Projektopgave efterår 2015 – jan 2016
02312-14 Indledende programmering, 02313 Udviklingsmetoder til IT-Systemer og 02315 Versionsstyring og testmetoder.

Projekt navn: *CDIOdel2*
Gruppe nr.: *38.*
Afleveringsfrist: *lørdag den 07/11 2015 Kl. 05:00*

Denne rapport er afleveret via Campusnet

Denne rapport indeholder 16 sider inkl. denne side.

Silas El-Azm Stryhn
s143599

Ramyar Hassani
s143242

Mikkel Hansen
s143603

Frank Thomsen
s124206

Martin Rødgaard
s143043

Indholdsfortegnelse

Timeregnskab

CDIO del 2 Timeregnskab					
	Silas	Martin	Mikkel	Ramyar	Frank
Analyse	2	2	2	1	2
Design	3	2	1	2	2
Implementering	2	3	7	7	6
Dokumentation	6	5	4	2	1
I alt	13	12	14	12	11

Resumé

Vores rapport indeholder et analyseafsnit, hvor vi har analyseret og opdelt alle kundens krav i vigtige og mindre vigtige dele, samt skrevet om kravene er funktionelle eller om de ikke er funktionelle. Vi har også udarbejdet nogle diagrammer til analysen.
I vores design afsnit har vi lavet og beskrevet diverse diagrammer, som gør vores spil mere overskueligt og forståeligt.
I implementeringsafsnittet har vi beskrevet alle de forskellige klasser. Hvordan de fungere og samarbejder, og hvad de har at sige for selve spillet flow.
Til sidst har vi lavet en test, som viser om man kan få/have en negativ beholdning i spillet, hvilket man ikke kan ifølge testen.

Indledning

Denne rapport præsenterer gennemgangen af den softwaremæssige udvikling af et terningspil for spilfirmaet IOOuterActive. Firmaet ønsker et spil hvor to personer skiftevis kaster med to terninger og hvor der findes forskellige felter man kan lande på. Afhængig af hvilket felt man lander på, vil man enten få point eller blive trukket i point. For at kunne udarbejde sådan et spil på den mest effektive måde og for at undgå eventuelle problemer der kunne opstå under udviklingen, er man nødt til at gennemgå fire stadier, der er nødvendige for at kunne danne sig en struktureret og overskuelig arbejdsplan: Analyse, design, implementering og test.
Rapportens basale formål er at give en grundlæggende forståelse for tankegangen igennem forløbet, hvilke overvejelser der er gjort og hvordan terningspillet er stykket sammen. Rapporten gennemgår således en analyse af kundens vision for spillet og udarbejdelsen af diverse diagrammer, som er med til at overskueliggøre tilgangen til udviklingen.

Hovedafsnit

Analyse

Vi har analyseret og udarbejdet en kravspecifikation ud fra opgavebeskrivelsen. Denne indikerer hvilke krav systemet skal overholde, deres prioritet og om det er en nødvendighed for at programmet kan fungere.
Vi har ligeledes, ud fra kundens vision, udarbejdet en navneords-analyse, hvis formål har været at danne grundlag for eventuelle klasser vi gerne ville oprette i spillet.
Dertil har vi også udarbejdet nogle diagrammer, der løbende er blevet ændret under udviklingen, efterhånden som vi har skrevet mere og mere på koden. Det vil sige, at diagrammerne der fremgår i denne rapport er et produkt af det færdige spil. Vi har ifølge opgavebeskrivelse udarbejdet hhv. Domæne- og BCE-modeller samt UseCase- og System-sekvensdiagrammer.

Kravspecifikation

1	**System**		
1.1	Spillet skal være et spil mellem 2 personer.	Vigtigt	Funktionelt
1.2	Spillet skal kunne spilles på databarerne på DTU.	Vigtigt	Ikke funktionelt
1.3	Spillerne slår på skift med et raflebærer med 2 terninger i.	Vigtigt	Funktionelt
1.4	Spillet skal bestå af 11 felter med hver sin positive eller negativ effekt for spillerens balance (se bilag for de forskellige felters effekter).	Vigtigt	Funktionelt
1.5	Når en spiller lander på et felt skal en unik tekst for det gældende felt udskrives.	Vigtigt	Ikke funktionelt
1.6	Spillerne starter med en balance på 1000.	Mindre Vigtigt	Funktionelt
1.7	Spillet slutter når en spiller har 3000.	Vigtigt	Funktionelt
1.8	Spillet skal kunne være let at oversætte til andre sprog.	Mindre Vigtigt	Ikke Funktionelt
1.9	Det skal være let at skifte til andre terninger.	Mindre Vigtigt	Ikke Funktionelt
1.10	Spilleren og hans pengebeholdning skal kunne bruges i andre spil.	Vigtigt	Ikke Funktionelt

2	**Versionering**		
2.1	Det skal kunne være muligt at arbejde videre på systemet.	Vigtigt	Ikke Funktionelt
2.2	Det skal være muligt for kunden at følge med i udviklingen af projektet og gruppemedlemmernes bidrag til dette.	Vigtigt	Ikke Funktionelt
3	**Testing**		
3.1	Spillet skal være testet grundigt og det skal være muligt at kunne gentage testene.	Vigtigt	Ikke Funktionelt

Navneordsanalyse

Gul = Navneord

Blå = Udsagnsord

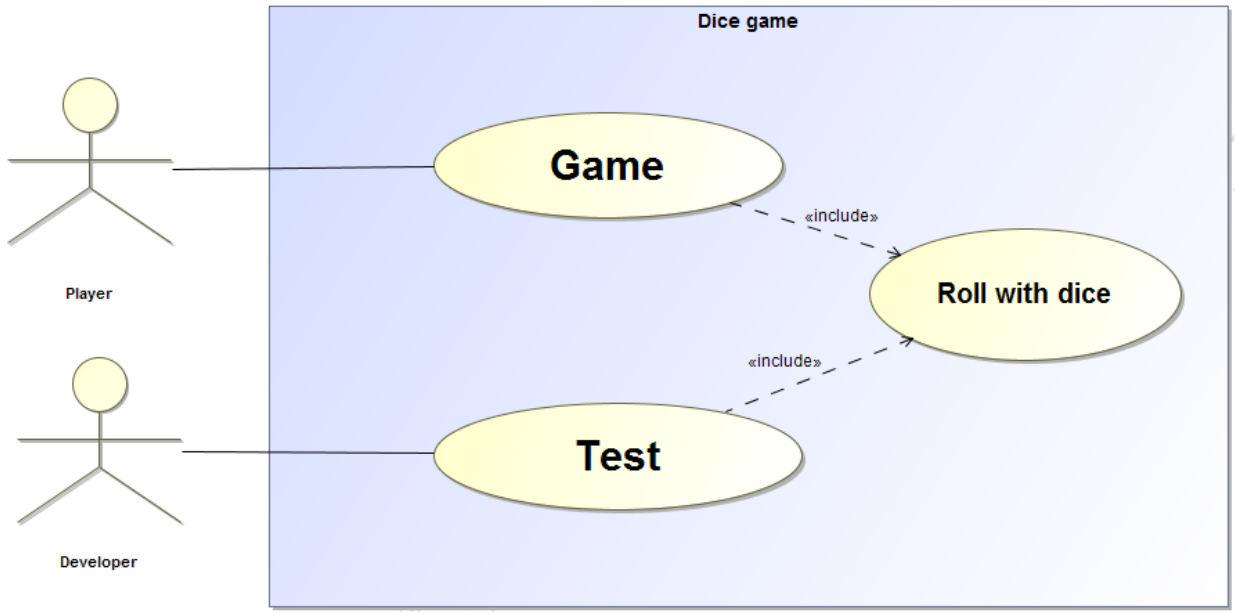
Spillerne slår på skift med 2 terninger og lander på et felt med numrene fra 2 12. At lande på hvert af disse felter har en positiv eller negativ effekt på spillernes pengebeholdning. (Se den følgende feltoversigt), derudover udskrives en tekst omhandlende det aktuelle felt. Når en spiller lander på Goldmine kan der f.eks. udskrives: “Du har fundet guld i bjergene og sælger det for 650, du er rig!”.
Spillerne starter med en beholdning på 1000.
Spillet er slut når en spiller har 3000.
Spillet skal let kunne oversættes til andre sprog.
Det skal være let at skifte til andre terninger.
Vi vil gerne have at I holder jer for øje at vi gerne vil kunne bruge spilleren og hans pengebeholdning i andre spil.

Ud fra det har vi lavet en Player-klasse, en Dice-klasse, en GameManager-klasse (den fandt vi frem til ud fra, at der skulle være en ny effekt alt efter hvad man slog) og en DiceCup-klasse. Udover de klasser, som vi har fået fra navneordsanalysen, har vi også en PlayerAccount-klasse, som varetager de forskellige spilleres point.

Diagrammer

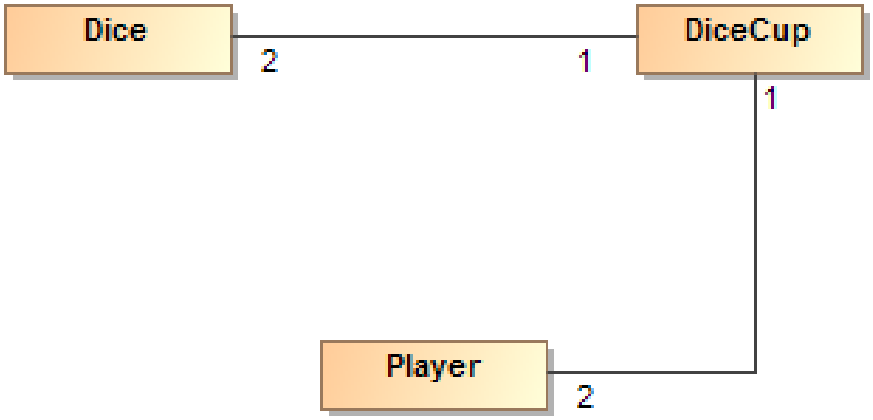
UseCase-diagram

Dette er vores før-programmerings Use-case diagram, som viser hvordan vi havde tænkt os at vores terningespil skulle fungere. Vi har en Player, som spiller spillet, ved at rulle med terningerne og så har vi en Developer, som tester spillet, ved at rulle med terningerne.



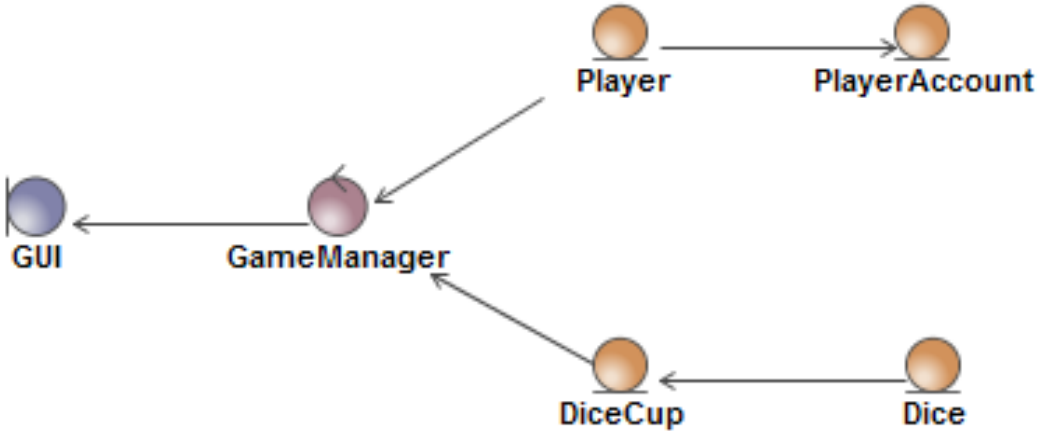
Domænemodel

Vi har i vores domæne model vist hvilke klasser, som vi synes er relevante for vores spil, så det kan fungere. Som det kan ses er der to Player's, som bruger en DiceCup, som der indeholder to Dice's.



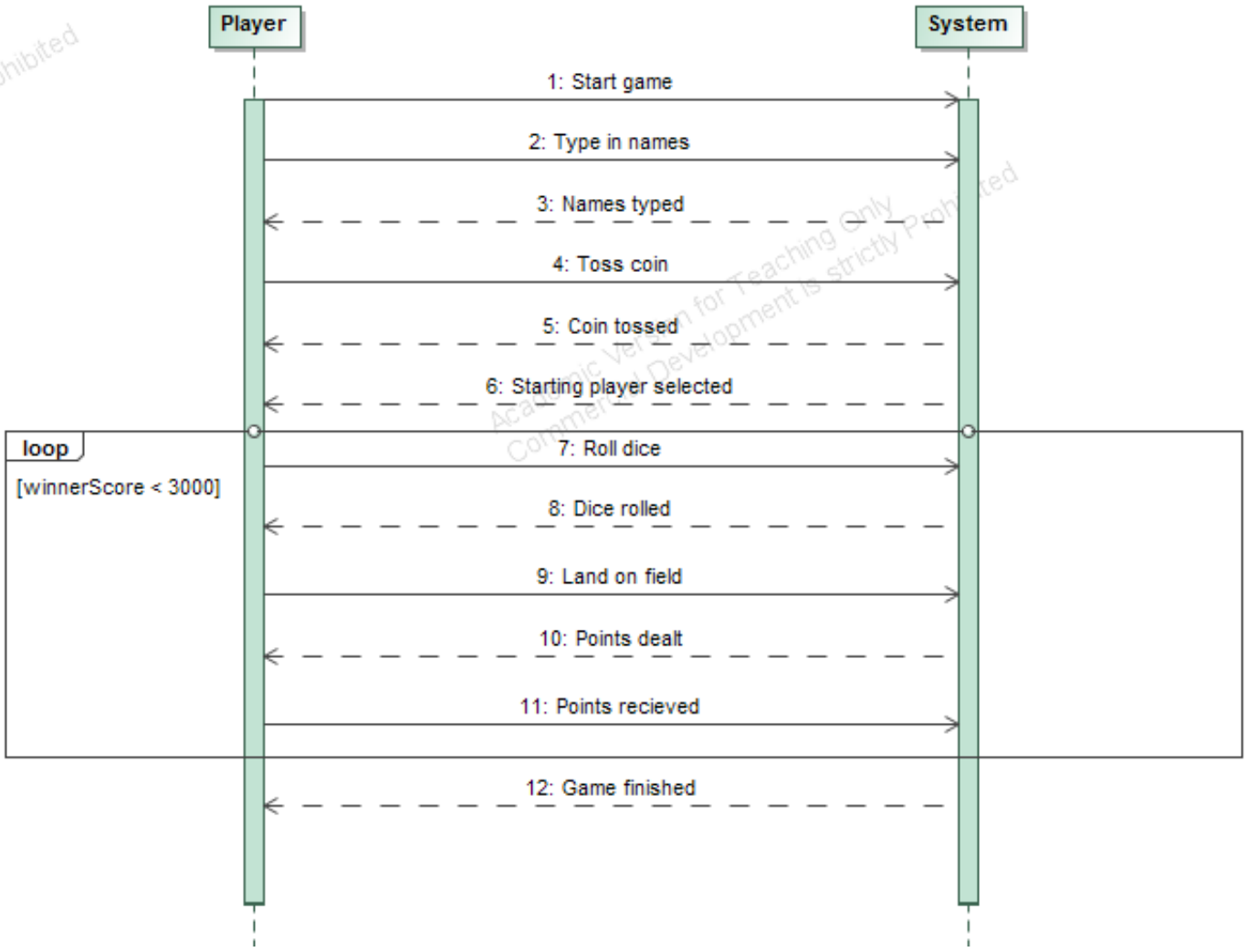
BCE model

Vores BCE model viser hvilke af vores klasser der er controllers, entitys og boundarys. Vores controllers styrer gameflowet, boundarys viser vores GUI, som vores spil så bliver kørt i. Entitys indeholder metoder, som vores controllers så kalder.



System-sekvensdiagram

Dette diagram viser hvordan brugeren og systemet kommunikere igennem selve spillet. Brugeren starter spillet og skriver navnene, så svarer systemet tilbage ved at skrive navnene på skærmen. Så kaster brugeren en mønt for at vælge hvem der starter. Systemet svarer så at mønten er kastet og at en tilfældig spiller er valgt. Så ruller brugeren med terningerne og dertil svarer systemet ved at vise terninger på skærmen. Så lander brugeren på et felt og systemet frigiver så point, som brugeren lægger til sine point. Spillet slutter så når brugeren har opnået 3000 point.



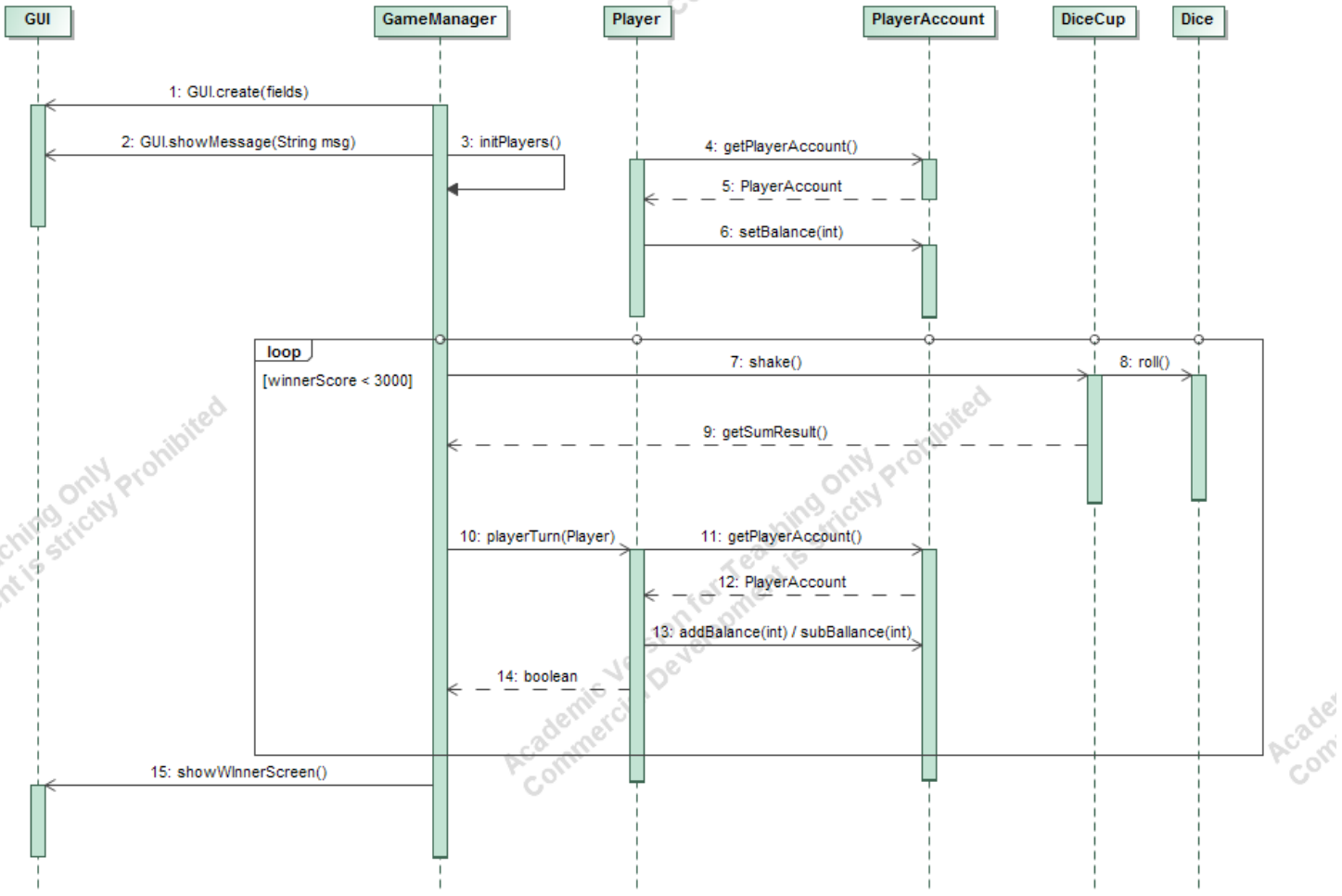
Design

Det kan være en rigtig god idé at udarbejde nogle diagrammer i forbindelse med udviklingen af et system. Diagrammer er med til at overskueliggøre hvordan systemet fungerer i forskellige situationer og det kan være en god måde at visualiserer opbygningen af systemet på overfor kunden. På baggrund af dette har vi udarbejdet nogle diagrammer som giver et overblik over hvordan vores system er sat sammen samt hvilke funktioner der indgår.
I designfasen har vi udarbejdet et design-klassediagram og et design-sekvensdiagram.

Diagrammer

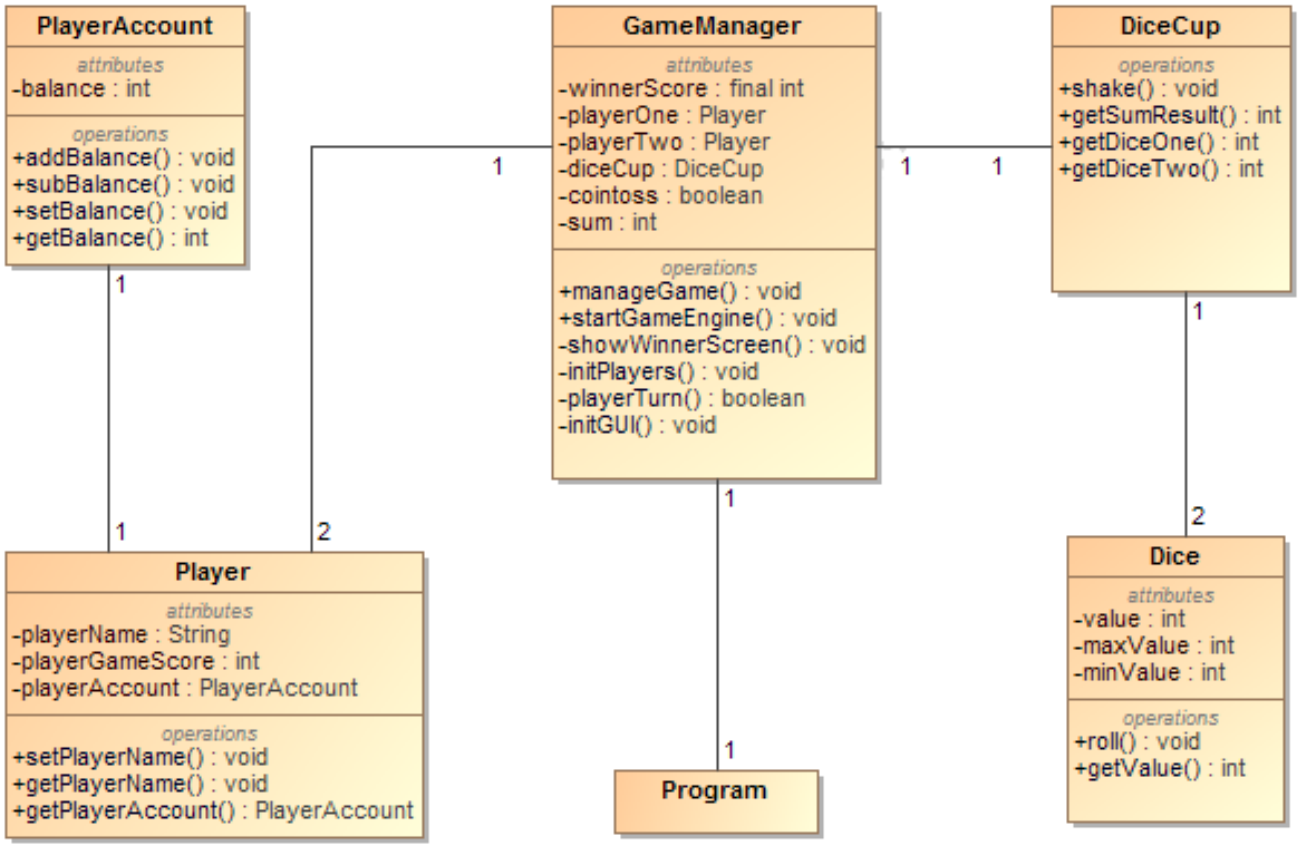
Design-sekvensdiagram

Dette diagram viser flowet i vores spil og i hvilken rækkefølge de forskellige metoder bliver kørt i. Det viser ligeledes hvordan de forskellige klasser arbejder sammen og hvordan det hele bliver kørt tilbage til GUI'en der gør spillet visuelt for brugerne. Dertil viser det også hvilke metoder der kører inde i det loop, der gør at der bliver skiftet tur mellem spillerne.



Design-klassediagram

Dette diagram er en videreudvikling af vores domæne model, som vi har lavet sideløbende med programmet. Diagrammet viser alle vores klasser, med attributter, metoder og multipliciteterne klasserne imellem.



GRASP

GRASP principperne Creator, Controller, Høj binding, Information
Expert og lav kobling har vi overholdt, da vores forskellige konstruktører selv opretter de objekter de skal bruge fra andre klasser, vores controllere uddelegere arbejdet til andre klasser, vi har høj binding i form af, at vi har en del forskellige klasser der sørger for hver deres del af programmet. Der er ikke noget der går på kryds og tværs af klasser. Information expert har vi brugt til at vælge hvilke klasser der skal have hvilket ansvar. Som udgangspunkt skal det være den klasse med mest information om noget der så håndterer opgaven. Lav kobling går hånd i hånd med høj binding og det har vi sørget for ved at lave klasser og metoder der fungerer som byggeklodser, lidt i stil med LEGO, hvor man frit kan bygge det op, som man nu engang har brug for.

Implementering

Se bilag for vejledning til installation og systemkrav.

Dice:

I denne klasse er der tre attributter, som alle er private. Den første attribut er en *int value*, den anden er *int maxValue* og den sidste er *int minValue*.
Herefter har vi lavet to konstruktører, som hedder *Dice* og som begge er public. Forskellen på disse to konstruktørerr er, at den ene er en normal 6-sidet terning, og den anden, kan man selv vælge antallet af sider, da den kan modtage to int værdier i form af *maxValue* og *minValue*.
Så har vi lavet en metode *public void roll()* hvor vi genere et random tal mellem vores *maxValue* og *minValue* og sætter det random genererede tal til vores int attribut *value*.
Den sidste metode i denne klasse er en *public int getValue()* som returnere vores int attribut *value*.

DiceCup:

Denne klasse er vores raflebæger, som indeholder to terninger. Klassen består af to attributter, en konstruktør og 4 metoder. Attributterne er *private* og og hedder *Dice diceOne* og *Dice diceTwo*.
Konstruktøren i klassen hedder *public DiceCup()* og den opretter to nye terninger og sætter dem henholdsvis til *diceOne* og *diceTwo*.
public void shake() er den første metode i klassen og genere et random tal til hver terning, ved at kalde metoden *roll()* fra klassen Dice.

Vores næste metode *public int getSumResult()* udregner summen af *diceOne* og *diceTwo*, sætter det lig med en lokal variabel, som vi har kaldt *int sum* og returnere derefter *sum.*
Metoden *public int getDiceOne()* returnere værdien af *diceOne.* Metoden bliver brugt i klassen GameManager, til at vise *diceOne* i GUI'en.
Det samme gælder den sidste metode *public int getDiceTwo()* som returnere værdien af *diceTwo* og som også bliver brugt i GameManager klassen og denne metode gør det samme som den forrige metode.

Player

I vores Player-klasse har vi to private attributter: *String playerName* og *PlayerAccount playerAccount*. Konstruktøren *public Player()* sørger selv for også at oprette en *playerAccount* med 1000 point. I denne klasse har vi også én setter metode og to getter metoder. *setPlayerName(String playerName)* tager imod et String input og setter herefter spillerens navn til det den får ind. *getPlayerName()* returnerer spillerens navn. *getPlayerAccount()* returnerer playerAccount, så vi kan få adgang til de metoder der ligge i PlayerAccount-klassen.

PlayerAccount

I denne klasse er der kun en attribut, en private int balance. Konstruktøren i PlayerAccount-klassen tager en int som argument. Hvis den int er mindre eller lig med 0, sætter den balance i PlayerAccount til 0. Ellers sætter den balance til det argument den får når den bliver oprettet. I PlayerAccount-klassen har vi fire metoder. En *addBalance(int amount)* som ligger amount til de point spilleren allerede har, en *subBalance(int amount)* der trækker amount fra de point spilleren allerede har. Hvis det bliver mindre end 0, sætter den blot balance til 0, *setBalance(int amount)* som sætter spillerens balance til amount, hvis den er mindre end 0, sætter den balance til 0 og til sidst har vi *getBalance()* der returnerer balancen på spillerens konto.

GameManager:

Dette er klassen hvor vi samler alt fra de andre klasser. Denne klasser har 6 attributter som alle er private: *final int winnerScore = 3000, Player playerOne, Player playerTwo, DiceCup diceCup, boolean cointoss og int sum.*
Metoden *private void initGUI()* sætter vores GUI op, ved at oprette 40 tomme felter, i stedet for selve Matador spillebrættet.

Vores metode *private void initPlayers()* opretter spilleplanen til spillet og derefter opretter den to nye spillere, sætter deres navn ved *setPlayerName()* metoden og derefter adder dem til spillet en ad gangen ved *GUI.addPlayer()* som modtager en String i form af navnet på spilleren, en int værdi, som sætter start kapitalen på kontoen og giver hver spiller en bil.
Herefter laver vi en ny terning med 2 sider, som vi slår med, for at vælge hvem der starter. Vi har sat *playerOne* til at starte ved en værdi på 1 og *playerTwo* til at starte ved en værdi på 2.
Vi har også metoden *private boolean playerTurn(Player player)*, som afgører hvilken spiller som er valgt. Vi har lavet metoden som en do/while, da vi først gerne vil have at en spiller, alt efter tur, slår med raflebægeret ved *diceCup.shake()* og får en værdi af summen af de to terninger. Inden i do har vi lavet en switch/case, fra 2 til 12, som angiver hvilken tekst og balance den valgte spiller nu får vist på GUI'en, alt efter hvilken case, som værdien af summen stemmer overens med. Nu kommer while ind i billedet, for hvis man slår 10 uanset hvilken addition, som er foregået, får man en ekstra tur ifølge case 10.
Metoden *public void startGameEngine()* bruger metoderne *initGUI()* og *initPlayers()*. I metoden er der en lokal attribut *boolean gameIsNotWon = true.* Nu har vi lavet en while løkke, som har argumentet *gameIsNotWon.* Inden i while løkken har vi en if/else. Hvis *cointoss = true* så går vi ind i vores if og laver to nye if sætninger. Disse to if sætninger har begge argumentet *gameIsNotWon.* Den første if er at hvis spillet ikke er vundet, så starter *playerTwo* og så skifter turen til *playerOne* hvis *playerTwo* ikke vandt spillet i det kast han lige har haft. Det samme foregår i else sætningen, som vi ryger ind i hvis *cointoss = false,* dog hvor *playerOne* starter.
Nu har vi metoden *public void manageGame()* som har en attribut *boolean wantRematch = true.* Så har vi en while løkke med argumentet *wantRematch.* Inde i løkken kalder vi først metoden *startGameEngine()* og herefter, når spillet er slut, så har vi lavet en funktion, som spørger om der vil være rematch. Hvis inputtet her er "Yes", så vil *wantRematch = true* og while løkken vil køre en gang til. Hvis inputtet derimod var "No", så ville *wantRematch = false* og spillet vil herefter slutte.
Vi har også en konstruktør *public GameManager()* som opretter en ny *DiceCup().*

Program:
Dette er vores main metode, som kører vores spil. I denne klasse, laver vi en ny *GameManager().manageGame()* som starter spillet op og kalder alle relevante metoder fra de resterende klasser.

Test

Vi har lavet en JUnit test case hvor vi starter med at oprette en Player. Inde i vores preconditions *(@Before)* initialiserer vi så denne Player og sætter dens balance til 0. Så har vi en *assertEquals(0, p.getPlayerAccount().getBalance());* der tester, at balancen er korrekt sat til 0.
Vores test består herefter af to tests. I den første prøver vi at sætte balancen til -1000 og i den anden test prøver vi at trække 500 fra den balance vi satte i preconditions.
Til sidst har vi så vores postconditions *(After)*. Her har vi endnu en *assertEquals(0, p.getPlayerAccount().getBalance());* der igen tester om balancen er 0. Såfremt balancen er 0, vil JUnit testen sige, at den har kørt 2/2 tests og, at der er 0 errors og 0 failures.

Konklusion

Vi har fået lavet vores terningspil, som spilles af to spillere, der bruger et raflebæger til at slå med terningerne. Vi har fået implementeret at summen af terninger har en positiv eller negativ effekt på spillerens konto. I rapporten har vi lavet diagrammer for at få en forståelse af, hvordan vi ville lave vores program og hvordan vi ville formidle vores idéer til programmet videre til andre. I opgaven har vi brugt forskellige metoder, fordelt i vores klasser, som i sidste ende alle bliver kaldt i vores main klasse (Program). Opgavens indhold er blevet lavet ud fra kravspecifikationerne, og kan nu bruges som det skal. Alle regler er blevet implementeret. Spillet er til sidst blevet testet med en jUnit test, som undersøger om en spillers konto kan have en negativ int værdi.

Bilag

Installation og afvikling

For at downloade vores online repository, bliver man nødt til først at oprette en bruger på hjemmesiden www.GitHub.com. Herefter har man flere valgmuligheder. Man kan importere det direkte til Eclipse ved at importere dette link https://github.com/Ramii1/CDIO2.git eller man kan klikke på linket og downloade det som ZIP fil. Hvis man importerer det til Eclipse skal det gøres via Files → Import → Git → Projects from Git → Clone URI → udfyld felterne (husk at logge ind i bunden). Når det er gjort henter Eclipse selv hele projektet ned. Det er meget vigtigt, at man henter Master branchen her, da vi ikke har skrevet på HEAD.
Når først man har hentet projektet ned til Eclipse, skal man blot køre klassen "Program" for at starte spillet.
Ønsker man at have spillet liggende på computerens skrivebord, så det er muligt at køre spillet uden brug af Eclipse, er det muligt at eksportere spillet fra Eclipse. Dette gøres ved at højreklikke på projektet i Eclipse → Eksport → vælge mappen Java → Runnable JAR file → trykke next → Under "Launch configuration" vælges spillets Main klasse "Program – 38_Del2" → dernæst vælges den ønskede destination → Finish.

Systemkrav

Vores spil kan køres på alle maskiner der som minimum er installeret Windows XP på. Det eneste systemkrav der er, er at man skal have installeret JAVA 1.8, for at kunne bruge det library som vi har anvendt (jre 1.8.0_60). Man kan dog godt bruge en nyere udgave, så skal man blot ind og ændre i, hvilke libraries der er tilknyttet projektet. Det er ikke noget større problem, men hvis man ikke føler sig sikker på hvordan man tilføjer / fjerner libraries vil vi ikke anbefale det.