphism* genoemd.
- B. Wat zijn de voordelen van de gerefactorde code t.o.v. de oorspronkelijke versie?
- C. Leg uit dat de nieuwe versie voldoet aan het Open/Closed Principle.



8.3.3 AdventureQuest

Bij deze opdracht beschouwen we het spel AdventureQuest (zie opdrachten 4.2.2 en 4.2.3). Het bij opdracht 4.2.2 getoonde domeinmodel is uitgebreid met een garage waarin voertuigen staan. Verder zijn de coördinaten van de positie van het vakje op het spelbord toegevoegd in de vorm van de attributen x-positie (horizontaal) en y-positie (verticaal) van Vakje.



Opdrachten:

- A. Programmeer een nieuwe versie van de systeemoperatie **meldAan** waarbij het nieuwe karakter een vervoermiddel uit de garage krijgt. Pas daarbij OO design principles, Maak vervolgens een nieuwe versie van het sequence diagram van **meldAan** en pas het design class diagram aan.
- B. Verbeter, indien nodig, de code en de sequence diagrams van **teleporteur** en **wisselVakje** (zie opdracht 4.2.3) gebruikmakend van OO design principles.
Vermeld bij beide systeemoperatie welk principe waar is toegepast en waarom.
- C. Programmeer de systeemoperatie **ruilVervoermiddel** en teken het bijbehorende sequence diagram. Hieronder zie je de pre- en postcondities bij deze systeemoperatie. Houd daarbij rekening met het Single Responsibility Principle en meer specifiek Information Expert.

Precondities van ruilVervoermiddel (naamKarakter1, naamKarakter2):

Vóórdat deze operatie uitgevoerd wordt bestaat de volgende situatie:

- Er zijn al instanties van Spel en Spelbord aangemaakt; ook alle instanties van Vakje bestaan al.
- Er is een karakter *k1* met de naam *naamKarakter1* en een karakter *k2* met de naam *naamKarakter2*.
- Karakter *k1* bevindt zich op vakje *vk1*, karakter *k2* bevindt zich op vakje *vk2*.
- Karakter *k1* rijdt in vervoermiddel *vm1*, karakter *k2* rijdt in vervoermiddel *vm2*.

Postcondities:

Nadat de operatie uitgevoerd is moet het volgende gelden:

ALS vakje *vk1* grenst aan vakje *vk2*

DAN Karakter *k1* rijdt in vervoermiddel *vm2*, karakter *k2* rijdt in vervoermiddel *vm1*.

8.3.4 SOLID – Dependency Inversion Principle

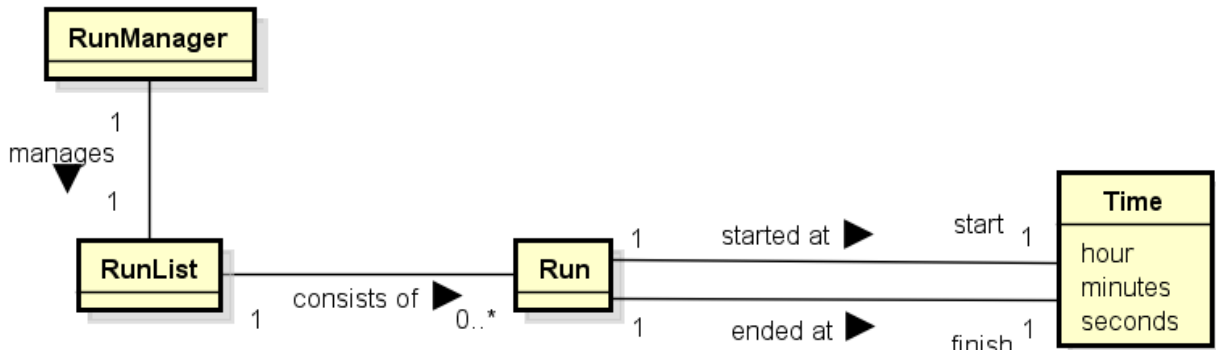
<< Hier wordt nog een opgave ingevoegd >>

8.3.5 AdventureQuest vervolg

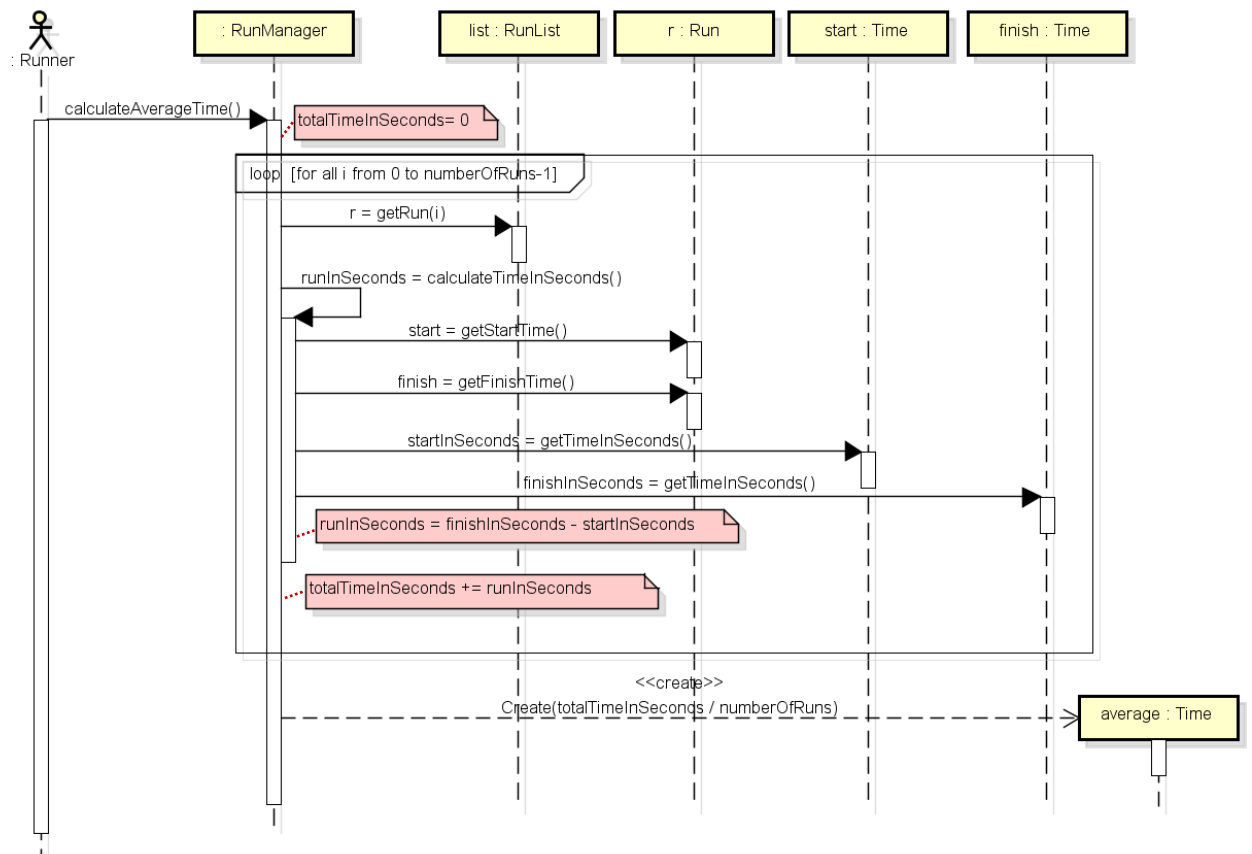
- A. Tot nu toe wordt overal in de code rechtstreeks tegen de klasse 'Karakter' geprogrammeerd. We willen het spel gaan uitbreiden met vijanden en andere personages. Wat zijn hierbij nadelen van de huidige code?
- B. Vanaf nu programmeren we niet meer rechtstreeks tegen de klasse 'Karakter'. Maak een interface die door Karakter wordt geïmplementeerd en pas je code hierop aan zodat deze overal wordt gebruikt.

8.3.6 Hardloper

Een hardloper loopt elke week hetzelfde rondje. Van elke keer dat hij het rondje loopt (dat wordt een *run* genoemd) slaat hij de start- en finishtijd op. Daarvoor is het volgende domeinmodel opgesteld.



Hieronder zie je een sequence diagram voor de operatie **calculateAverage()**, voor het berekenen van de gemiddelde tijd van de runs. Daarin zijn assignment statements als notes weergegeven.

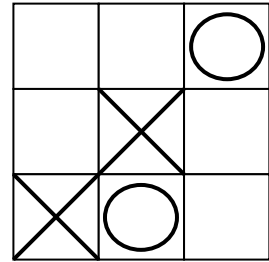


Opdrachten:

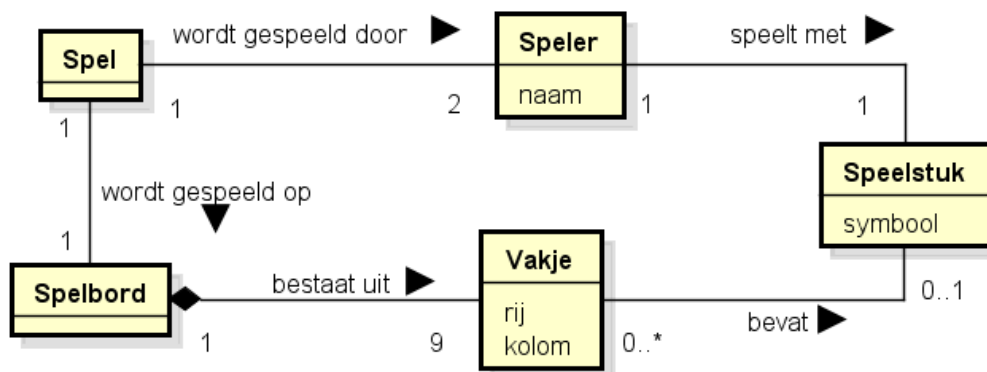
- Vind je dit een goed sequence diagram? Motiveer je antwoord.
- Maak het design class diagram dat hierbij hoort. Geef daarin ook alle dependencies aan.
- Maak een sequence diagram dat het aantal dependencies minimaliseert. Welke OO design principles pas je daarbij toe? Licht de gemaakte keuzes toe.

8.3.7 Boter Kaas en Eieren

Het spel Boter, Kaas en Eieren wordt gespeeld tussen twee spelers op een bord van drie bij drie vierkanten. Een van de spelers zet het bord en de stukken klaar. De speler die begint ('PlayerX') kiest een vierkant van het bord en zet er een kruis in. De ander speler ('PlayerO') kiest een leeg vierkant en zet er een rondje in. Daarna plaatsen de spelers om de beurt hun stuk in lege vierkanten. De eerste speler die drie van zijn merktekens op een rechte lijn (horizontaal, verticaal, of diagonaal) weet te plaatsen is de winnaar. Als het bord gevuld is zonder dat een speler dit voor elkaar heeft gekregen, is het gelijkspel.



Bij dit spel is het volgende domeinmodel opgesteld.



Tijdens analyse van het spel is een system operation **doeZet (naam, x, y)** gevonden, waarvan hieronder het contract volgt.

Name: doeZet (naam, x, y)
 Responsibilities: Het speelstuk van de speler in het beoogde vakje plaatsen.
 Pre-conditions:

- Er bestaat een Speler splr met de gegeven naam.
- Het Vakje vk dat correspondeert met de gegeven waarden x en y (vk.kolom = x, vk.rij = y) is leeg
- Speler splr speelt met Speelstuk stk

Post-conditions: Het Vakje vk bevat het Speelstuk stk.

Opdracht

- Maak een sequence diagram voor de system operation **doeZet (naam, x, y)**. Pas OO design principles toe waar mogelijk en motiveer de beslissingen die je maakt. (Pas zonodig het domeinmodel aan).
- Vorm een groepje van drie en vergelijk de uitwerkingen van het sequence diagram van opdracht 1. Probeer het met elkaar eens te worden over welke oplossing de beste is en geef aan waarom.

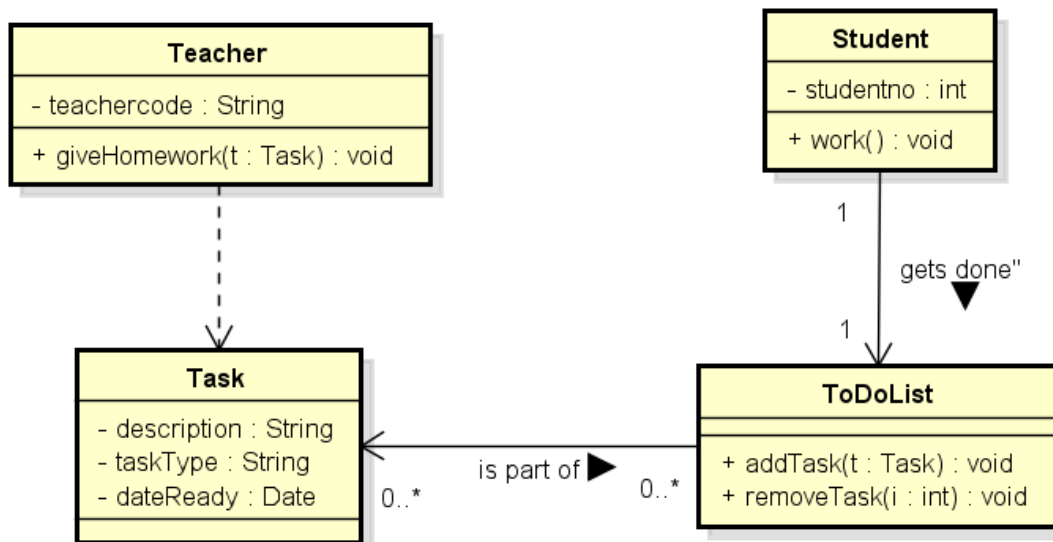
9 GOF DESIGN PATTERNS: OBSERVER, STRATEGY EN STATE

9.1 Instapopdracht

Casus Huiswerk maken

Een docent wil natuurlijk graag dat het huiswerk dat hij geeft z.s.m. door de studenten wordt opgepakt, d.w.z. dat studenten de huiswerktaken toevoegen aan hun to do list. Als een student een taak heeft afgerond, dan verwijdt hij die van zijn to do list. Er zijn twee soorten taken: leeswerk en maakwerk.

Voor de huidige situatie is het volgende design class diagram opgesteld.



Opmerking: Het attribuut taskType van de class Task kan de volgende waarden aannemen: "read" en "do".

Vraag: Hoe regel je dat zodra een docent een taak als huiswerk opgeeft, studenten deze ook direct op hun to do list zetten?

Idee: Laat studenten zich aanmelden bij de docent zodat zij ervan op de hoogte worden gebracht als er huiswerk wordt gegeven. Indien nodig kunnen ook andere personen dan studenten (bijvoorbeeld een directielid dat wil volgen wat een docent doet) zich aanmelden bij de docent.

Opdracht 1

Maak gebruik van het Observer Pattern (zie voorbeeld op OnderwijsOnline) om het bovenstaande idee vorm te geven.

- Breid het design class diagram uit met classes/interfaces, attributen, methoden en associaties die nodig zijn om het hiervoor beschreven idee vorm te geven.
- Teken ook een sequence diagram dat beschrijft wat er precies gebeurt als de methode `giveHomework` wordt uitgevoerd.

Studenten hebben verschillende werkwijzen bij het maken van het huiswerk. De volgende werkwijzen zijn bekend:

- het huiswerk dat als eerste is opgegeven huiswerk wordt als eerste gemaakt ('FIFO');

- alleen het maakwerk op de to do list wordt gedaan; het leeswerk wordt van de to do list verwijderd zonder het te hebben uitgevoerd ('alleen maakwerk');
- alleen de eerste taak op de to do list wordt gedaan, de overige taken worden van de to do list verwijderd ('liever lui dan moe');
- de taken op de to do list die morgen klaar moeten zijn worden gemaakt ('doe wat morgen af moet zijn').

Wie weet komen er in de toekomst nog nieuwe werkwijzen bij. Verder kan de werkwijze van een student in de loop van de tijd veranderen, bijvoorbeeld een student die eerst het 'alleen maakwerk'-principe toepaste en die nu overstapt op het FIFO-principe.

Vraag: Hoe kun je de hierboven beschreven werkwijzen in het class diagram verwerken zodat het maken van het huiswerk volgens een bepaalde werkwijze zo flexibel mogelijk is en waarbij het eenvoudig is om nieuwe werkwijzen toe te voegen?

Opdracht 2

Formeer een groepje van twee of drie personen. Breid als groep het design class diagram uit met classes/interfaces, attributen, methoden en associaties die nodig zijn als antwoord op de bovenstaande vraag.

Opdracht 3

Werk de ontwerpen die je bij opdrachten 1 en 2 heb gemaakt uit in Java-code. Maak gebruik van de startcode op OnderwijsOnline (zie het bestand **startcode StudentTeacher.rar**).

9.2 Zelfstudie en lesvoorbereiding

Kijkwijzer Design Patterns

Bekijk op PluralSight de volgende modules van de course **Design Patterns in Java: Structural**:

- Adapter Design Pattern (zie ook opdracht 9.2.1A)
- State Pattern (zie ook opdracht 9.2.1B)

Leeswijzer Design patterns

STRATEGY PATTERN

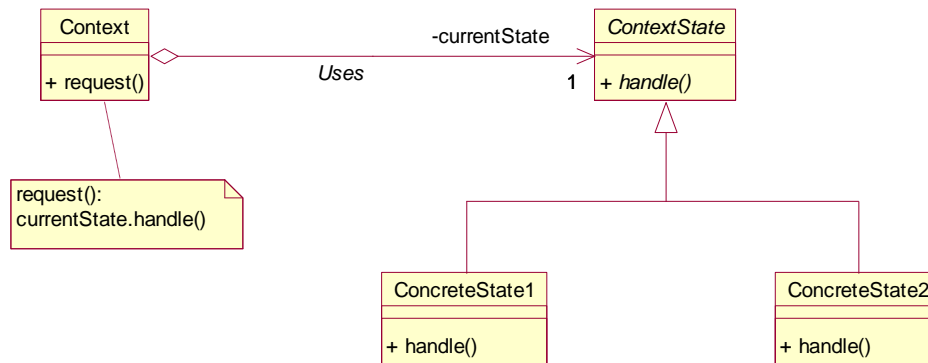
- Lees [LAR] paragraaf 26.7.

STATE PATTERN

- Een definitie (probleem/oplossing) staat in [LAR] pag 644. Het pattern wordt daar behandeld in het kader van het ontwerp van een persistence framework, wat op zich geen onderwerp van deze course is. Maar in [LAR] fig. 37.14 zul je het pattern herkennen. Een persistent object kan hier in één van vier verschillende toestanden zijn die gemodelleerd zijn als subclasses van een abstracte superclass PObjectState. Als een PersistentObject een commit moet doen, dan delegeert hij dat aan zijn huidige state. Als het persistent object bijvoorbeeld in state OldDirtyState is dan doet die state de commit. Is hij echter in OldCleanState, dan gebeurt er niets: de commit() van de super class

PobjectState wordt uitgevoerd, maar die doet in feite niets. Je ziet dat er heftig gebruik is gemaakt van polymorfisme, wat altijd gedaan wordt bij toepassing van het State Pattern.

State Pattern solution:



Dit is de vorm die in de literatuur meestal wordt gegeven. Zoals bij alle design patterns kan afhankelijk van het systeem waar je het pattern wilt toepassen er nogal wat aan gesleuteld worden.

Toelichting:

Context class

Instanties van deze class vertonen stateful gedrag. Een instantie kent zijn huidige state (currentState: ContextState) omdat hij een referentie heeft naar een instantie van een concrete subclass van de ContextState class.

ContextState class en ConcreteState classes

De concrete ConcreteState classes zijn subclasses van de abstracte class ContextState en implementeren de handle method.

- STATE pattern gevolgen: de code voor elke state zit alleen in zijn eigen class. Zo kunnen makkelijk nieuwe states worden toegevoegd. Er zijn geen ingewikkelde If-statements meer nodig.

9.3 Opdrachten GoF Design Patterns

9.3.1 PluralSight examples

De volgende vragen hebben betrekking op de PluralSight course **Design Patterns in Java: Behavioral**.

- In de module **Strategy Pattern** wordt als voorbeeld de *validation strategy* van een creditcard besproken.
Maak een class diagram bij dit voorbeeld waarin de toepassing van het pattern duidelijk is te zien.
- In de module **State Pattern** wordt een voorbeeld besproken dat gaat over de states van een ventilator (Engels: fan).
Maak een class diagram bij dit voorbeeld waarin de toepassing van het pattern duidelijk is te zien.

9.3.2 AdventureQuest

Bij deze opdracht ga je het Strategy Pattern en daarmee ook de S en de O van SOLID toepassen bij AdventureQuest.

- A. Het spelbord kan verschillende vormen hebben. Pas het Strategy Pattern toe om het spelbord uit 1 rij, 2 kolommen of 4 rijen te laten bestaan. Het bord bestaat altijd uit 64 vakjes.
De Strategy-interface dient in ieder geval een methode

```
public Vakje getVakje(int i);
```

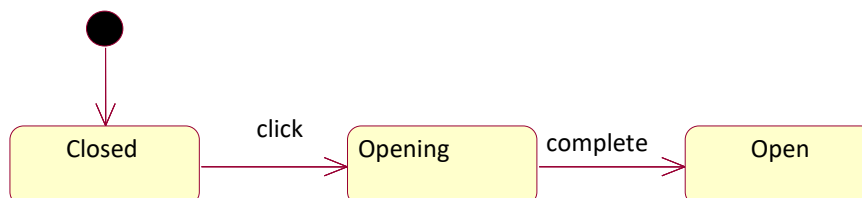
te hebben.
- B. Een karakter kan gewond zijn, extra krachtig zijn, of normaal. Als een karakter gewond is dan kost iedere beweging de volle 20 energie. Normaal kost dat 10 energie en met extra kracht maar 2. Een karakter gaat van gewond naar normaal door een appel te eten en van normaal naar gewond door een schok te krijgen. Een karakter gaat van normaal naar extra krachtig door een appel te eten. Als een karakter extra krachtig is dan levert een appel 20 extra energie op. Een karakter blijft 2 acties extra krachtig en wordt daarna weer normaal. Een schok doet niets bij een extra krachtig persoon, maar levert 10 energie op bij een gewond persoon. Pas het State Pattern toe om dit te realiseren.

9.3.3 Slimme deur in fabriek

In een fabriek worden materialen opgeslagen in een ruimte via een lopende band en een slimme deur. De deur wordt bediend met één enkele button.

Als de deur gesloten is en er wordt op de knop gedrukt, dan begint hij zich met een constante snelheid te openen. Als je nog een keer drukt vóórdát de deur volledig geopend is, dan begint de deur zich met een constante snelheid te sluiten. Als de deur open is begint hij zich weer te sluiten na een time-out van 2 seconden. Klik je dan vóórdát de deur gesloten is, dan gaat hij zich weer openen.

- A. Vul het onderstaande state machine diagram van de deur aan.



- B. Maak een class diagram van class Door en zijn mogelijke states door het STATE pattern toe te passen. Maak het diagram zo compleet mogelijk (o.a. methoden en associaties met multipliciteiten en navigatie).

10 GOF DESIGN PATTERNS: FACTORY METHOD, ADAPTER

10.1 Zelfstudie en lesvoorbereiding

Kijkwijzer Design Patterns

Bekijk op PluralSight de volgende modules:

- **Adapter Pattern** van de course **Design Patterns in Java: Structural**
 - **Factory Method Pattern** van de course **Design Patterns in Java: Creational**

Leeswijzer Design patterns

Het ADAPTER Pattern

- Lees [LAR] 26.1 t/m 26.3.

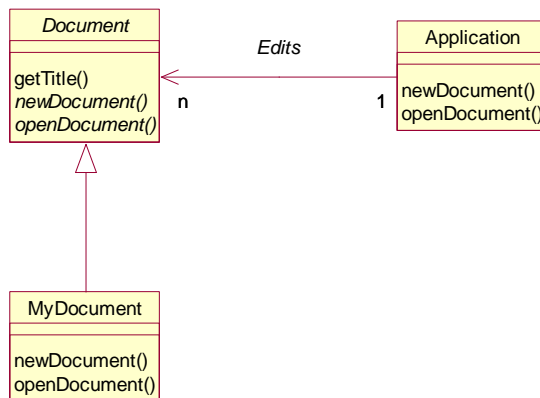
Het FACTORY METHOD Pattern

- Als je Creator toepast heb je bepaalt welk object een instantie van een ander object mag aanmaken. Toepassing van Creator ondersteunt low coupling, want de koppeling wordt niet verhoogd omdat de gecreëerde class waarschijnlijk al zichtbaar was voor de creator, via een bestaande associatie.

Maar door zo'n navigeerbare associatie tussen creator en created is de creator wel afhankelijk van de gecreëerde class: een navigeerbare associatie is in feite een dependency relatie. Op zich is dat niet erg: een OO-systeem werkt omdat objecten elkaar boodschappen sturen, dus een bepaalde koppeling en afhankelijkheid moet er altijd bestaan.

Soms kan die afhankelijkheid echter te ver gaan en dan is FACTORY METHOD het overwegen waard.

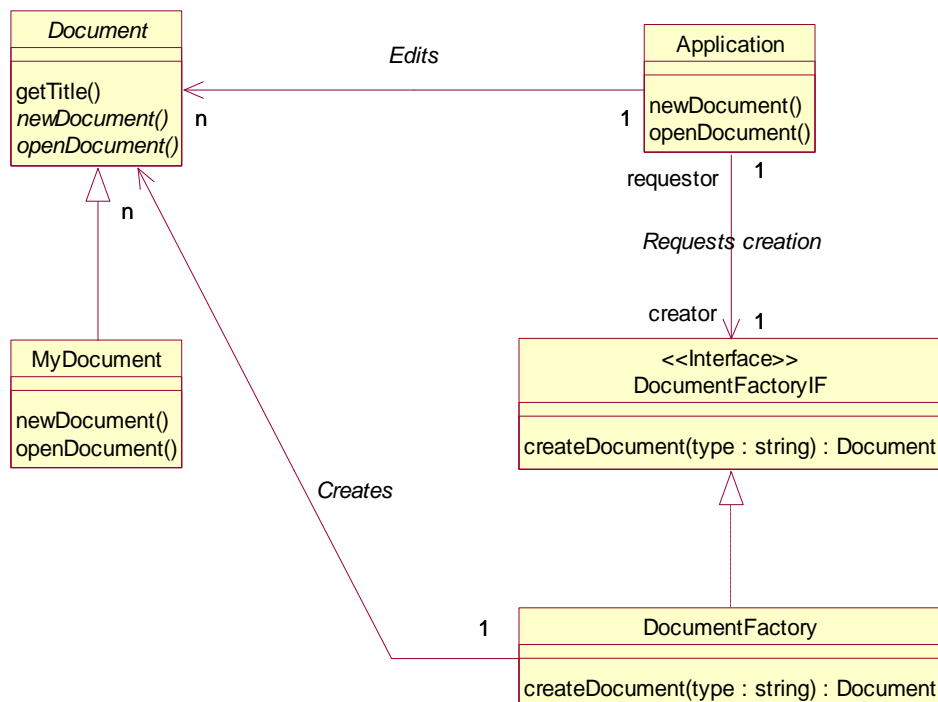
- Lees [LAR]26.4
In deze paragraaf wordt het FACTORY METHOD (of CONCRETE FACTORY) pattern gebruikt om adapters te bouwen, zoals de adapters getoond in [LAR] fig. 26.1. Er wordt een ServicesFactory gebruikt die dit doet. Als dit niet was gedaan, dan was waarschijnlijk een domeinobject als Register de klos geweest en was afhankelijk geworden van al die adapters en, bovendien, had die een lagere cohesie gekregen: een domeinobject moet domeindingen doen en zich niet druk te hoeven maken over design classes als adapters.
- Nog een voorbeeld:
Stel dat je een framework voor desktopapplicaties moet maken. Dat wil zeggen er moet een systeem komen dat tekstdocumenten, spreadsheets, presentaties etc. kan aanmaken, bewaren en openen. Voor het gemak nemen we aan dat er een class Application is die dat verzorgt.
De meeste logica om die operaties te implementeren varieert per type document, en dan delegeert Application die verantwoordelijkheden naar een of ander document object. Maar sommige operaties zijn weer gemeenschappelijk voor elk type, zoals bijvoorbeeld het tonen van de titel van het document. Je zou daarom kunnen kiezen voor een abstracte class Document, die applicatie-onafhankelijk is en subclasses daarvan voor de concrete soorten documenten.



Application is hier de Creator. Maar om een document van een bepaald type aan te maken moet Application nu kennis hebben van dat type, met andere woorden het heeft een dependency van elk type document. Er is een feite een koppeling tussen Application en een tekstdocument, een spreadsheetdocument, enzovoorts.

De uitdaging is nu die koppeling te verlagen. Dat kan bereikt worden door het werk te delegeren aan een class die die koppeling wél mag hebben, omdat hij specifiek voor die taak gemaakt is: een DocumentFactory. Immers als je aan een fabriek doorgeeft welk product je wil hebben, dan zorgt die fabriek daar wel voor en heb je zelf geen zorgen meer voor de bouw van die produkten. Application vraagt als requestor om een bepaald document, en dat doet hij door een simpele string als parameter door te geven. Hij wordt zo afhankelijk van die Factory en moet de juiste strings doorgeven, maar hij is niet meer afhankelijk van elk product. De Factory is een helper object met een sterke cohesie.

De afhankelijkheid kan nog verder verlaagd worden door een *interface* te gebruiken, die de DocumentFactory moet implementeren.



10.2 Opdrachten

10.2.1 AdventureQuest

Bij deze opdracht ga je de patterns Factory Method (en daarmee ook de S en de O van SOLID) en Adapter (en daarmee ook de S en de D van SOLID) toepassen bij AdventureQuest.

- A. Een karakter kan als vervoermiddel toegewezen krijgen: een auto, een fiets, een scooter of een step. Dit zijn vier verschillende implementaties van een vervoermiddel. Gebruik het Factory Method Pattern om vanuit de Garage de vervoermiddelen te creëren.
- B. Er is gekozen om gebruik te maken van een klasse 'BelangrijkVoorwerp', die door een externe partij wordt geleverd. We mogen deze code niet wijzigen. Maak gebruik van het Adapter Pattern om deze klasse ongewijzigd te kunnen gebruiken.

10.2.2 Koenen en Kramers

Doel van deze opdracht is het toepassen van een combinatie van de design patterns Adapter en Factory Method in een software ontwerp en dit vervolgens zelf realiseren. Stel, een programmeur moet een client class **TranslateToDutch** schrijven die een woord uit het Engels naar het Nederlands kan vertalen. Hij wil natuurlijk gebruik maken van een bestaande dictionary als die er is. Voorlopig maakt hij zich niet druk om een specifieke dictionary. Hij gaat ervan uit dat een dictionary class wel een methode **translate(String word): String** zal hebben en dat hij de naam van de dictionary te weten kan komen via een methode **getName(): String**.

Er worden hem gelukkig door andere ontwikkelaars twee goede dictionary-classes beschikbaar gesteld; een volgens Koenen en een volgens Kramers. Als **TranslateToDutch** een verzoek tot vertaling van een woord krijgt laat hij gewoon een dictionary aanmaken. Als een woord niet in de **KoenenDictionary** voorkomt moet de andere worden geprobeerd. Als hij het woord niet vindt geeft hij dat aan; vindt hij het woord wel, dan geeft hij de vertaling en de bron weer. Helaas kennen de classes geen methode **translate()**. Koenen zoekt met een methode **lookup(String word): String** en Kramers met **find(String word): String**. Je mag de sourcecode en dus de interface van beide classes niet aanpassen.

- A. Maak een UML design class diagram uitgaande van bovenstaande eisen. Pas Adapter, Factory Method en Singleton toe naar analogie van het voorbeeld in [LAR]26.1 t/m 26.5.
- B. Bekijk de broncode van de KoenenDictionary en KramersDictionary maar neem de broncode *niet* op in je project. In plaats daarvan voeg je een Maven-dependency toe:

```
<dependency>
  <groupId>nl.oose.ooad.koenenkramers</groupId>
  <artifactId>dictionaries</artifactId>
  <version>1.0</version>
</dependency>
```

Voeg ook in je pom.xml een extra repository-sectie aan om de dependency te kunnen downloaden:

```
<repositories>
  <repository>
    <id>github-rody</id>
```

```
<url>https://raw.githubusercontent.com/ddoa/workshop-koenen-kramers/master/</url>  
<snapshots>  
  <enabled>true</enabled>  
  <updatePolicy>always</updatePolicy>  
</snapshots>  
</repository>  
</repositories>
```

<<TO DO: Maven-dependencies checken>>

- C. Door deze library te gebruiken kun je niet meer de broncode aanpassen en is het noodzakelijk te werken met het Adapter-design pattern. Pas dit nu toe in de broncode samen met het Factory Method-design pattern.