Projektnavn: CDIO 2
Gruppenummer: *18*
Afleveringsfrist: *Fredag den 04/11 2016 Kl. 23:59*
Fag og retning: Diplom Softwareteknologi
Denne rapport er afleveret via Campusnet (der skrives ikke under).
Denne rapport indeholder 37 sider inklusiv denne.

KASPER LEISZNER S165218

BIJAN NEGARI S144261

FREDERIK VON SCHOLTEN S145005

HELENE ZGAYA S104745

TROELS LUND S161791

Timeregnskab:

Timer i alt pr. gruppemedlem							
	Deltager	Design	Impl.	Test	Dok.	Andet	Ialt
	Kasper	2	2,5	0,5	8	0,5	13,5
	Troels	4	4	0,5	3,5	1,5	13,5
	Bijan		1			11	12
	Helene	2			5	4	11
	Frederik	5	3,5	0,5	4,5	4,5	18
	SUM	13	11	1,5	21	21,5	68

Resumé

Vores projekt går ud på at udvikle et spil for to spillere der på skift skal slå et slag med to terninger, lander på det tilhørende felt som enten har en positiv eller negativ effekt på spillernes pengebeholdning. Derudover skal spillet let kunne oversættes til andre sprog, det skal være let at kunne skifte til andre terninger og man skal kunne bruge spilleren og hans pengebeholdning i andre spil. Vi får skrevet et program der opfylder disse krav, og projektet er vellykket.

Indholdsfortegnelse

Indledning

Denne rapport er udarbejdet som led i undervisningen i fagene *Indledende programmering (02312), Udviklingsmetoder til IT-systemer (02313) og Versionsstyring* og *testmetoder (02315)* på første semester på diplom software.
Rapporten dokumenterer planlægning og udvikling af et program der skal simulere et meget enkelt brætspil mellem 2 spillere. Vi har skrevet i det objektorienteret programmeringssprog Java.

Først og fremmest har vi udfærdiget en kravspecifikation som er udført mellem os udviklere, projektleder og kunde, og som har til formål at sikre et færdigt produkt som begge parter er enige om opfylder kundens ønsker. Denne beskrives mere udførligt i hovedafsnittet.
Derudover har vi benyttet flere modeller til at visualisere analyse samt dokumentation for udvikling af vores program.

Det primære mål er altså udvikle nedenstående spil ved objektorienteret analyse og design
Sekundært ønsker vi at implementere en GUI der er blevet udviklet på forhånd.

Brætspillet foregår altså mellem to spiller der skiftes til at slå med to terninger. Værdien af disse placerer spilleren på et af 11 felter og modtager eller taber penge afhængigt af hvilket felt spilleren lander på. Første spiller der når 3000 kroner har vundet. Spilleren opnår ekstra tur såfremt spilleren slår to ens (?)
Opsummering:

Følgende er altså beskrevet i denne rapports hovedafsnit:

- Krav og analyse (dertilhørende modeller)
- Design-dokumentation (dertilhørende modeller)
- Beskrivelse af koden
- Test samt dertilhørende beskrivelser
- GRASP

Sidst vil der forelægge en diskussion samt en konklusion som grunder i hvorvidt vi har formået at fuldføre alt beskrevet ovenfor.

Indledende overvejelser omkring proces
Inden start på projektet gjorde vi os nogle overvejelser om hvordan det ville være mest hensigtsmæssigt at strukturere projektet. Vi valgte af følge UP iterationer og sætte deadlines efter hver iteration. Planen er vedlagt som bilag. Planen skal ikke opfattes som endelig eller uforanderlig plan. Tværtimod har der undervejs været flere afvigelser fra planen i hver enkelt fase hvor vi har været nødsaget at gå tilbage og foretage ændringer i eksempelvis vores UML diagrammer.
Ikke desto mindre har benyttelsen af UP givet os et bedre overblik og gjort os i stand til at planlægge arbejdet effektivt.

Analyse

Til udvikling af vores program har vi først og fremmest udfærdiget en kravspecifikation ud fra kundens vision. Dernæst har vi udarbejdet flere modeller med udgangspunkt i UML og ved hjælp af objektorienteret analyse.
Følgende modeller er udviklet til visualisering af vores overvejelser for hvordan programmet skal se ud overordnet, for derefter at kunne udvikle modeller der beskriver udviklingen mere i detaljer:

- Kravspecificering
- Use-case diagram
- Use-case beskrivelser
- Domæne-model
- BCE
- Klasse diagram analyse[1]

Kravspecifikation Analyse:

Vi har udarbejdet en kravspecifikation ud fra kundens vision, hvilket har til formål at danne nogle specifikke og klare retningslinjer. Det gøres for at skabe en fuldstændig forståelse mellem udvikler og kunde, så det færdige produkt opfylder kundens ønske og så man undgår misforståelser og i øvrigt får klargjort overfor kunden hvad der er muligt, da kunden oftest ikke selv er udvikler. På denne måde agerer kravspecifikationen også som kontrakt mellem udvikler og kunde.
En anden metode til udvikling af kravspecifikation kunne være FURPS (functionality, Usability, Reliability, performance, supportability), men bruges hovedsageligt til større projekter end vores.
De funktionelle krav beskrives nedenfor, mens den fulde kravspecifikation findes som bilag til denne rapport:

Funktionelle krav

- Det skal være et spil mellem 2 spillere.
- Spillerne slår på skift med 2 terninger.
    - Hver terning har 6 sider som er lige sandsynlige.
    - Spilleren lander på det felt (nr.) som er summen af de 2 terninger.
- Spillet skal have felter nummeret 2-12 (se feltliste).
- Hvert felt har en henholdsvis positiv eller negativ effekt på spillerens pengesum.
- Spillerne starter med 1000 kroner hver.
- Spillet slutter når en spiller har opnået 3000 kroner.
- Det skal være let at skifte til andre terninger (terninger med andre værdier).

[1] Findes i afsnittet for dokumentation, da den hører sammen med design-klasse-diagrammet.

Navneords analyse

Navneords analysen har taget udgangspunkt i opgavens givet kravspecifikation. Det vil sige at der systematisk er blevet gået igennem hvert ord, har det været et navneord eller udsagnsord, er det blevet indsat i to tabeller som set på figur a og b.

Navneord

Terningespil	Spil	Spiller	Databar	Udvikler
Person	Forsinkelse	Felt	Nummer	Positiv
Negativ	Effekt	Pengebeholdning	Tekst	Goldmine
Sprog	GUI	Nyt	Tidligere	Maskine
Feltliste	Metode			

Figur a tabel over navneord

Udsagnsord

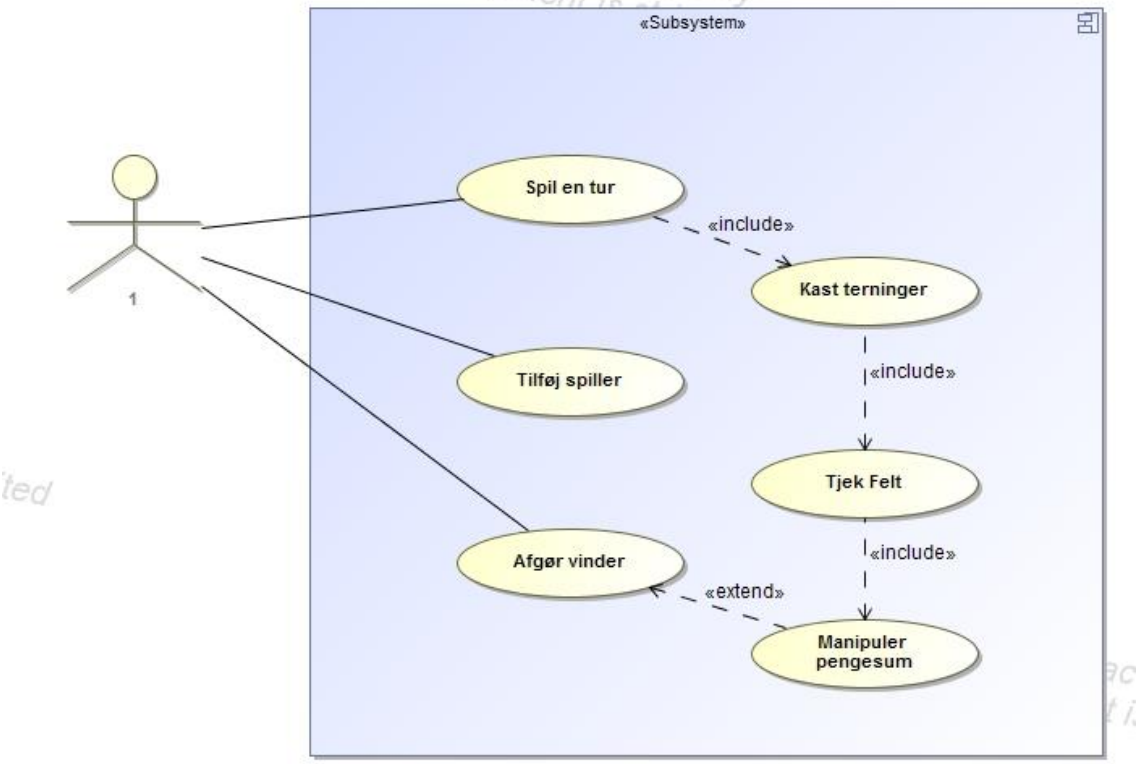
Imponere	Udvikle	Spille	Slå	Lande
Oversætte	Skifte	Bruge	Starte	Slutte

Figur b tabel over udsagnsord

Disse to analyser bliver brugt senere i designprocessen til at fremstille en domænemodel og et klassediagram.

Use-case diagram

Bruges til at visualisere forskellige scenarier samt aktører samt analysere anvendelsesområderne. Man kigger grundlæggende på hvordan brugere og systemet interagerer med hinanden. Vi har kun udfærdiget ét use-case diagram da vores system ikke er kompliceret nok til at vi synes det var nødvendigt at udforske brugsmønstrene yderligere.



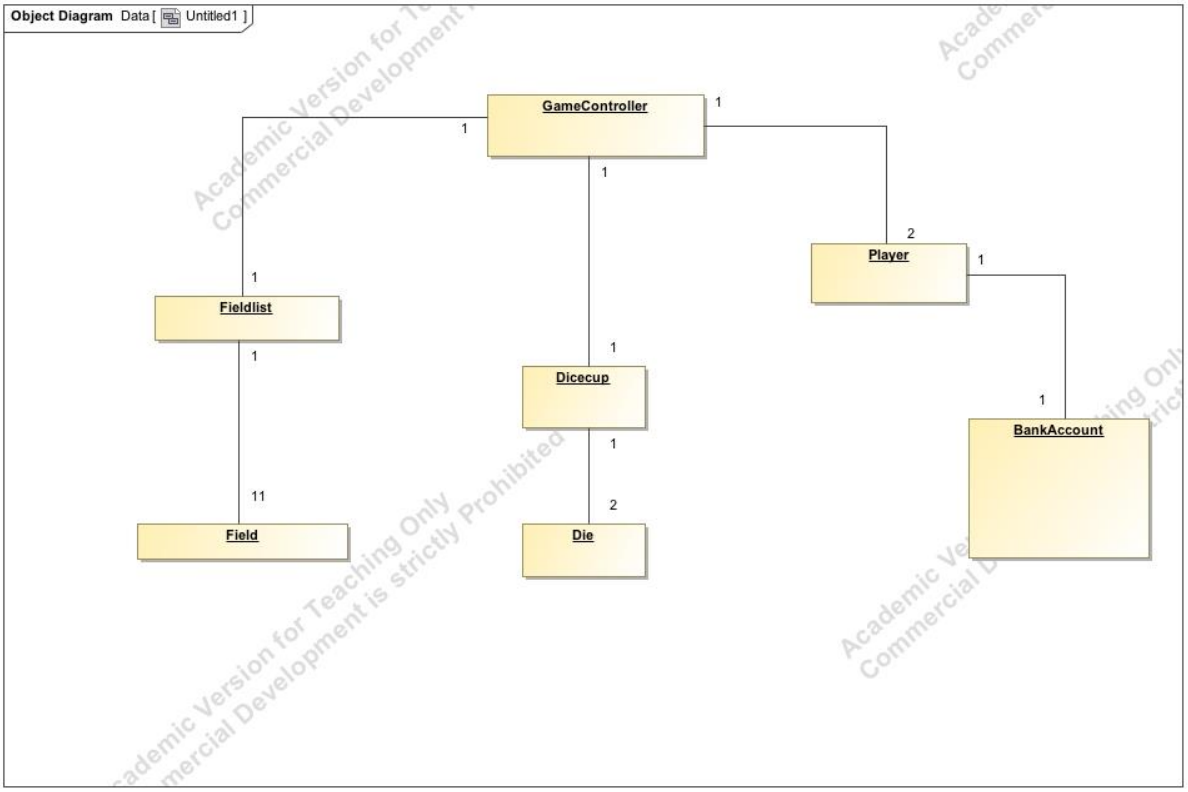
Den primære og eneste aktør er spilleren af vores spil. Dette use case diagram beskriver en tur for en spiller. Det at spille en tur inkluderer at kaste med terningerne samt at holde den værdi som terningerne giver tilbage op mod feltlisten for derved at afgøre manipulationen af pengesummen. Afgør vinder use casen kommer i spil efter endt tur hvor den afgører hvorvidt en spiller som har turen kan siges at have vundet, dvs har en sum større end, eller lig med 3000.

“Spil en tur” er den højest liggende use case da de andre underliggende use cases indeholdes af denne.

Use-case beskrivelser

Beskriver scenarierne med hovedforløb, eventuelle før betingelser og efter betingelser, samt eventuelle alternative forløb. Se bilag.

Domænemodel

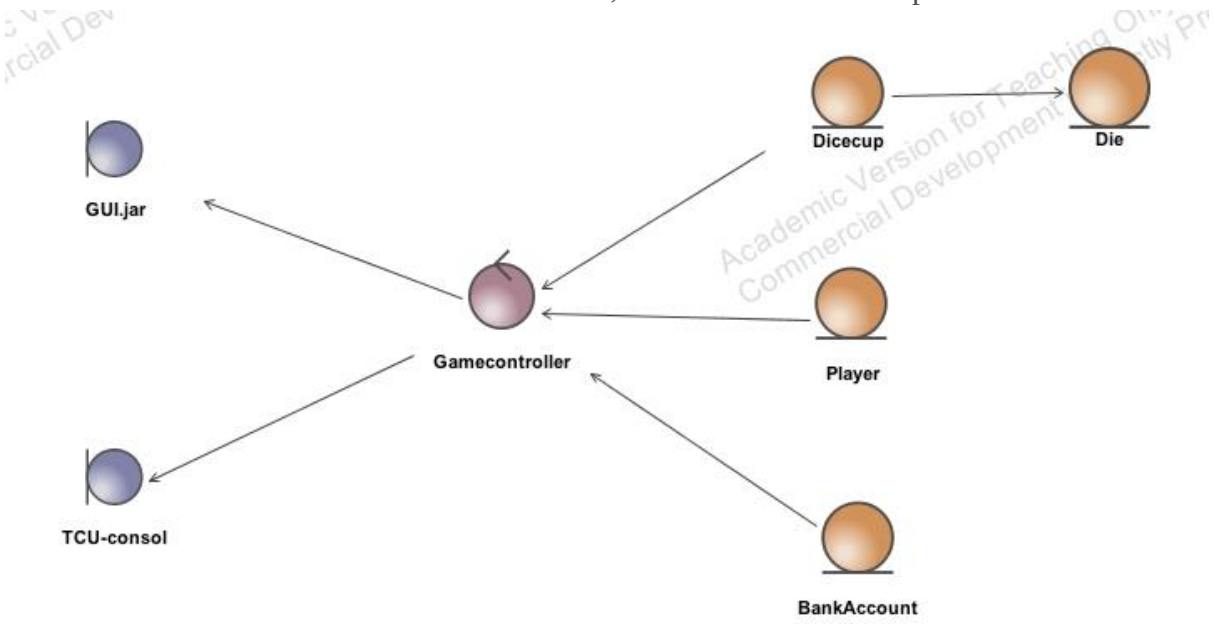


På vores domænemodel ses at vi har vores GameController som er forbundet til Fieldlist, Dicecup og Player. Ved Player klassen har vi multiplicitet 2, da gamecontrolleren indeholder 2 spillere. Hver spiller har da 1 BankAccount. Det ses på modellen da Player er forbundet til BankAccount med multiplicitet 1.

GameController klassen indeholder også et raflebæger eller Dicecup som vi kalder det i vores program. I Dicecup er der 2 terninger (die). Det ses på modellen, GameController forbundet til Dicecup med multiplicitet 1, og Dicecup til Die med multiplicitet 2 for de 2 terning. Herudover indeholder vores GameController også en Fieldlist, som er vores spilleplade. Den indeholder 11 felter. GameController forbundet med Fieldlilst med multiplicitet 1 for 1 spilleplade. Og Fieldlist til Field med multiplicitet 11, for de 11 forskellige felter.

BCE model

BCE-modellen fortæller noget om sammenspillet mellem klasserne, og hvor dataen kommer fra. Har man en klasse der ikke indeholder data, vil den ikke blive vist på BCE-modellen.



Det ses at GameControlleren får data som er indeholdt i klasserne Dicecup, Player og BankAccount. Og derfra bliver data implementeret til boundary klasserne, som er dem der interagerer med aktøerne/brugerne. På vores model ses det at dataen kommer til udtrykker i TCU-consolen og GUI'en som brugeren ser på sin skærm, når han bruger vores program.Boundery elementet ligger på ydergrænsen af systemet og styre og styre kommunikation. Controleren styre flowet af interactionen og entity indeholder dataen.

Design-dokumentation

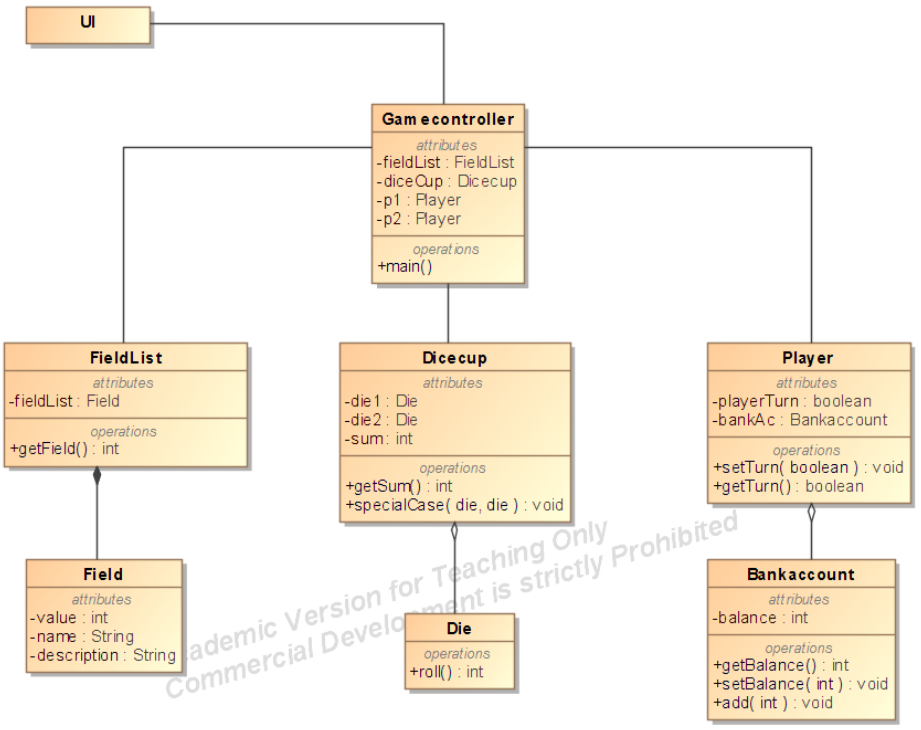
Følgende afsnit omhandler designet af vores program hvor vi har udarbejdet diagrammer med UML notation ud fra objektorienteret analyse og design metoderne. Vi har her udarbejdet følgende diagrammer:

- Design-klasse-diagram
- System Sekvens Diagram
- DSD'er (Design Sekvens Diagrammer)
- Implementering
- Test
- Dokumentation for overholdt GRASP

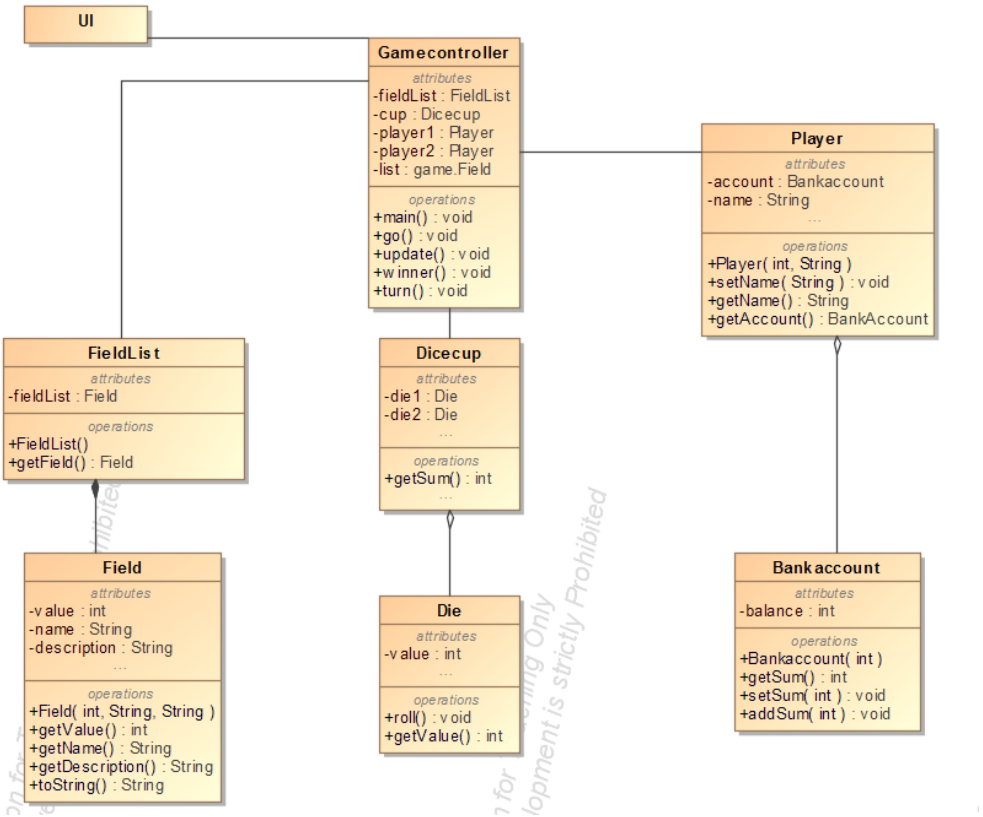
Klasse diagrams analyse

De to klassediagrammer figur 1 og figur 2, er blevet fremstillet for at danne overblik og en slags startpunkt, for projektet. Klassediagrammerne indeholder en generel beskrivelse af de forskellige klasser og deres attributter og metoder. I et klassediagram skal man være særligt opmærksom på plusserne og minusserne foran hver operator og attribut.
Plus betyder adgangstypen er public, hvor minus betyder at adgangstypen er private. Hver binding mellem klasserne har hver sin betydning. En linje uden nogen pil betyder at klasserne ikke nødvendigvis hænger sammen. Hvor en pil uden fylde betyder at de tilnærmelsesvis hænger sammen, og til sidst den pil med fylde betyder at de to klasser hænger sammen.

Undervejs i designprocessen har vi produceret et klassediagram (figur 1). Dette klassediagram bliver brugt som en model til når programmet skal udvikles. Det er altså en model som bruges til at danne et overblik over hvordan det endelige stykke software skal se ud til slut. Det vil sige at gennem projektets udvikling kan det forekomme at analyse diagrammet vil ændre og tilpasse sig. Hver klasse har altså taget udgangspunkt i analyse klassediagrammet (figur 1). Efter programmet blev færdigskrevet, opdateres analyse diagrammet (figur 1) og laves til et helt nyt diagram (figur 2). Det nye diagrams formål er at beskrive det færdige produkts klasser, i form af dens indeholdende attributter og metoder.



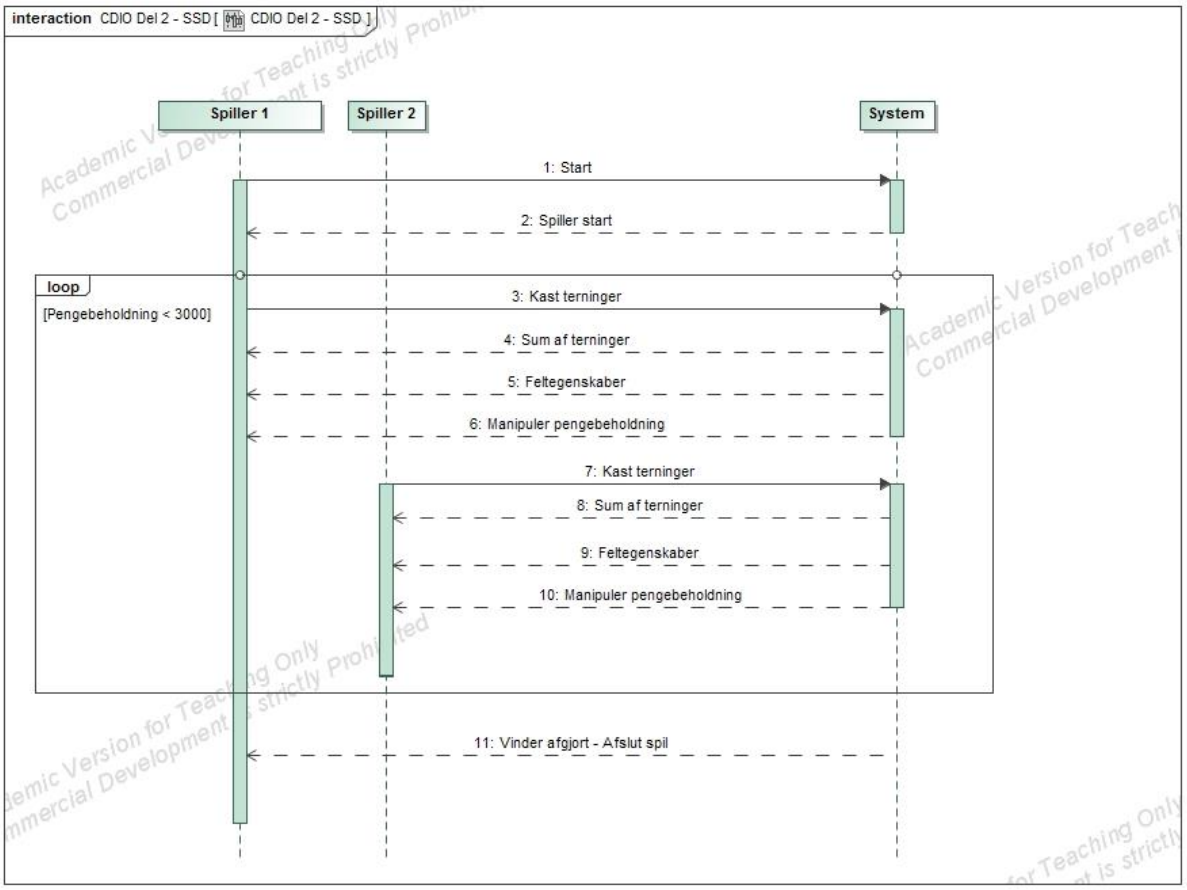
Figur 1


UI
Gamecontroller
attributes
-fieldList : FieldList
-cup : Dicecup
-player1 : Player
-player2 : Player
-list : game.Field
operations
+main() : void
+go() : void
+update() : void
+winner() : void
+turn() : void
Player
attributes
-account : Bankaccount
-name : String
operations
+Player(int, String)
+setName(String) : void
+getName() : String
+getAccount() : BankAccount
FieldList
attributes
-fieldList : Field
operations
+FieldList()
+getField() : Field
Dicecup
attributes
-die1 : Die
-die2 : Die
operations
+getSum() : int
Field
attributes
-value : int
-name : String
-description : String
operations
+Field(int, String, String)
+getValue() : int
+getName() : String
+getDescription() : String
+toString() : String
Die
attributes
-value : int
operations
+roll() : void
+getValue() : int
Bankaccount
attributes
-balance : int
operations
+Bankaccount(int)
+getSum() : int
+setSum(int) : void
+addSum(int) : void

Figur 2

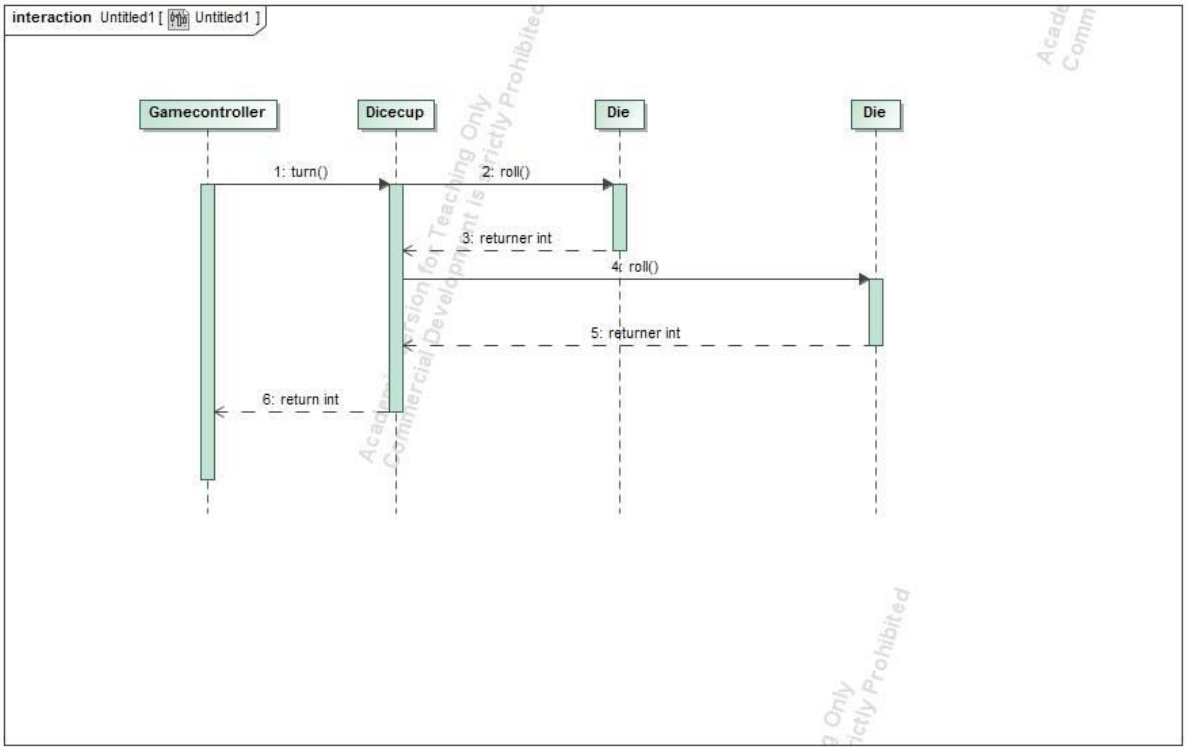
System sekvens diagram:

Systemsekvensdigrammet viser sekvens / main flow for programmets usecases, altså hvordan og hvornår usecases relateret opgaver udføres. Da systemet er forholdsvis simpelt med få use cases har vi valgt at illustrere alle usecases i et SSD i stedet for et diagram for hver enkelt usecase.



I vores tilfælde har vi to aktører, nemlig spiller 1 og spiller 2. Først gennemfører spiller 1 sin tur og derefter gennemfører spiller 2 sin tur. Dette gentages i et loop indtil en spiller opnår 3000 eller derover.

Design sekvens diagram:



Dette design sekvens diagram viser interaktionen mellem klasserne i use casen ”spil en tur” og deres dertilhørende metoder.

Implementering

Implementering af Gamecontroller klassen

Main metoden initialisere et nyt Gamecontroller objekt og kalder *setup()* metoden hvor ved programmet er igang.

Denne Gamecontroller er et helt centrale del af koden, Gamecontrolleren står for logiken bag ved spillet.

Gamecontroller består grundlæggende af attributterne og en nogle metoder som skal bruges til at styre spillet gang.

I Gamecontrolleren bliver der initialisere samt reserveret plads til attributterne, Player 1 og 2. Disse peger som udgangspunkt på null. og får senere en reference i *setup()*. De bruger begge visibility modifieren private for at disse er indkapslet korrekt, i henhold til gode kode skik.

Setup metoden søre for at alt er sat korrekt op ingen spillet starter. Den giver *cup* en reference til en til et *Dicecup* objekt, angiver referencer til både player 1 og 2 samt kalder metoden *addPlayer* i den udleveret GUI så de også grafisk er i framen disse medtager også et car objekt som på samme på samme vis er blevet initialiseret tidligere.

Endvidere defineres grafisk felter, og bliver drawn til canvas med *GUI.create(fields);*.
Til allersidst kaldes næste trin i kørslen af metoden, nemlig en anden metode kaldet *update();*

Update() indeholder et forever loop, som søgere for at skifte tur på spillerne. Begge spillers tur indeholde et kald til GUI'en med en meddelelse om det er den spillers tur , samt et kald til *turn()* med player parameter med den spiller som får turen, som afgøre denne turs videre forløb. Dernæst kaldes *Winner(), som tjekker hvorvidt der kan findes en vinder efter en foregående tur. Dernæst køres disse kald igen for den anden spiller. Den process fortsætte indtil en Winner findes.*

```
public void update(){
    System.out.println("update køre");
    while(true){
        GUI.showMessage(player2.getName() + "`s turn.");
        turn(player2);
        winner(player2);


        GUI.showMessage(player1.getName() + "`s turn.");
        turn(player1);
        winner(player1);
    }

}
```

Turn() indeholder et switch statement som modtager summen af terningerne fra *cup* objektet.
Hver mulig værdi har der efter en tilhørende case som svarende til et felt på vores fiktive spillebræt, some eksekvere hvad dette specifikke felt gør.

Winner() tjekker om den spiller som havde den foregående tur har over 3000 og dermed har vundet. Og giver mulighed for at spille et nyt spil.

Field

Field klassen er den klasse der gør det muligt for en spiller at lande på et bestemt felt. Den er altså en del af spillets logik og ikke det grafiske. Field klassen gør det muligt at oprette et såkaldt field. Et field er beskrevet ved et navn, en beskrivelse og en værdi. Navnet og beskrivelsen anvendes mest af alt til at give spilleren noget feedback. Så når programmet tegner felter på skærmen, så hentes dataen den skal tegne fra Field klassen. Field klassen indeholder mere præcist en konstruktør, tre get metoder, samt en toString metode. Konstruktøren sørger for at de nødvendige informationer bliver defineret hver gang der oprettes et objekt af Field klassen. De tre "get" metoder gør det muligt at hente data for et bestemt fields navn, beskrivelse og værdi. Hvor toString metoden sender alle data om et field på en gang.

Player og BankAccount

Det er et krav at vores kode gør det muligt at både spilleren og hans bankbeholdning skal kunne bruges i andre spil, og at det er let at skifte til andre terninger. Derfor indeholder BankAccount klassen ingen referencer til andre klasser og Player klassen indeholder kun reference til BankAccount klassen. Således kan spilleren og hans pengebeholdning bruges i andre spil da det kun er de to klasser der er afhængig af hinanden.

Metoder i BankAccount
Vi vil beskrive følgende metoder i klassen *BankAccount - getSum*, *setSum* og *addSum*. *getSum*-metoden returnerer den enkelte spillers pengebeholdning. Metoden bruges i forlængelse med en spillers konto, så metoden netop returnerer den spillers pengebeholdning. *setSum*-metoden går ind i den enkelte spillers pengebeholdning og overskriver den, med en ny mængde. *addSum*-metoden bruges når en spiller lander på et felt. Som navnet siger tillægges en værdi til en spillers pengebeholdning. Værdien afhænger af hvilket felt spilleren lander på, og kan godt være negativ. Samtidig sørger *addSum* for at hvis der bliver trukket mere fra en spiller end der er på kontoen, vil pengebeholdning ikke gå i minus, men bare være 0.

Metoder i Player
I klassen Player har vi tre metoder. getAccount returnerer spillerobjektets account af typen BankAccount. En anden metode er getName-metoden som returnerer spilleropbjektets navn. Og derudover er der setName-metoden som ændre spilleropbjektets navn til noget andet.

Terningen

Die klassen indeholder metoden for terningen *roll*. Die indeholder ingen referencer, så den er fuldstændig uafhængig, for at kunne udfylde kravet om at terningen skal være let at skifte og kunne genbruges i andre spil.

Vi benytter os af java funktionen *random.math()* til at generere en terning - i dette tilfælde en terning der returnere et tilfældigt heltal mellem et og seks. Dette kan nemt skiftes til andre værdier.

Test

Under udvikling af dette spil har vi ønsket at benytte os af agile development tilgang med iteratirationer som har udgangspunkt i UP modellen, deraf har vi haft fokus på TDD, som et led i risiko formindske process**.**

Dette er i praksis gjort ved at teste hver enkelt klasse, og dens dens tilhørende metoder, via en tilhørende JUnit test, alle JUnit testene er samlet i et test suite, disse findes adskilt i sin egen pakke med navnet *tests.*

Denne tilgang har vi valgt for, hurtigt og nemt at kunne identificere problemer i systemets forskellige enheder, samt overskueliggøre hvor de fejl der måtte komme undervejs opstår.

Testene kunne være foretaget ved at skrive en normal java klasse men JUnit har en fordel at attributterne bliver redigeret individuelt i selve testen, man skal derfor ikke tage højde for hvorvidt forud for testen er blevet ændret i attributterne. En anden fordel er at alle test er samlet i et JUnit test suite, og derved giver et hurtigt overblik om der er opstået nogle fejl efter modificering af koden.

Dette gør testingen nemmere og lægger op til en større mængde test under udviklingen, med et minimalt tidsforbrug, hvilket i sidste ende vil resultere i færre kritiske problemer undervejs og især mod slutningen af projektet, hvor større design fejl vil kunne være katastrofal, med denne metode er disse fejl forhåbentlig er blevet opdaget i den tidligt i udviklingen.

Test af terningens ægthed samt mulige værdier
Vi tester at terningen kun kan returnere værdier mellem 1 og 6, det det er en normal seks sidet terning. Derfor bruger vi et *if* statement der returnere "false" hvis terningen returnere værdier uden for intervallet. Testen returnerer "true" hvilket betyder at terningen kun kan returnere 1 til 6. Vi vil teste fordelingen af terningens værdier er fordelt ligeligt. Altså undersøger vi ægtheden af terningen. Vi bruger en JUnit test til at teste for fordelingen af de seks mulige værdier - terningens seks sider. Vi kaster terningen 10000 gange i et *for* loop for at få et fornuftigt billed af den statistiske fordeling af terningens slag. Der skulle gerne være lige stor sandsynlighed for at lande på terningens sider 1 til 6. Vi benytter os af en *switch* til at sortere og tælle terningens slag for hver værdi (fordelingen udskrives i konsollen). Vi kunne godt havde brugt et *Array* til at sortere vores kast og gjort vores test kode en del kortere. Derefter bruges *assertEquals* funktionen til at undersøge om terningens tilfældige. Vi forventer at lande på hver side ca. 1666 gange med en fejlmargin på mindre en 1% ca. 166 gange. Hvis dette er tilfældet returnere testen "true".

Test af players konstruktør og BankAccount
I *Player* bliver konstruktøren testet at feltet *String name* og *int sum* bliver sat korrekt via get-metoder. Når man opretter en ny spiller skal man også angive en balancen og navn, som parametre. Vi starter med at oprette et nyt Player objekt. Vi tjekker at konstruktøren har oprettet spilleren med korrekt navn og balance. Det testes vha. *assserEquals* funktionen som returnere "true" for hhv. navn og balance hvis konstruktøren har sat det korrekt.

Vi tester desuden *BankAccount set-* og *get*-metoderne samt *addSum.* Først testes at den balance vi har angivet bliver sat korrekt og at den nye balance vi har sat med *setSum* er sat korrekt. Dernæst tester vi at *addSum* bliver korrekt lagt til balancen. Dette gør vi med *assertEquals* funktionen. Vi undersøger også om balancen kan være negativ og om den kan sættes med *setSum* til negativ balance. Dette tester vi med *assertEquals* funktion der gerne skulle vise at balancen ikke kan blive negativ.

```
public class JUnitTestPlayer {

    @Test
    public void testPLayer() {

        Player p = new Player(50 ,"Knud");

        assertEquals("Knud", p.getName());
        assertEquals(50, p.getAccount().getSum());


        p.getAccount().setSum(0);
        assertEquals(0, p.getAccount().getSum());

        p.getAccount().setSum(100);
        p.getAccount().addSum(200);
        assertEquals(300, p.getAccount().getSum());

        assertEquals(p.getAccount(), p.getAccount());

        p.getAccount().addSum(-500);
        assertEquals(0, p.getAccount().getSum());




        Player p2 = new Player(-21 ,"Brian");

        assertEquals(0, p2.getAccount().getSum()); // summen må ikke være negativ

        p2.getAccount().setSum(-250000);

        assertEquals(0, p2.getAccount().getSum()); // summen må ikke være negativ

    }

}
```

Automatisk test

Derudover kunne man have automatisk test for at teste hele systemet igennem med præfabrikeret testdata, evt læst fra ekstern fil, så resultatet var kendt. Dette er en god måde at vide at det overordnet system returnere de rigtige data tilbage når de skal arbejde sammen. Dette er hurtigt og nem løsning, da testen kan genbruges, så man ikke ved hver iteration i skal udviklinge en ny test. Det kunne fx. Være en fordel ved testning af et specifikt felt på spillepladen, sådan at man ikke skal køre spillet og kaste terninger tilfældigt indtil man

landede på det specifikke felt, hver gang man foretager en test. Der var dog ikke tid til at implementere automatisk test i dette projekt.

GRASP

Information Expert
Vores program er opdelt i forskellige klasser. Hver klasse har noget ansvar, derudover indeholder hver klasse kun de nødvendige metoder.

Creator
Et godt eksempel på brugen af creator, er at vores player sørger for at oprette et objekt af klassen BankAccount.
Controller
I programmet har vi en klasse Gamcontroller. Gamecontrolleren sørger for at kontrollere og styrer alle klasserne i programmet.

Low Coupling
Vi har tre meget adskilte klasser som kan redigeres hver for sig, uden at der vil opstå problemer. Klasserne Dicecup, Fieldlist og Player, har ingen direkte copling til hinanden.

High Cohesion
Die har et meget lille ansvar og den sørger for at kunne give en værdi mellem 1 og 6. Derfor er det muligt for os at oprette to terninger i klassen Dicecup som kan ligge værdierne sammen.

Polymorphism
Dette har vi ikke noget af.

Protected Variations
Programmet er skrevet således at andre kun kan tilgå de nødvendige data. Ved brugen af eksempelvis modifierne public og private.

Konfiguration

Bruger- og systemkrav
Velfungerende testet krav:
Spillere: 2
Hardware: Mus
USB-ports: 2
OS: Windows 7-10/Vista/XP
Processor: Intel Core i7-3630QM
Memory: 16 GB
Graphics: NVIDIA Quadro K2000M
Space Available: 12 KB

Nødvendig software
For at kunne køre programmets kildekode er det krævet at flere stykker software er installeret på den anvendte computer. Det er vigtigt at "Java" og den tilhørende "JDK" er hentet fra nettet og blevet installeret på computeren. Derudover for at kunne køre kildekoden, skal endnu et stykke software hentes og installeres. Softwaren hedder "Eclipse", og er et slags værksted hvor kildekoden produceres og testes.

Kildekoden
For at køre kildekoden på en hurtigt og alternativ måde, er det muligt at åbne programmet "Eclipse". Her er det muligt at identificere en værktøjs bar i toppen af programmet, hvorefter der findes en grøn afspilningsknap. Denne knap køre programmet, hvis man har identificeret klassen med main metoden i. For vores program eller spil, findes main metoden i klassen "GameController", trykkes der herefter på kør knappen, starter programmet. Skal programmet køres uden for "Eclipse" navigeres der til exporter funktionen, under fanen filer. Her er det muligt at exporter programmet til en jar-fil som kan køres på alle maskiner hvor "Java" er installeret.

Git
For at importere kildekoden i "Eclipse", fra et såkaldt git repository, navigeres der til programmets import funktion som findes i fanen file. Nu skulle der gerne dukke et nyt vindue op, her udfoldes mappen "git". Tryk derefter på knappen git projekt. Et nyt vindue vil dukke op, i URL feltet indskrives en kopieret URL af det eksisterende repository.
CDIO del2 gruppe 18 git URL: https://github.com/trolund/CDIO18_del2.git

Konklusion

Vi har formået at udvikle et program der simulerer et meget enkelt brætspil mellem to spillere. Produktet overholder alle kundens funktionelle krav. Programmets funktionalitet er dokumenteret og testet med JUnit tests. Der lægges vægt på at vores kode gør det muligt at både spilleren og hans bankbeholdning skal kunne bruges i andre spil, og at det er let at skifte til andre terninger.

Vi har tilgået softwareudviklingen ved hjælp af *Agile software development*. Der er taget udgangspunkt i *Unified Process* med tidsbegrænset iterationer fordelt i tre faser, forberedelsen (inception) fasen er undladt, i henholdsvis etablering (elaboration), konstruktion (construction) og afslutningsvis i overdragelse (transition).

Versionsstyringen er foregået i Github. Tekniske udfordringer i kommunikationen mellem eclipse og Github og manglende konsensus (rammer, regler) for Git flow i gruppen har dog betydet at versionsstyringen er blevet noget rodet undervejs.

Alt i alt er projektet overordnet set lykkedes.

Bilag 1:

Kravspecifikation

Introduktion

Formål
Formålet med kravspecifikationen er at beskrive og dokumentere "softwarens" specifikke krav.

Kravspecifikationen har til formål at skabe enighed og forståelse for hvad man ønsker softwaren skal kunne og hvad der kan lade sig gøre. Den udfærdiges mellem udviklerne af softwaren og kunden.

Specifikke krav

Funktionelle krav

- Det skal være et spil mellem 2 spillere.
- Spillerne slår på skift med 2 terninger.
    - Hver terning har 6 sider som er lige sandsynlige.
    - Spilleren lander på det felt (nr.) som er summen af de 2 terninger.
- Spillet skal have felter nummeret 2-12 (se feltliste).
- Hvert felt har en henholdsvis positiv eller negativ effekt på spillerens pengesum.
- Spillerne starter med 1000 kroner hver.
- Spillet slutter når en spiller har opnået 3000 kroner.
- Det skal være let at skifte til andre terninger (terninger med andre værdier).

None- funktionelle krav

- Spillet skal kunne køres på DTU's maskiner i databarene.
- Koden skal kunne bruges (genbruges) til andre spil
- Spillet skal kunne køre uden forsinkelser på over 1 sek

Krav til koden[2]

- Lav en klasse Spiller der indeholder en spillers attributter og funktioner.
- Lav tilsvarende en klasse Konto der beskriver Spillerens pengebeholdning.
- Overvej typer attributter Spiller, samt Konto skal have. Lav passende konstruktører.
- Lav passende get og set metoder.
- Lav passende toString metoder.
- Tilføj metoder til at indsætte og hæve penge på en Konto.

[2] Taget fra opgavebeskrivelsen

- Ændr nu Konto-klassen således at der ikke kan komme en negativ balance, ligeledes skal metoderne fortælle om transaktionen er blevet gennemført (***Hint***: brug statementet **return** til at returnere denne information).
- Lav det spil kunden har bedt om med de klasser I nu har.
- Hvis I vælger at bruge GUI'en kan I evt. benytte metoderne i Bilag 1.
- Husk at skrive en oversigt over pakkerne og deres klasser - klassernes ansvarsområder og evt. spændende funktioner.

Noter til kravspecifikationen

**Undladt i kravspecifikationen - argumenter hvorfor?*
Under introduktion er følgende undladt: definitioner, referencer og oversigt

Spørgsmål til Ronni

Hvor mange sider skal hver terning have? Vi antager at den maksimale sum af de 2 terninger er 12. og dermed er en normal 6 sidet terning.

Forbliver man på det felt, som man har slået forrige gang, og rykker derfra ved næste slag? (f.eks du slår 3 første gang og rykker til felt nr. 3. Næste tur slår du 7. og rykker til nr. 10)

I skriver "Vi vil gerne have at I holder jer for øje at vi gerne vil kunne bruge spilleren og hans pengebeholdning i andre spil." hvordan skal det forstås? skal spillerens "id" pengebeholdningen gemmes og huskes så spilleren kan bruge / overføre i et andet spil. Eller er det blot koden der skal kunne genbruges?

Svar fra Ronni:

Hver terning har 6 sider som er lige sandsynlige.

Ved starten af hver tur nulstilles spillerens position. Hvis første slag giver 3 rykkes spilleren til felt 3. I næste tur slår samme spiller 7 og lander nu på felt 7. Ikke noget med at rykke videre.

Kunden er måske også i gang med at lave et ludo spil og vil derfor gerne kopiere "jeres" kode. Det betyder at de nævnte klasser ikke må indeholde "matador"-specifik logik.

Tilføj spiller er usecasen hvor spillerens navn indtastes og spiller objektet oprettes med tilhørende konto. Jeg er ikke rigtig i stand til at følge jeres logik - Bare fordi en af jeres usecases har preconditions, betyder det vel ikke at i kan ignorere andre usecases. Er i med?

Bilag 2

Use-case diagrammer

Use case: Spil en tur
ID: 01
Kort beskrivelse af use casen: En tur spilles. Spiller kaster med terningerne terningens værdi svarer til et felt på vores "spillebræt". Feltet afgører hvorvidt der bliver lagt penge til- eller fra spillers sum. Hvis spilleren ikke opnår 3000 kroner går spiller tilbage til udgangspunktet. Næste spiller kaster terningen. Fortsætter som ovenstående. Når en af spillerne har en pengesum på 3000 har den pågældende vundet. Spillet har endvidere en maksimal forsinkelse på et sekund.
Primary actors: Spiller
Secondary actors: Ingen
Precondition: 1. Spilles på DTU's databarer 2. Det er spiller x's tur
Main Flow: 1. Spiller klikker på "Kast terning" knappen. 2. Terningen giver en værdi der udskrives på UI. 3. Værdien bliver tjekket op mod feltliste. 4. Point lægges til eller trækkes fra på baggrund af ovenstående felts værdi. 5. Besked bliver skrevet på UI.
Postcondition: 1. En tur er blevet spillet og turen overgår til den anden spiller.
Alternative flow: None

Use case: Tilføj spiller
ID: 02
Brief description: Der skal oprettes to spillere før spillets start. Dernæst kan spillet begynde som beskrevet i usecase ID 01
Primary actors: Spiller x
Secondary actors: Ingen
Precondition: 1. Minimum to deltagere i spillet 2. Hver spiller har 1000 kroner fra start
Main flow: 1. Programmet starter 2. Spillere bliver tilføjet.
Postcondition: 1. Systemet er klar til at begynde spillet.
Alternative flow: None

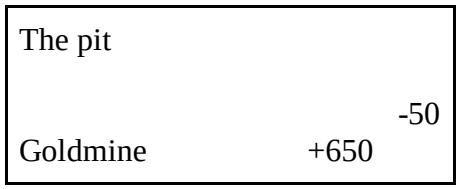
Use case: Kast terninger
ID: 03
Brief description: Terningen bliver kastet for at afgøre en værdi.
Primary actors: Den spiller som har turen.
Secondary actors: Ingen
Precondition: Spillet er igang mellem to spillere, og det er derved en af spiller x tur.
Main flow: 1. Spiller x klikker på roll die. 2. En tilfældig værdi bliver genereret ud fra terningen. 3. Værdien bliver dernæst vist på skærmen. 4. Information omkring feltets egenskaber bliver vist. 5. Spillerens sum bliver manipuleret på baggrund af feltliste 6. Turen overgår til den anden spiller.
Postcondition: 1. Systemet er klar til at begynde spillet.
Alternativ flow: None

Use case: Afgør vinder
ID: 04
Kort beskrivelse af use casen: Vinderen af spillet afgøres.
Primary actors: Spiller
Secondary actors: Ingen
Precondition: Det er ikke første tur.
Main flow: 1. Spillet spilles efter use case ID 01 2. En spiller lander på et felt 3. Terningens værdi udskrives på UI 4. Point fra felt lægges til total sum 5. Total sum bliver mere end 3000 kroner 6. Spilleren vinder 7. Der udskrives at spilleren har vundet i UI
Postcondition: Point gemmes til eventuel videreførsel i andet spil.

Use case: Felt tjek
ID: 05
Kort beskrivelse af use casen: Terningens værdi svarer til et felt på vores brætspil. Feltet har nogle egenskaber som påvirker spiller x sum på en positiv eller negativ vis.
Primary actors: Spiller
Secondary actors: Ingen
Precondition: 1. Der er blevet klikket på roll die og derved blevet genereret værdi.
Main flow: 1. Den genereret ternings værdi bliver holdt op mod feltlisten.

1.1 Feltliste:

Tower	+250
Crater	-100
Palace gates	+100
Cold Desert	-20
Walled city	+180
Monastery	0
Black cave	-70
Huts in the mountain	+60
The Werewall (werewolf-wall)	-80 men spilleren får en ekstra tur.

 2. Påvirkningen af spillerens sum udfra det feltet.
Postcondition: 1. Det har lige været spiller x og der blev gennemført en hel tur. 2. Det er nu den anden spillers tur.

Bilag 3

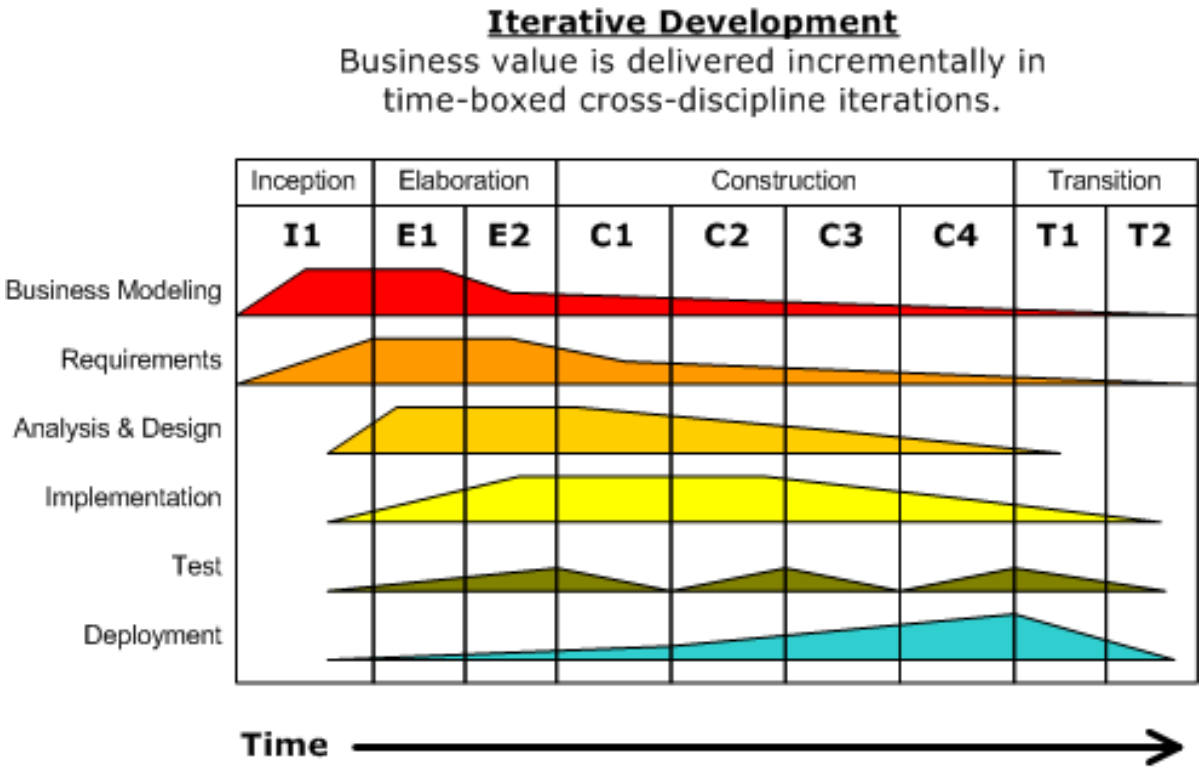
Traceability matrix

	R1	R2	R3	R4	R5	R6	R7	R8	R9
U1	x							x	x
U2		x			x				
U3		x					x		
U4						x			
U5			x	x					

Vi har benyttet os af en Traceability matrix for at sikre overensstemmelse mellem de specifikke krav og Use case scenarierne, og der dermed ikke er nogen krav som ikke bliver beskrevet gennem en use case.

Bilag 4

Vores plan for Udviklingsprocessen



Fase	E1
Opgaver	• Vision → Kravspecifikation - Frederik • Navneord/Udsagnsord Analyse - Kasper • Domænemodel - Bijan • Usecase diagram - Helene/Troles • Use Case Beskrivelse - Helene/Troles • Traceability Matrix - Helene/Troles Skriv dokumentation til ovenstående
deadline	Forventer 5 timer Dato: Onsdag klokken 12 i ferien. 19-10-2016 kl. 12

Fase	E2
Opgaver	• BCE - Bijan

	• Sekvens Diagram - Frederik • Design Class Diagram - Kasper • Design sekvens diagram - Helene/Troles Skriv dokumentation til ovenstående
deadline	Forventer 7 timer Dato: Mandag klokken 12 efter ferien. 24-10-2016 kl. 12

Aftalte fælles møder: Mandag klokken 12 efter ferien. 24-10-2016 kl. 12

Fase	C1 - Start development

Opgaver	• Etablere klasser og uddel ansvar ud fra klassediagramet
	• Konstruere main klasser
	  ○ Controller - Bijan
	  ○ Dicecup - Frederik
	  ■ Die
	  ○ Player - Troels (+ Helene)
	  ■ Bankaccount
	  ○ FieldList - Kasper
	  ■ Field
	  ○ Testklasse - Alle efter behov
	• Laver Junit test på Objekter. (fx terning)
	• Revidere diagrammer.
	
	Skriv dokumentation til ovenstående

deadline	Onsdag d. 26/10 23:59 Forventer 10 timer

Aftalte fælles møder:

Fase	C2 - Hovedkonstruktion færdiggøres. + Bugfix
Opgaver	• Evaluering af klasser og ansvar. • Evaluer main klasse • Revidere diagrammer. * Skriv dokumentation til ovenstående
deadline	Forventer 10 timer

Aftalte fælles møder:

Fase	C3 - Udbygning med ekstra features + Bugfix
Opgaver	• Udbygning af features * • Revidere diagrammer. * • JUnit test af ekstra features *

	• Evaluering af ekstra features! (go eller no go) * Skriv dokumentation til ovenstående
deadline	Forventer 5 timer

Aftalte fælles møder:

Fase	T1 - Færdiggør system.

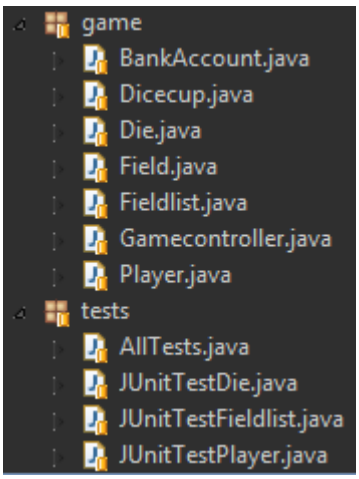
Opgaver	• Revidere Junit test og tilhørene dokumentation
	• Revidere diagrammer. (Er alle extra features der?)
	• Gennem Test system. + HotFix
	• Færdiggørelse af dokumentation.
	  ○ Konklusion
	  ○ Abstract
	  ○ Litteraturliste

deadline	Forventer 5 timer

Bilag 5

Oversigt over pakker og klasser

Projektet indeholder to pakker, med en række klasser. Game pakken indeholder selve spillet, hvor tests pakken indeholder forskellige tests af klasserne til programmet.



Figur x Oversigt over pakker og klasser

Som det ses på figur x indeholder programmet to pakker, nemlig “game” og “tests”.

Game pakkens indhold:

- BankAccount
- Dicecup
- Die
- Field
- Fieldlist
- Gamecontroller
- Player

Tests pakkens indhold:

- AllTests
    - JUnit Test Suite
- JUnitTestDie
- JUnitTestFieldlist
- JUnitTestPlayer