Technical University of Denmark

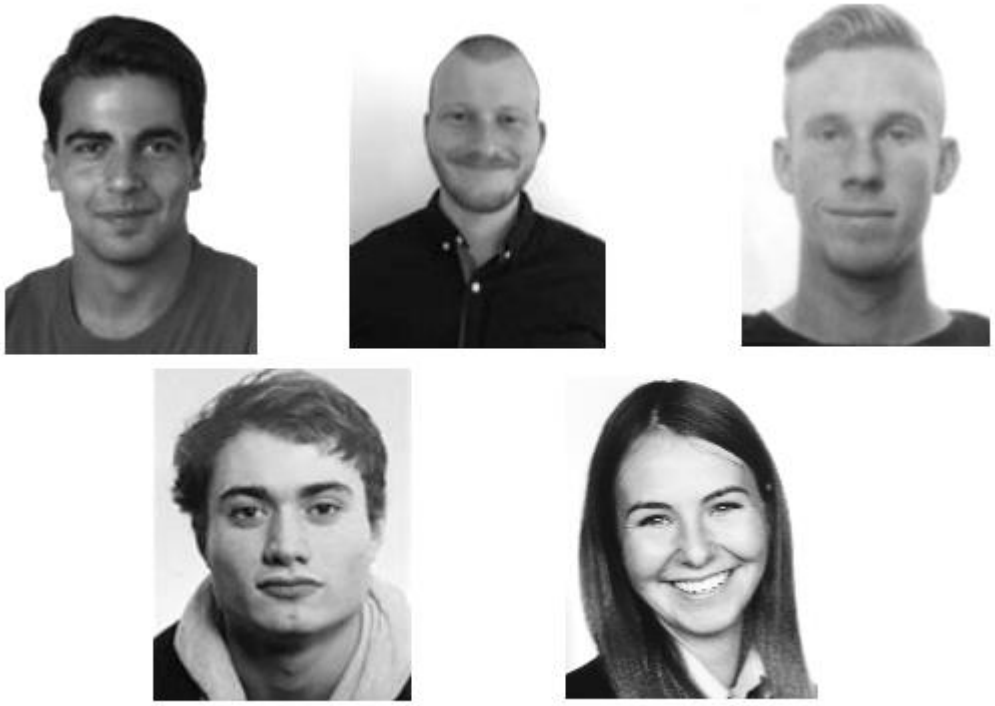
CDIO 2. Del

Hold A Gruppe 25

Fag:
02312, 02314, 02315

Afleveringsfrist 10-11-2017

Nicolas Vahman (s164404), Emil Vinkel (s175107) og Rasmus Hjorth (s175120) Simon Boye(s175118) & Simone Zethof (s175110)



Timeregnskab for CDIO 2. del

Dato	Deltager	Design	Impl.	Test	Dok.	Ialt
10/11/2017	Simon Boye	2	1		3	6
	Nicolas Vahman	2	3		1	6
	Emil Vinkel		3	1	2	6
	Simone Zethof		1	3	2	6
	Rasmus Hjorth		4	1	1	6

Abstract

The objective of this assignment was to create a board game, with the knowledge gained from three different courses. We act as employees from the company IOuterActive and were assigned to the development of a board game with some specific requirements given by a customer. The board game is an expansion of our previous CDIO project and as such it was more demanding to develop. An important factor of the development process, were that we had to work together as a group, representing a real-life work situation. During the development process we took advantage of the usage from the different types of diagrams from the Unified Modelling Language. Furthermore, we employed Unified Process as our main development method.
The coding process is primarily based on our knowledge obtained in the course Introductory programming. The test conducted is a JUnit test. This whole development process leads to our conclusion were we successfully achieved full implementation of the requirements given by the customer.

Indholdsfortegnelse

1.0 Indledning

Vi har i denne del af vores afleveringsopgaver arbejde med CDIO 2, som endnu engang skal være et terningspil. I det nedenstående, har vi udarbejdet en rapport, som vil beskrive spillet. Rapporten indeholder vores idéer til udvikling af spillet, og hvilke metoder vi har tænkt over og brugt. I sidste del af rapporten, vil vi vise vores test af programmet, som er udført ved en JUnit test, hvilket er blevet anbefalet at udføre i opgavebeskrivelsen. Opgaven er således et samspil mellem fagene, indledende programmering, udviklingsmetoder til IT-systemer og versionsstyring og testmetoder.

Det spil som vi har lavet i CDIO 2 er et terningspil, hvor, der ligesom i CDIO 1 er to spillere som skal spille mod hinanden. Hver spiller starter med 1000 point, hvorefter spilleren slår med to terninger, og det antal spilleren slår, er det antal spilleren rykker på spillebrættet. Hvert felt, som spilleren potentielt kan lande på, indeholder point, som kan tilføres eller fratrækkes spilleren pengebeholdning. Spillet slutter når en spiller når 3000 point. Spillet i CDIO 1 er et terningspil, hvor to spillere skal kunne spille imod hinanden. Selve spillet er hvor 2 spillere skiftes til at slå med to terninger. Den spiller der først når summen 40 point vinder.

2.0 Analyse

2.1 Formålet

Formålet med denne rapport er at udarbejde et terningspil skrevet i JAVA kode på baggrund af de analyseværktøjer vi er blevet introduceret til i Unified Modelling Language. Vi bruger altså de modelleringer til at skabe overblik over, hvad problemet skal kunne og hvordan det skal kunne det. Dette er særligt vigtigt, da det er en god metode, at kunne skitsere programmet uden noget kode, hvorefter at skitserne danner grundlag for programmeringsfasen. Programmeringsfasen skrives på baggrund af undervisningen i faget 'Indledende programmering'.

I opgavebeskrivelsen er det anbefalet, at der bliver udført en JUnit Test, som det sidste step i udviklingsfasen. Denne JUnit test er lavet på baggrund af undervisningen i 'Versionsstyring og Testmetoder'.

2.2 Kravspecifikation:

Vi har fra kunden fået oplyst en kravliste, som vi skal have implementeret i vores spil. Nedenstående er kravene fra kunden opgivet på listeform:

Kravliste**:**

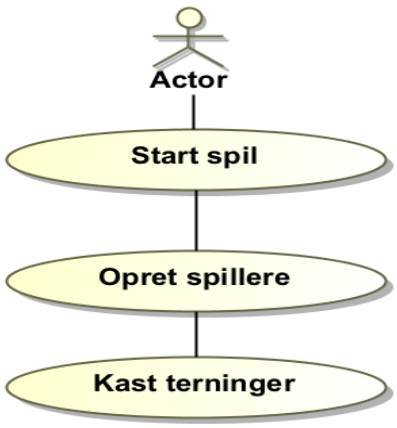
1. Spillerne skal på skift slå med 2 terninger og skal lande på et felt med numrene fra 2 - 12.
2. At lande på hvert af disse felter har en positiv eller negativ effekt på spillernes pengebeholdning.
3. Der skal udskrives en tekst, der omhandler det aktuelle felt.
4. Spillerne starter med en pengebeholdning på 1000.
5. Spillet er slut når en spiller når 3000.
6. Spillet skal let kunne oversættes til andre sprog.
7. Det skal være let at skifte til andre terninger.
8. Det skal være et spil som skal spilles mellem 2 spillere.
9. Det skal kunne spilles, uden større fejl på DTU's computere.

Nedenstående feltliste viser, de forskellige navne på felterne på spillepladen. Dernæst ses, hvordan hvert felt påvirker spillerens pengebeholdning. Felt 10 er specielt, da det giver spilleren en ekstra tur, men samtidigt trækker 80 point fra spillerens pengebeholdning.

Feltliste

1. (Man kan ikke slå 1 med to terninger)
2. Tower +250
3. Crater -100
4. Palace gates +100
5. Cold Desert -20
6. Walled city +180
7. Monastery 0
8. Black cave -70
9. Huts in the mountain +60
10. The Werewall (werewolf-wall) -80, men spilleren får en ekstra tur.
11. The pit -50
12. Goldmine +650

2.3 Use case diagram



Dette er et usecase diagram. Use case diagrammer bliver benyttet, så man lettere kan håndtere, samt overskue den opgave/det projekt som man står overfor. Dette use case diagram viser overordnet hvordan en spiller interagerar med vores system. Først starter vores bruger/aktør spillet. Derefter bliver spilleren præsenteret med muligheden at oprette sig som spiller dette trin skal udføres for at kunne fortsætte. Til sidst får man muligheden for at kaste med terningen og dermed mulighed for at starte selve spillet.

2.4 Usecase - *briefudgave*

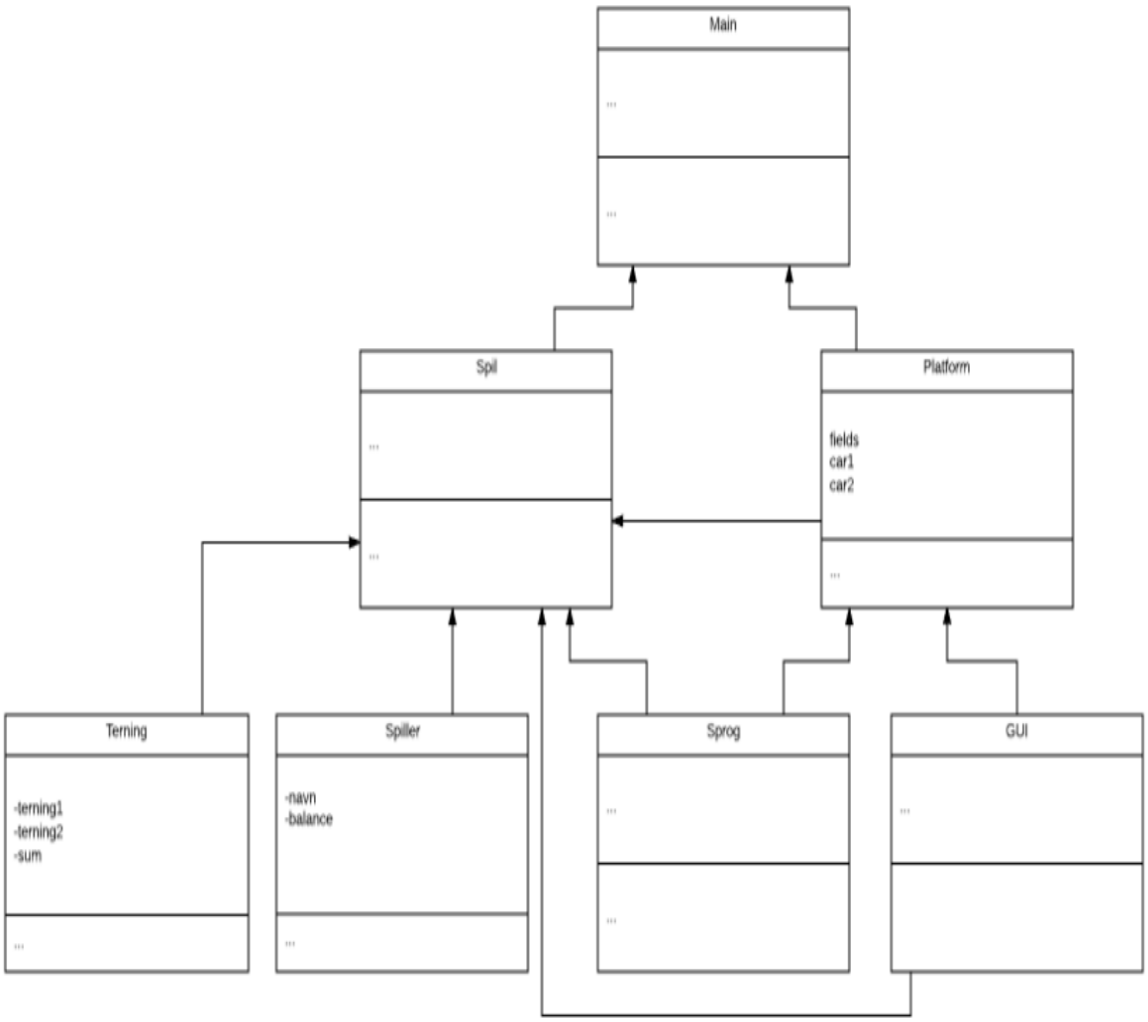
Nedenstående use case viser hvordan spillet helt simpelt forløber. En bruger starter programmet og spiller spillet ved at kaste med terningerne. Dette gentages indtil en har en pengebeholdning på 3.000 og vinder spillet.

Use case:
ID: 1
Brief description:
Primary actors: Spillere x 2
Secondary actors: spillet
Preconditions: • Spillet skal være startet • Der skal være oprettet to nye spillere • Hver spiller starter med en pengebeholdning på 1000
Main flow: 1. *Use case starter når spiller1 kaster de to terninger* 2. *Spiller1's pengebeholdning bliver opdateret alt efter det tilsvarende beløb for summen af øjnene på terningen.* 3. *Spiller2 kaster de to terninger* 4. *Spiller2's pengebeholdning bliver opdateret alt efter det tilsvarende beløb for summen af øjnene på terningen.* 5. *…. 1-6 gentages indtil én af spillerne opnår en pengebeholdning på 3000*
Postconditions: • *Der bliver udpeget en vinder.*

2.5 Use case - fully dressed udgave

Use case:
ID: 1
Brief description: Denne use case, beskriver samme handlingsforløb som den forrige. Dog er denne en udvidet version, hvilket betyder at der er kommet flere scenarier på. Der er fx. En forklaring på hvad der sker, hvis en af spillerne lander på felt 10. Skemaet er med til at vise de enkelte trin som programmet skal opfylde, og vi kan derfor integrere den i vores videre arbejdsproces, for at komme frem til det endelige produkt/spil.
Primary actors: Spillere x 2
Secondary actors: Spillet
Preconditions: • Spillet skal være startet • Der skal være oprettet to nye spillere • Hver spiller starter med en pengebeholdning på 1000 • Efter hver terningkast skifter turen, medmindre andet er angivet.
Main flow 1. *Use case starter når spiller1 kaster de to terninger* *1.1 Hvis summen er 2 lægges 250 til spiller1's pengebeholdning* *1.2 Hvis summen er 3 trækkes 100 fra spiller1's pengebeholdning* *1.3 Hvis summen er 4 lægges 100 til spiller1's pengebeholdning* *1.4 Hvis summen er 5 trækkes 20 fra spiller1's pengebeholdning* *1.5 Hvis summen er 6 lægges 180 til spiller1's pengebeholdning* *1.6 Hvis summen er 7 sker der intet med spiller1's pengebeholdning* *1.7 Hvis summen er 8 trækkes 70 fra spiller1's pengebeholdning* *1.8 Hvis summen er 9 lægges 60 til spiller1's pengebeholdning* *1.9 Hvis summen er 10 trækkes 80 fra spiller1's pengebeholdning* *1.9.1 spiller1 har lov til at slå en ekstra runde* *1.10 Hvis summen er 11 trækkes 50 fra spiller1's pengebeholdning* *1.11 Hvis summen er 12 lægges 650 til spiller1's pengebeholdning* 2. *spiller2 kaster de to terninger* *2.1 Hvis summen er 2 lægges 250 til spiller2's pengebeholdning* *2.2 Hvis summen er 3 trækkes 100 fra spiller2's pengebeholdning* *2.3 Hvis summen er 4 lægges 100 til spiller2's pengebeholdning* *2.4 Hvis summen er 5 trækkes 20 fra spiller2's pengebeholdning* *2.5 Hvis summen er 6 lægges 180 til spiller2's pengebeholdning* *2.6 Hvis summen er 7 sker der intet med spiller2's pengebeholdning* *2.7 Hvis summen er 8 trækkes 70 fra spiller2's pengebeholdning* *2.8 Hvis summen er 9 lægges 60 til spiller2's pengebeholdning* *2.9 Hvis summen er 10 trækkes 80 fra spiller2's pengebeholdning* *1.9.1 spiller2 har lov til at slå en ekstra runde* *2.10 Hvis summen er 11 trækkes 50 fra spiller2's pengebeholdning* *2.11 Hvis summen er 12 lægges 650 til spiller2's pengebeholdning* 3. *1-2 gentages indtil en af spillernes pengebeholdning er 3000 eller større*
Postconditions: • *Der bliver udpeget en vinder.*

2.6 Domain model

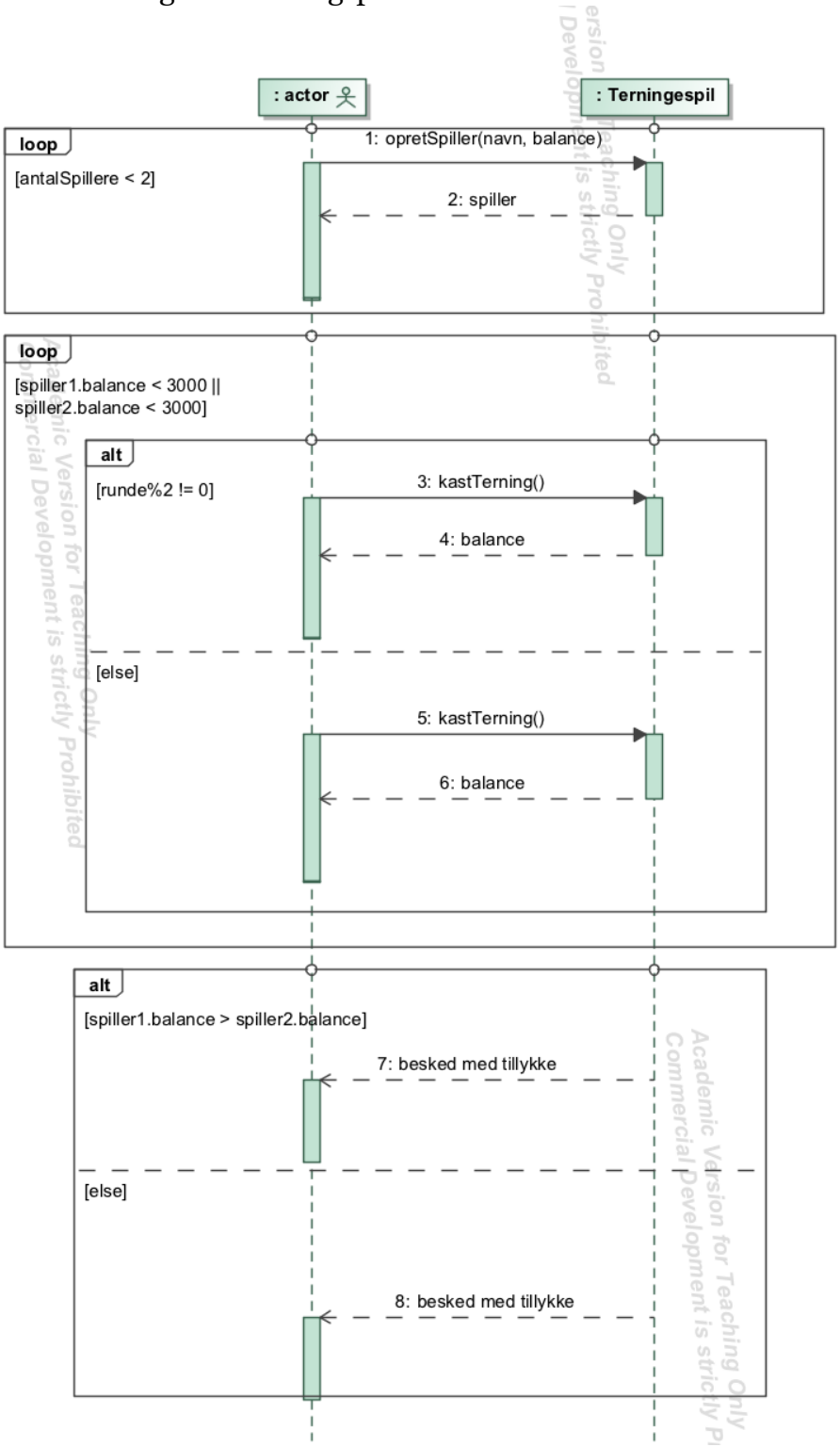


Dette diagram er en domænemodel. En domænemodel er med til at dele de forskellige klasser op i ting man har fra virkeligheden. Man benytter altså en domænemodel, til at give et bedre overblik over hvordan tingene fra virkeligheden kan blive implementeret og laves til et program.

I ovenstående model har vi vores klasser fra vores spil repræsenteret. I dette diagram har man kun attributter med, hvilket forklarer hvorfor der kun er 3 af vores klasser som indeholder dette.

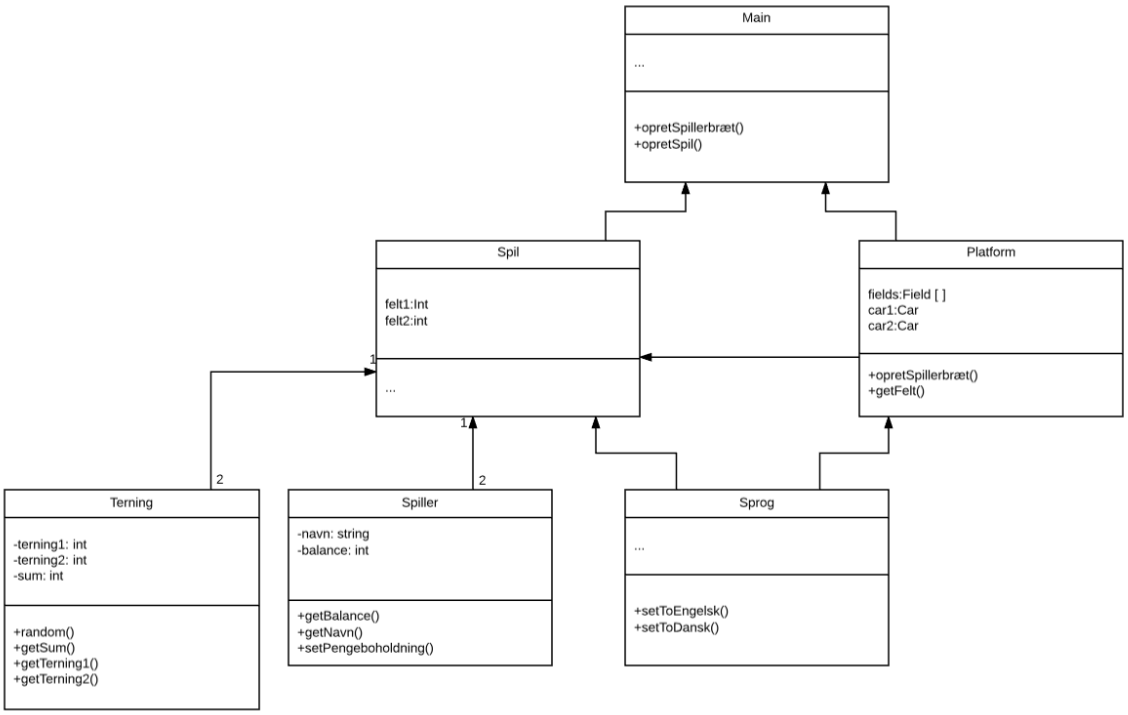
2.7 Systemsekvensdiagram

Nedenstående diagram, er et systemsekvensdiagram, som vi har udviklet til vores spil. Et systemsekvensdiagram er et diagram, som illustrerer hvordan aktør og systemet kommunikere, via system operationer. Systemet bliver betragtet som en black box forstået på den måde, at det viser al kommunikation mellem aktør og system, men at man ikke ser hvad der foregår i ”Terningspil”.



3.0 Design

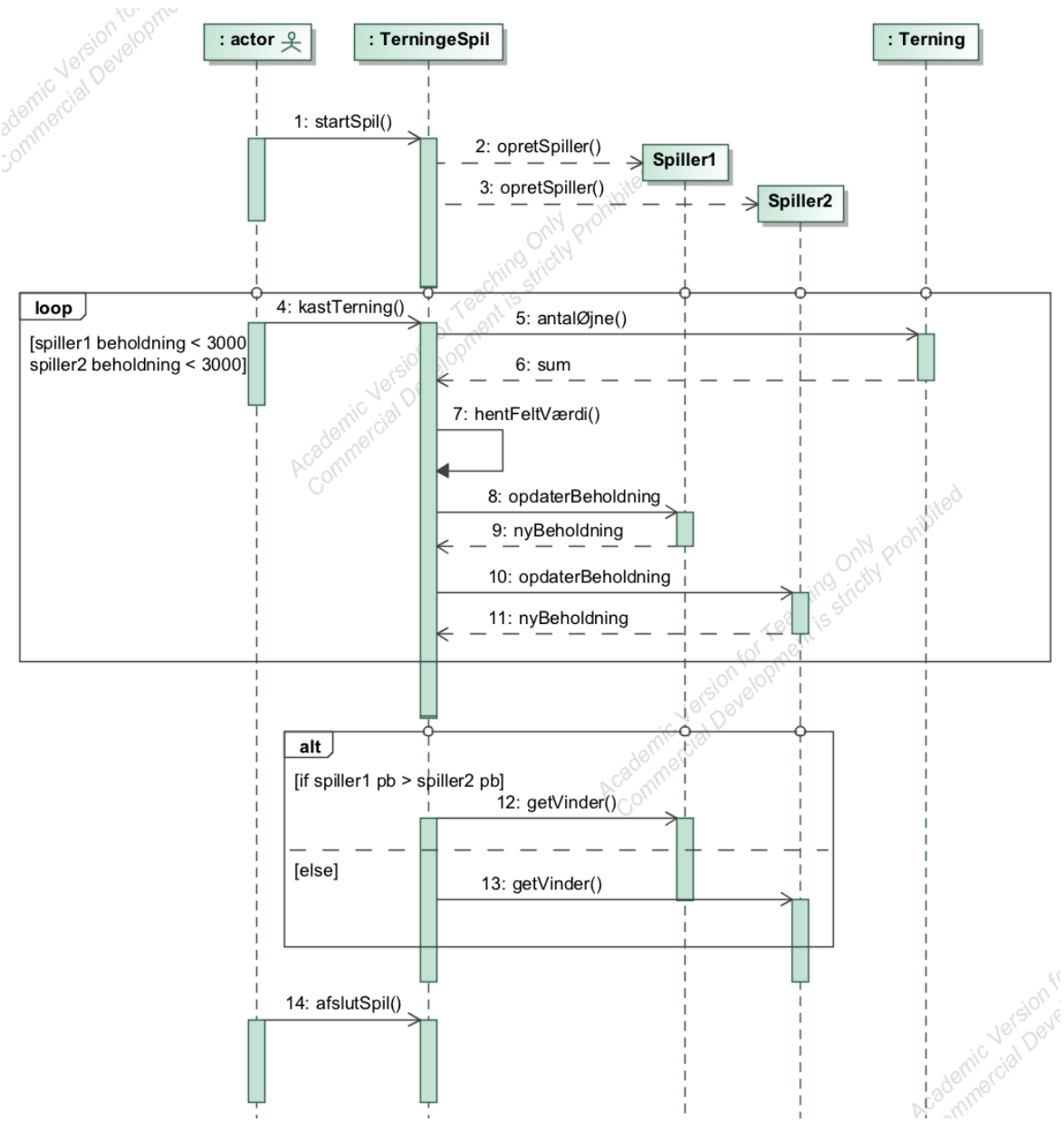
3.1 Design klassediagram



Dette er vores design klassediagram. Design klassediagram minder om domain modellen, forskellen ligger i at design klassediagrammer også indeholder metoder. Man benytter denne form for diagram, for igen at holde overblik over de forskellige klasser til brug i den senere udvikling i java. Metoderne som indgår i diagrammet, er altså også dem der senere skal implementeres i koden.

Vi har i dette design klassediagram tilføjet alle metoderne til de klasser som har nogle. Desuden har vi tilføjet multiplicities på de klasser hvor det gav mening. Som det fremgår i diagrammet er der 2 terninger pr. Spil og der er 2 spillere pr. Spil.

3.2 Sekvens Diagrammer



Vores interaktionsmodel, er det man kalder sekvensdiagram. Sekvensdiagrammet er en viderebygning på domænemodellen.

4.0 Test

Testen af spillet kan findes i filen *TestCageOfDoom.java*
Der testes for følgende metoder i vores spil:
getNavn, setPengebeholdning(), getBalance() og random()
Der oprettes en ny spiller, som bruges til at kalde de første tre metoder.

getNavn() metoden er meget simpel. Her returner metoden den String, som den nye spiller er blevet navngivet.

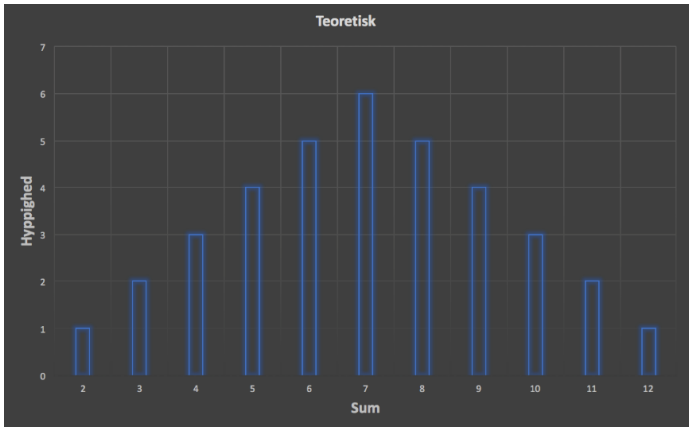
Den første getBalance() test forventes der at programmet returnerer en værdi på 1000. Dette er fordi, at vi i Spiller klassen har sat denne pengebeholdning som startværdi, når der oprettes en ny spiller. Testen opfylder forventningen, hvilket er godt, da det er en af kravene.

Når setPengebeholdning() testes, bruges et input på +2000 point. Der forventes at når vi kalder getBalance(), at den returnerer en værdi på 3000 point. Dvs. de 1000 fra start som ligges oven i de plus 2000. Testen er positiv.

Test nummer 2 af getBalance(), tester om pengebeholdningen kan blive negativ. Der bruges et input på - 20.000 for at sikre at den samlet burde blive mindre end 0. Dog har vi i Spiller klassen sat metoden til, at når den bliver negativ, skal den sættes lig 0. Dette viser vores test også at den gør. Dette betyder altså, at man i spillet ikke kan risikere at få en negativ pengebeholdning, selvom man er uheldig kun at lande på de felter der giver negative point.

Ligesom i vores CDIO1 er der testet for antallet af forekomsten af hver sum. Dette gøres for at tjekke om random funktionen fungerer efter de teoretiske sandsynligheder når terningerne kastes.
Nedenstående skema er resultatet af vores test. Det ses at testen stemmer overens med den teoretiske sandsynlighed, som er grafisk afbildet i diagrammet som blev oprettet i forrige CDIO 1 projekt.

sum	forekomst
2	2801
3	5613
4	8410
5	11197
6	13825
7	16662
8	13868
9	11008
10	8299
11	5551
12	2766



[1]

Diagram 1

[1] Diagram 1 er genbrugt fra CDIO 1

5.0 Konfiguration

For at køre spillet Cage Of Doom på din maskine, anbefaler vi fra IOOuterActive, at du opdaterer dit styresystem til nyeste version. Dette gøres under AppStore på en Mac. På en windows, kan man bruge dette link til Windows Support for, at finde nyeste tilgænglige version af Windows: https://support.microsoft.com/en-us/help/4027667/windows-update-windows-10

Kompatibilitet for Cage Of Doom:

Mac OS X Sierra 10.8 eller højere
Windows 7 eller højere
Den nyeste version af Eclipse

Cage of Doom kan findes på følgende Git repository link:
https://github.com/LordHjorth/CDIO2.git
For at importere spillet til Eclipse udføres følgende trin:

1. Eclipse åbnes.
2. Click file og import…
3. Søg efter Projects from Git, eller scroll down i menuen og find punktet på listen.
4. Herefter vælges “Clone URl”
5. Man vil nu blive bedt om Authentication details til Github.
6. Herefter vil der være mulighed for at vælge Branches, master branch indeholder final release, men det kan vælges at hente de resterende for at følge arbejdsprocessen.
7. I det følgende vælges hvor ens Git repository skal ligge lokalt.
8. Dernæst vælges det at importere projektet som et eksisterende eclipse projekt.
9. Herefter vælges lokation i workspace, og man vil nu have tilgang til projektet.
10. Vinduet Git Repositories bør nu være synligt i Eclipse, hvis ikke kan det findes ved at clicke på Window>Show View>other og søge efter Git Repositories.

Det rekommanderes at man bruger Windows eller Mac OS, men programmet kan principielt køre på alle platforme og alle processor typer (fx 32 bit, 64 bit). Fleksibiliteten på tværs af operativsystemer er et resultat af måden Java koden kompileres på. Kildekode for Cage of Doom kompileres til java bytecode. For at køre kildekoden, er det nødvendig at bruge en translatør, JVM (Java Virtual Machine) og denne skal være installeret på platformen. JVM konvertere java bytecoden til den pågældende platforms maskinkode, hvilket gør platformen i stand til at forstå koden og køre den 1.

6.0 Projektplanlægning

Vi har arbejdet løbende med design- og analysefasen, hvor vi har skitseret modeller og diagrammer.

Vi har i forhold til planlægningen af projektet arbejdet med Unified Process, i det omfang, at vi har været igennem en idéfase. Her har vi i gruppen debatteret omkring muligt indhold til vores program, denne del er kendt som inception-fasen i Unified Process. Her har vi tegnet diagrammer og startet implementationen af koden. Denne del er kendt som elaboration-fasen i Unifed Process. Dernæst har vi lavet den primære implementation, denne del er kendt som construction-fasen i Unified Process og slutteligt har vi testet vores program, denne del er kaldet transition-fasen i Unifed Process.

I forhold til kodedelen, så har vi arbejdet sammen på denne del af projektet i et gruppelokale på CBS. Dette har vi tilføjet, fordi det kan se ud som om, at kun en i gruppen har 'committed' til GitHub, men dette skal ikke tage noget fra de resterende gruppemedlemmer.

7.0 Diskussion

I opgavebeskrivelsen står der, at det skal være nemt at skifte mellem terninger. I spillet Cage Of Doom, er dette dog ikke relevant, da spillepladen består af 12 felter og det skal være muligt at lande på felterne 2-12. Det vil sige at det er en nødvendighed, at spillet indeholder 2 terninger, med værdierne 1-6 eller lignende. Kravet er dermed modstridende i forhold til programmets funktionalitet. Hertil skal det dog tilføjes, at hvis man ønskede at benytte terningerne til et andet spil, ville det ikke være problematisk at ændre dem således, at det fx var 12 siders eller 20 siders terninger, da vi har oprettet en terning klasse. Det skal dog tilføjes at da det er blevet valgt at bruge GUI´en i opgaven, vil det ikke give mening at ændre terningerne, da de visuelle terninger viser værdierne fra 1-6.
Et yderligere krav til opgave var mulighed for nemt at skifte sprog for spillet. Det har vi implementeret som en brugervenlig knap under hele spil seancen, hvor spilleren kan vælge enten danske ellers engelske output messages. Yderligere har vi oprettet sprog som en klasse for sig selv, hvilket gør det nemt for enhver der ønsker at arbejde videre med programmet at implementere flere sprog.

Vi har i spillet Cage of Doom valgt ikke at oprette en konto klasse, da vi har brugt GUI'en, der er blevet stillet til rådighed. GUI'en indeholder en setBalance(String, int) metode, som vi har brugt til at opdatere og holde styr på pointene for hver af spillerne.

7.1 Oversigt over klasser og pakker
Vi har i selve spillet en pakke som hedder Cage Of Doom, hvilket er den pakke som indeholder alle klasser som er relateret til selve spillet. Klasserne er:

Main.java
Platform.java
Spil.java
Spiller.java
Sprog.java
Terning.java
Yderligere har vi oprettet en pakke sprog, der indeholder henholdsvis Dansk og Engelsk tekstfil, med mulighed for at tilføje yderligere sprog, skulle dette ønskes.

Ansvarsområderne er fordelt således:
Spil klassen har ansvaret for terning klassen og spiller klassen. Desuden har både spil klassen og platform klassen ansvaret for sprog-klassen. Grunden til, at ansvaret for sprog klassen er fordelt både til platform- og spil klassen er at funktionen med at skifte sprog har indflydelse på både sproget i platformen, altså selve brættet men også på de beskeder som spillet kommer med, som er samlet i spil klassen. Det kunne være "Tillykke du har vundet", denne skal jo oversættes til engelsk, hvis man spiller på engelsk, derfor har det også betydning for spil klassen.

Til sidst er det main klassen, som har ansvaret for spil klassen og platform klassen.

Vi har forsøgt, at definere nogle klasser, som kun kalder hinandens 'public' metoder. Dette er gjort, så man lettere kan foretage ændringer uden, at det har nogen betydning for den del af koden man ikke ændrer i. Derudover har vi forsøgt, at give klasserne det bedste mulige ansvar. Det er ud fra lav kobling høj samhørighed princippet vi har forsøgt at gøre.

Lav kobling er, at er klasserne er så uafhængige af hinanden som muligt. En klasse skal kun associeres med de få klasser, der er nødvendige for at den kan opfylde sit ansvarsområde. Dette gør det mere overskueligt at forstå klassen. Det er altså hvad vi har sigtet efter, under implementeringen af vores klasser.
Høj samhørighed er, at der er en rød tråd igennem klassen. Vi har forsøgt at give vores klasser et veldefineret ansvar, så det er klart og tydeligt hvad klassen har ansvar for. Metoderne i vores klasser er relateret til hinanden. Dette er med til at gøre vores program lettere at forstå, så man ved at kigge på et klassediagram kan forstå hvad der sker.

Funktionalitet i platform klassen:
Vi har i platform klassen tilføjet noget funktionalitet, som vi mener forbedre interface designet. Vi har tilført hvert af de felter som man kan lande på en individuel farve. Det har vi valgt, for at gøre selve designet på interfacen mere behageligt at se på, og for at give en øget brugervenlighed idet det gør det nemmere at adskille felterne fra hinanden.

Funktioner i spil klassen:
I spil klassen har vi tilført en funktion som gør, at hver gang man slår med terningen, så rykker spillerens bil med over på feltet man flytter over på. Det vil altså sige, at hvis en spillers bil holder på felt 3, og spilleren slår den samlede sum 6, så rykkes bilen til feltet for sum 6, samtidig med, at bilen bliver slettet fra det felt som man før stod på. Vi mener at det ville være et realistisk element at tilføre, ligesom når man spiller et brætspil i virkeligheden, så flytter man sin brik med rundt i eksempelvis ludo eller matador.

8.0 Konklusion

Formålet med opgaven var, at udarbejde et terningspil skrevet i Java kode, på baggrund af analyseværktøjer fra UML. Vi kan konkludere at vi har benyttet unified modelling language som analyseredskab igennem opgaven. Dette kommer til udtryk i form af vores forskellige diagrammer, som stammer fra UML. Disse diagrammer blev benyttet i faserne, som ledte op til programmeringsfasen. Dernæst var det påkrævet at man udarbejdede en Junit test som en af de sidste led i opgavebesvarelsen. Dette trin kan vi dermed konkludere at have fuldført. I vores Junit test, testede vi for 4 forskellige metoder, som var inkluderet i programmet. Alle disse tests testede positive og lever dermed op til kravene.

Samlet set, skulle vores program også leve op til forskellige specifikke krav, givet af en imaginær kunde. Der var både overkrav og underkrav, som programmet skulle leve op til. Vi kan konkludere at vores program opfylder disse krav. Nogle af kravene var blandt andet, at der var specifikke værdier på bestemte felter, hvilket også er repræsenteret i vores program.

9.0 Glossary [2]

UML (Unified Modeling Language):

En universel standard for udseendet af diagrammer hvormed man kan beskrive strukture og forløb i objekt orienteret programmering.

UP(Unified process):

En process til udvikling af objekt orienteret software. Den er iterativ, man arbejder i iterationer, faserne Inception, Elaboration, Construction og Transition. Den er arkitektur centreret, use case og risk driven. Yderligere er den inkrementel, dvs man får i hver iteration en udvidelse af det færdige system.

Domænemodel:

En model der repræsentere koncepter fra den virkelige verden.

Use Case Diagram:
Den simpleste form for diagram over en brugers interaktion med et system. Et sådant diagram kan være med til at identificere de forskellige bruger af systemet.

Use Case:
En sekvens af handlinger (og variationer), som et system kan udføre, der giver et målbart resultat som har værdi for en der interagere med systemet.

Interaction Model/Design:
Viser interaktionen mellem de involverede klasser og interfaces.

GUI (Graphical User Interface):
En grafisk brugerflade, der gøre det lettere for en person at interagere med systemet.

Multiplicity:
Et tal der repræsenterer hvor mange elementer/objekter på et UML diagram der interagere med hinanden. Det er muligt at der ikke er nogen øvre grænse for interaktionen, det annoteres ved *.

[2] Genbrugt fra CDIO 1

10.0 Litteratur og kilde fortegnelse

Bøger:
"*Applying UML and patterns*"
- 3rd Edition published 2004
Author: Craig Larman
ISBN-13:9780131489066

"Java Software Solutions"
8th Edition
Authors: John Lewis, William Loftus

Hjemmesider:
https://en.wikipedia.org/wiki/Glossary_of_Unified_Modeling_Language_terms
Sources: Fowler, Martin. *UML Distilled: A Brief Guide to the Standard Object Modeling Language* (3rd ed.). Addison-Wesley. ISBN 0-321-19368-7.
Tom, Pender (2003). *UML Bible*. John Wiley & Sons. ISBN 0-7645-2604-9.
http://www.oracle.com/technetwork/java/compile-136656.html