Projektopgave efterår 2012 - jan 2013
02312-14 Indledende programmering og 02313 Udviklingsmetoder til IT-Systemer.

Projekt navn: del1
Gruppe nr: 14.
Afleveringsfrist: Mandag den 22/10-2012 kl. 05:00
Denne rapport er afleveret via Campusnet (der skrives ikke under)
Denne rapport indeholder 29 sider incl. denne side.

Studie nr, Efternavn, Fornavne

s122996, Caspersen, Martin
Kontakt person (Projektleder)

s124334, Thideman, Anna

s123267, Svendsen, Caroline Sofie

s121701, Bjaarnø, Jonas Torbenfeldt

s124255, Kristiansen, Nicolai Kamstrup

del1							
Time-regnskab	Ver. 2012-10-18						
Dato	Deltager	Design	Impl.	Test	Dok.	Andet	Ialt
2012-09-25	Martin	2					2
2012-09-25	Anna	2					2
2012-09-25	Jonas	2					2
2012-09-25	Caroline	2					2
2012-10-02	Martin	2					2
2012-10-02	Anna	2					2
2012-10-02	Jonas	2					2
2012-10-05	Martin		4	1			5
2012-10-06	Nicolai		3				3
2012-10-08	Nicolai		4		1		5
2012-10-10	Martin		4	1			5
2012-10-12	Martin				5		5
2012-10-13	Martin				4		4
2012-10-13	Nicolai				2		2
2012-10-14	Martin				4		4
2012-10-16	Martin				4		4
2012-10-18	Martin				1		1
							0
	Sum	14	15	2	21	0	52

Indhold

1 Indledning[1]

Denne rapport er udarbejdet i kurserne 02313 Udviklingsmetoder til IT-systemer, 02312 Indledende programmering og 02314 Indledende programmering på første semester af Diplomingeniør IT. Opgaven er del et af tre CDIO opgaver på første semester, det overordnede formål med opgaverne på semesteret er at lave et Matador spil. Denne første opgave fokuserer på at implementere en terningklasse og en MatadorRafleBaeger klasse til at holde terning instanser. Den nærmere kravspecificering til opgaven findes i "CDIO_opgave_del11.pdf"[Nyb12], de relevante dele fra denne vil blive omhandlet i rapporten. Opgaven bygger i på undervisning modtaget i ovenstående kurser hvor der blev benyttet bøgerne [Lar04] og [LL12]. Igennem opgaverne vil klasser blive vist på følgende måde *KlasseNavn* og metoder på følgende måde `metodeNavn()`.

Rapporten er sat med LaTeXog alle UML diagrammer er lavet med TikZ-UML.

1.1 Ansvarsfordeling

Ansvarsfordelingen i opgaven er anført i overskrifterne til de forskellige afsnit og benytter følgende numre for at beskrive gruppens medlemmer:

1. Martin Caspersen (s122996)
2. Anna Thideman (s124334)
3. Caroline Sofie Svendsen (s123267)
4. Jonas Torbenfeldt Bjaarnø (s121701)
5. Nicolai Kamstrup Kristiansen (s124255)

Ansvarsfodelingen overskues nemmest i indholdsfortegnelsen. Hvis der er nummer på et af hovedafsnittene og ikke nogen numre på underafsnit er det pågældende medlem ansvarlig for alle underafsnittene også.

Ansvaret for kodningen af de forskellige klasser i vore færdige er beskrevet herunder:

- Die [1, 4, 5]
- MatadorRafleBaeger [1, 4, 5]
- Game [1]
- Player [1]
- GameController [1]
- BoundaryToGUI [1]
- BoundaryToPlayer [1]
- Main [1]

- TestMatadorRafleBaeger [1]

I forbindelse med ansvarsangivelsen for kodningen af vores endelige kode er der nogle ting der skal påpeges. Jonas og Nicolai har også kodet et fuldt virkende spil omend uden alle udvidelsesmulighederne. Vi valgte blot at benytte Martins kode i forbindelse med det færdige produkt, derfor er ansvarsfordelingen som anført.

2 UML inden programmeringen[1]

Inden programmeringen skulle følgende UML arbejde udføres.

> *I kan betragte ovenstående (afsnittet "Programmering af Matador terninger") som en kravspecifikation og det er jeres opgave at gøre følgende ved kravspecifikationen:*
>
> - *Lege djævlens advokat og påpege alle de områder på hvilken beskrivelsen er ufuldstændig. Med ufuldstændig mener vi f.eks. steder, hvor der er implicitte antagelser om viden hos jer som dem, der skal honorere disse krav (NB! Listen kan blive meget lang)*
> - *Omskriv kravspecifikationen, således at et passende specifikationsniveau er til stede.*
> - *Konvertér kravspecifikationen til følgende UML-diagrammer:*
>   - *Designklassediagram*
>   - *Sekvensdiagram [Nyb12]*

Ovenstående vil blive behandlet i dette afsnit.

2.1 Djævlens Advokat[1, 2, 3]

I dette afsnit vil vi gennemgå [Nyb12] for passager hvor beskrivelsen er ufuldstændig.

> *(...)der kan simulere kast med to terninger.[Nyb12]*

Det er uklart hvilken slags terninger der her er tale om, er terningerne tilfældige og hvor mange sider har terninger? Desuden nævnes terninger flere gange i dokumentet og det er ikke klart om det er de samme slags terninger der omtales hele vejen igennem.

Vi har besluttet at der er tale om sekssidet tilfældige terninger og det vil være terninger af denne type der er tale om i hele dokumentet.

> *Klassen skal indeholde en metode, getSum, (...), og en metode, getEns,(...)[Nyb12]*

Det er uklart om klassen må indeholde andre metoder og må der bruges andre klasser?

Vi har besluttet at klassen gerne må indeholde andre metoder og der må bruges alle de klasser vi har lyst til.

> *Implementér et spil med 2 deltagere.[Nyb12]*

Ved vi noget om de 2 deltagere, skal der gemmes noget information om dem?

Vi har besluttet at der ikke er nogen krav om at der skal gemmes noget information om deltagerne. Derfor må vi gemme den information vi finder nødvendig.

> *(...)spillet stopper når en spiller har opnået 25.[Nyb12]*

Det er her uklart hvad det er en spiller skal nå 25 af. Desuden har der ikke været beskrevet noget om at point skal gemmes mellem runder, det er svært med to terninger at nå 25 point uden at gemme værdier mellem runder.

Vi har besluttet at det er 25 point en spiller skal opnå og pointene skal gemmes mellem runderne.

> *Når en af spillerne (f.eks. spiller 1) opnår 25 point eller flere afsluttes spillet og flg. udskrives efter status:[Nyb12]*

Udover at der skal udskrives noget efter status skal der ske andet når spillet afsluttes?

Vi har besluttet at der er ikke er nogen krav om at der skal ske noget når spillet afsluttes udover at den givne tekst skal printes.

> *Hvis man slår 2 enere mister man alle sine point[Nyb12]*

Skal de 2 enere slås i samme tur eller mister man alle sine point hvis man i løbet af hele spillet har slået 2 enere?

Vi har besluttet at de 2 enere skal slås i samme tur for at man mister sine point.

> *Hvis der slås to ens får man ekstra tur.[Nyb12]*

Skal to ens slås i samme tur eller gælder det også hvis man har slået to ens i løbet af hele spillet? Hvad skal der ske hvis man slår 2 enere, skal man både miste sine point og få en ny tur?

Vi har besluttet at de to ens skal slå i samme tur får at man får en ekstra tur. Hvis man slår 2 enere mister man alle sine point og får en ny tur.

> *Hvis man slår 2 seksere to gange i træk slutter spillet, den pågældende spiller har vundet.[Nyb12]*

Skal de 2 seksere slås i samme tur eller skal man bare i alt i løbet af spillet have slået 4 seksere (2*2)?

Vi har besluttet at de 2 seksere skal slås i samme tur og det skal man slå to ture i træk for at vinde.

Med disse afklaringer på plads vil vi i næste afsnit skrive en kravspecifikation med et passende specifikationsniveau.

2.2 Kravspecifikation[1, 2, 3]

Dette afsnit indeholder en omskrivning af kravspecifikationen fra den udleverede opgave for at få et passende specifikationsniveau. Dette er baseret på de afklaringer vi har beskrevet i afsnit 2.1 på side 5. Derfor er der følgende kilde til dette afsnit: [Nyb12].

Til brug ved udviklingen af et Matador-spil i 3-ugers perioden ønskes en klasse ved navn *MatadorRafleBaeger*, der kan simulere kast med to terninger. Terningerne er tilfældige terninger med seks sider, altså vil alle seks værdier have lige stor chance for at blive slået. I resten af kravspecifikationen vil referencer til terninger være terninger af denne type. Klassen skal indeholde de to følgende metoder, `getSum`, som returnerer summen af øjnene på de to terninger, og `getEns`, som fortæller, hvorvidt de to terninger viser samme værdi. Disse metoder skal være public, mens datafelterne skal være private. Klassen må udover disse to metoder gerne indeholde andre metoder. Desuden må der gerne bruges yderligere klasser til at løse opgaven.

Hint: For at generere tilfældige tal kan man bruge metoden `nextInt(int n)` på klassen *java.util.Random*. Du kan læse i Java API'et hvordan metoden fungerer.

2.2.1 1.

Implementér et spil med 2 deltagere. Kaldet spiller 1 og spiller 2. Der er ingen krav om at der skal gemmes information om spillerne udover hvad der er nødvendigt for at spillet kan køre.
Man skiftes til at slå.
Der slås med 2, 6 siders terninger.
Det gælder om at slå flest øjne, og spillet stopper når en spiller har opnået 25 point. Antallet af point gemmes mellem hver runde for begge spillere, så der kan opnås de 25 point.

Spillet kan f. eks. afvikles ved følgende konsol dialog:

Det er spiller 1's tur. Tast 1 for at slå.
>1

Programmet udskriver antal øjne der slås samt total antal point for spiller 1 og 2, f.eks.:

Status
terning 1: 2, terning 2: 4
spiller 1: 6, spiller 2: 0

Herefter:
Det er spiller 2's tur. Tast 2 for at slå.

>2

Status
terning 1: 6, terning 2: 3
spiller 1: 6, spiller 2: 9

Når en af spillerne (f.eks. spiller 1) opnår 25 point eller flere afsluttes spillet og flg. udskrives efter status:
Spiller 1 har vundet!

Udvid spillet med følgende hvis tiden tillader der:

Der er 5 udvidelsesmuligheder som kan implementeres individuelt

A Hvis man slår 2 enere i samme kast mister man alle sine point.

B Hvis der slås to ens i samme kast får man ekstra tur.

C Hvis man slår 2 seksere, i samme kast, to gange i træk slutter spillet, den pågældende spiller har vundet.

D For at vinde spillet skal man have min. 52 point, point gemmes efter hvert træk og Herefter afslutte med 2 ens.

E Brug af udleveret GUI til at vise terningslagene. Man kalder funktionen `setDice(int faceValue1, int faceValue2)` i GUI-klassen.

2.2.2 2.

Lav en testklasse *TestMatadorRafleBaeger*, der anvender klassen *MatadorRafleBaeger* til at kaste terningerne et antal gange (fx 1000). Optæl, for hver mulig værdi af summen (2-12), antallet af forekomster. Optæl desuden antallet af kast, hvor terningerne er ens. Kontrollér at resultatet stemmer overens med de teoretiske sandsynligheder.

* bemærk at opgaven skal laves med et konsol interface, der må ikke anvendes GUI interface som vi ikke beskæftiger os med i dette kursus. (Undtaget er den udleverede GUI, der ikke må rettes i.)

Projektets struktur:

Lav et nyt workspace i eclipse med følgende struktur:

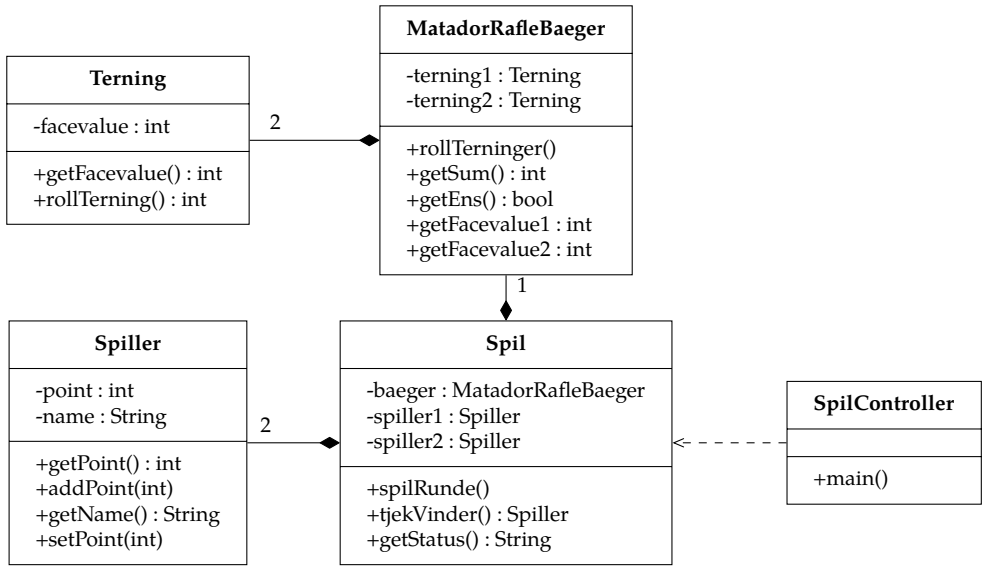
Workspace name: 02312 Project name: 06_del1 (vigtigt at 06 udskiftes med jeres gruppenummer)

Opret en pakke med navn matador_BusinessLogic hvor I placerer jeres klasser.

2.3 Designklassediagram[1]

På baggrund af kravspecifikationen udarbejdede vi det designklassediagram som ses i figur 2.1. Af diagrammet ses det at vi planlægger at have 5 klasser: *Terning*, *MatadorRafleBaeger*, *Spiller*, *Spil* og *SpilController*. *Terning* repræsenterer en fysisk terning og holder i variablen facevalue værdien af terningen, klassen indeholder også metoder til at rulle terningen og hente værdien ud. Vi planlægger at vores *MatadorRafleBaeger* skal indeholde to instanser af *Terning* og have metoder til at rulle alle terningerne, hente værdierne af dem ud og sammenligne dem. *Spil* skal indeholde en instans af *MatadorRafleBaeger* på den måde vil *Spil* ved hjælp af *MatadorRafleBaeger* have adgang til to *Terning* instanser. *Spil* skal også indeholde to instanser af *Spiller* som bruges til at indeholder oplysninger som point om de to fysiske spillere som skal deltage i spillet. *Spiller* har metoder til at tilføje og sætte point samt hente dem ud igen. *Spil* indeholder metoder til at stye selve spillet. Til sidst har vi *SpilController* som skal starte spillet ved at kalde *Spil*.

Figur 2.1: *Designklassediagram inden programmeringen.*



2.4 Sekvensdiagrammer[5]

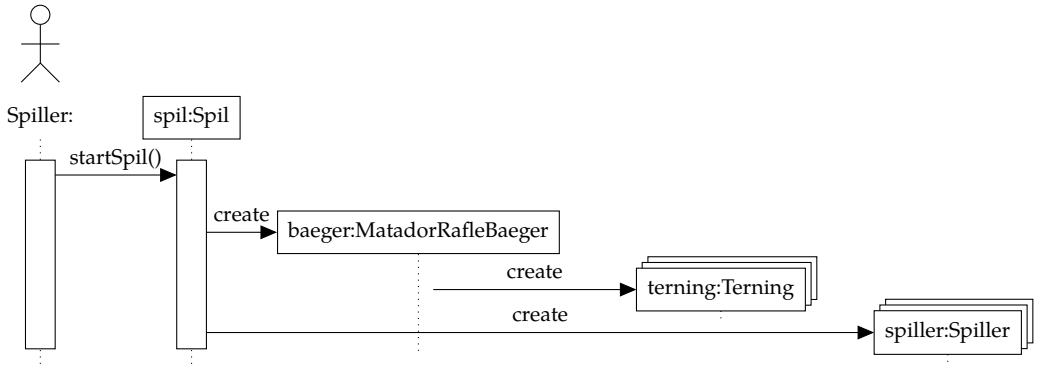
For bedre at kunne overskue sekvensdiagrammet for vores spil har vi valgt at opdele vores sekvensdiagrammet i to dele. Et diagram der viser starten af spillet og er beskrevet i afsnit 2.4.1. Det andet diagram viser forløbet af en runde og er beskrevet i afsnit 2.4.2 på næste side. I sekvens diagrammerne benytter vi de klasser og metoder som vi har beskrevet i designklassediagrammet som kan ses i figur 2.1.

2.4.1 Start af spillet

Sekvensdiagrammet for at starte et spil kan ses i figur 2.2 på næste side. For at starte spillet kalder den fysiske spiller metoden `startSpil()` på en instans af *Spil*. *Spil* op-

retter så en instans af `MatadorRafleBaeger` kaldet baeger. baeger instansen opretter to instanser af `Terning`. `Spil` opretter to instanser af `Spiller`. Nu er spillet klar til at der kan spilles en runde, sekvensdiagrammet for en runde behandles i afsnit 2.4.2.

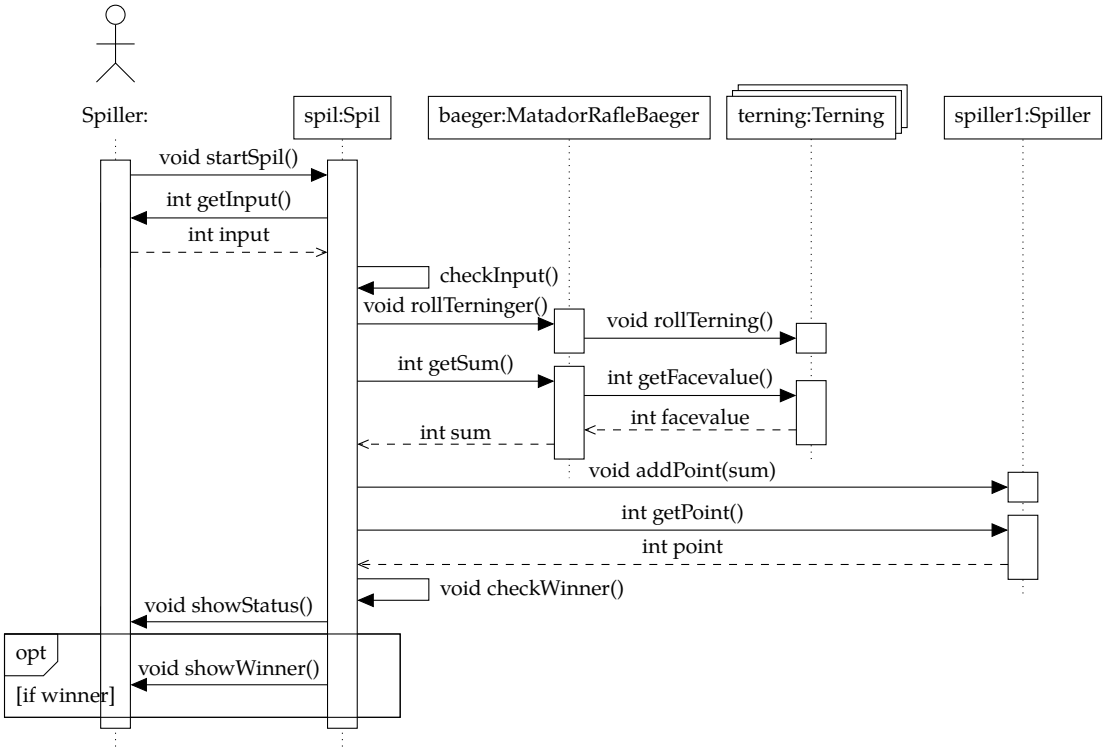
Figur 2.2: *Sekvensdiagram som viser starten af spillet.*



2.4.2 En runde

Inden der kan spilles en runde skal spillet være startet, sekvensdiagram for start af spillet bliver vist i afsnit 2.4.1 på foregående side. Sekvensdiagrammet for en runde er baseret på det scenarie hvor det er spiller 1 der har turen. Sekvensdiagrammet for spiller 2s tur ville ikke blive vist, eneste forskel er dog at instansen af `Spiller` i dette tilfælde vil hedde spiller2.

Figur 2.3: *Sekvensdiagram som viser en tur for spiller 1.*



I figur 2.3 på forrige side ses sekvensdiagrammet for en runde for spiller 1. Spillet er som tidligere nævnt allerede startet og derfor kræver *Spil* input fra den fysiske spiller 1. Den fysiske spiller 1 skal så i dette tilfælde taste 1 for at spille sin tur. Vi tager nu den antagelse at den fysiske spiller 1 taster 1. Når spiller 1 har tastet 1 kontrollerer spillet indtastning, efter denne er godkendt bliver terningerne rullet med et kald til `rollTerninger()` i baeger instansen af *MatadorRafleBaeger*. baeger kalder så `rollTerning()` på de to instanser af *Terning* som den indeholder. Efter dette kald kalder *Spil* baeger efter summen af *Terning* instansernes facevalues. baeger kalder de to *Terning* instanser med `getFacevalue()`. *Terning* instanserne returnerer deres facevalue som en int til baeger, de to int summeres og returneres til *Spil* som en int. *Spil* tilføjer denne int til spiller1 instansen ved at kalde `addPoint(sum)` på spiller1 instansen med summen som baeger instansen returnerede som parameter.

For at kontrollerer om der er en vinder af spillet kalder *Spil* nu spiller1 instansen med `getPoint()`, denne metode returnerer spiller1s point som en int. Denne returnerede værdi holdes op imod det antal point som kræves for at vinde. Spillets status printes ud i konsollen til den fysiske spiller. Hvis der er en vinder printes vinderen. Hvis der ikke er nogen vinder starter en ny runde hvor det er den fysiske spiller 2 som har turen, *Spil* vil så også skifte til at kalde spiller2 instansen af *Spiller*.

3 Programmering[1]

Dette afsnit af rapporten omhandler programmeringen af klassen *MatadorRafleBaeger* og andre nødvendige klasser for at leve op til kravene. Kravene blev nærmere omhandlet i afsnit 2.2 på side 7. Koden til alle klasserne kan ses i Eclipse projektet, alle klasserne ligger i pakken matador_BusinessLogic. For at køre spillet køres klassen *Main* i Eclipse, køres *Main* som den er, startes spillet hvor alle udvidelsesmulighederne fra opgaven er aktiveret. Hvis det ønskes at køre spillet i den simple konfiguration uden udvidelsesmuligheder udkommenteres `GameController game = new GameController();` og kommentar fjernes fra `GameController game = new GameController(2,0,false,-false,false,false,false);`. Se nærmere om udvidelsesmulighederne i afsnit 2.2.1 på side 7

TestMatadorRafleBaeger kan køres direkte i Eclipse, nærmere omkring denne klasse kan læses i afsnit 3.10 på side 19.

3.1 Overvejelser omkring klasser

Inden vi startede med at programmere gennemgik vi „CDIO_opgave_del11.pdf“ for at finde alle de krav som blev stillet til systemet. Dette førte til udarbejdelsen af kravspecifikationen som kan ses i afsnit 2.2 på side 7. I forbindelse med udarbejdelsen af kravspecifikationen lavede vi også et Designklassediagram og et sekvensdiagram. Ud fra disse to kunne vi få en idé om hvilke klasser vi regnede med at få brug for. Det bedes bemærket at vi efter at have lavet diagrammerne og imens vi kodede blev enige om at benytte engelske betegnelser hvor muligt i koden. Undtaget er selvfølgelig

MatadorRafleBaeger og metoderne `getEns()` og `getSum()` som var navngivet fra kravspecifikationen af.

3.2 Die

Denne klasse er i designklassediagrammet i afsnit 2.3 på side 9 kaldet Terning. I vores kode har vi valgt at kalde den for *Die* som er det engelske navn for én terning. Det ses fra designklassediagrammet at *Die* klassen skal indeholde én attribut, facevalue som skal være en int og være privat. Vi har designet vores terning så antallet af sider kan sættes i konstruktøren. Derudover skal den indeholde metoderne `int getFacevalue()` og `void rollTerning()` begge metoder skal være public. Klassen skal selvfølgelig også indeholde en konstruktør.

Herunder følger en kort beskrive af metoderne som vi valgte at implementere.

`public Die() & public Die(int)` er klassens konstruktører, som kaldes af *MatadorRafleBaeger* for at oprette en *Die*. Vi har valgt at den oprettede *Die* starter med at få en tilfældig værdi. Dette opnår vi ved at konstruktøren kalder *rollDie()* som bliver beskrevet senere i afsnittet. Hvis man sender en int parameter med vil denne angive hvor mange sider terningen skal have.

`public void rollDie()` bruges til at "rulle"*Die*, denne bruges både af klassen selv og af *MatadorRafleBaeger* derfor er det en public metode. For at give *Die* en tilfældig værdi benytter vi os af `nextInt()` fra *java.util.Random*.

`public int getFacevalue()` kaldes får at få værdien af *Die* instansen. *MatadorRafleBaeger* kalder denne metode og derfor er metoden public.

`public String toString()` bruges ikke, men kan repræsenterer *Die* som en String.

3.3 MatadorRafleBaeger

I designklassediagrammet i afsnit 2.3 på side 9 ses det at *MatadorRafleBaeger* skal indeholde to attributter af typen Terning kaldet terning1 og terning2. Derudover skal klassen indeholde de fem nedenstående metoderne `rollTerninger()`, `getSum()`, `getEns()`, `getFacevalue1()` og `getFacevalue2()` alle sammen public. Derudover skal klassen selvfølgelig også indeholde en konstruktør.

I forbindelse med undervisningen nåede vi inden projektet skulle afleveres at have om array og fik derfor lov at benytte disse i projektet. Som tidligere nævnt skiftede vi til engelske betegnelser. Derfor har vi valgt at implementere vores *MatadorRafleBaeger* sådan at det indeholder et array af *Die* instanser i stedet for kun to *Die* instanser. Dette giver mulighed for at *MatadorRafleBaeger* senere kan bruges i sammenhænge hvor der er behov for mere end to terninger i et spil. På grund af dette har vi i stedet kun en attribut som kaldes dice, denne er af typen et array af *Die* objekter og indeholder de *Die* in-

stanser der er behov for i den givne sammenhæng. Vi har valgt at *MatadorRafleBaeger* altid opretter instans af *Die* med seks sider.

Metoderne ændrer sig også lidt når et array bliver benyttet til at holde terningerne i stedet for. Vi valgte derfor at oprette nedenstående metoder:

`public MatadorRafleBaeger() & public MatadorRafleBaeger(int)` bruges til at konstruere et *MatadorRafleBaeger*. Uden parametre oprettes et *MatadorRafleBaeger* indeholdende to *Die* instanser. Med en int parameter bruges denne til at bestemme antallet af *Die* instanser som der bliver oprettet i *MatadorRafleBaeger*. Dette gøres ved at konstruktøreren sætter længden af arrayet til det antal *Die* instanser der ønskes, derefter kaldes `createDice()` som opretter *Die* instanserne i arrayet.

`private void createDice()` bruges af klassens konstruktører til at oprette *Die* instanserne i det oprettede array. Metoden er private da den kun skal bruges internt i klassen.

`public boolean getEns()` returnerer true hvis alle *Die* instanserne har samme værdi og false hvis ikke. Med én *Die* instans vil denne altid returnere true.

`public int getSum()` returnerer summen af alle *Die* instansernes facevalue.

`public void rollTerninger()` ruller alle *Die* instanserne. Dette gøres ved at `rollDie()` kaldes på hver enkelt *Die* instans som er indeholdt i arrayet.

`public int[] getFacevalues()` returnerer et array af ints som indeholder alle *Die* instansernes facevalue. Denne bruges af klasser som bliver omtalt senere til at sende *Die* instansernes facevalue til den udleverede GUI og konsollen.

`public String toString()` bruges ikke, men kan repræsenterer *MatadorRafleBaeger* som en String.

3.4 Game

Game klassen er i designklassediagrammet i afsnit 2.3 på side 9 beskrevet som Spil klassen. I diagrammet ses det at udover at indeholde en instans af *MatadorRafleBaeger* kaldet baeger skal *Game* også indeholde to instanser af klassen Spiller kaldet spiller1 og spiller2. Vi har ligesom med *MatadorRafleBaeger* valgt at benytte et array til at holde Spiller instanserne, derudover er Spiller klassen blevet omdøbt til engelsk. Derfor ender vi med at vores *Player* instanser er holdt i et array af *Player* instanser kaldet players. Deruodver havde vi inden programmeringen tænkt at den skulle indeholde metoderne `spilRunde()`, `tjekVinder()`, `getStatus()` og en konsturktør.

I forbindelse med kodningen af spillet oprettede vi yderligere instans variabler i *Game* klassen, som kort vil blive beskrevet. activePlayer angiver med en int hvilket index i players arrayet som har turen. winpoint angiver med en int hvor mange point

der skal til for at vinde spillet. baeger indeholder den instans af *MatadorRafleBaeger* som spillet benytter sig af. players er en array af *Player* instanser som deltager i spillet. winner angiver med en boolean om der er en vinder i spillet denne, er fra starten sat til false. twoOnes, twoSame, twoSixes og win52 angiver med boolean om udvidelsesmulighederne A-D er slået fra eller til.

Vi valgte under programmeringen at omdøbe metoderne til engelsk og tilføje nogle ekstra metoder, hovedsaligt for at implementere udvidelsesmulighederne.

`public Game() & public Game(int,int,int,bool,bool,bool,bool)` er klassens to konstruktører. Standardkonstruktøren starter et spil uden nogen valgmuligheder, spillet bliver startet med 2 *Player* og 2 *Die* instanser. Derudover starter "Spiller 1", alle udvidelsesmulighederne er slået til når denne konstruktør bruges. Vælger man i stedet at bruge den anden konstruktør kan man med de tre første int parametre vælge henholdsvis antallet af *Player* instanser, antallet af *Die* instanser og hvilken *Player* instans der skal starte spillet. De fire sidste boolean parametre kan sættes til false hvis man ønsker at slå henholdsvis udvidelsesmulighed A-D fra.

Begge konstruktører opretter en array af *Player* instanser med den nødvendige længde og bruger den private hjælpemetode `createPlayers()` til at oprette *Player* instanserne i arrayet.

`private void createPlayer()` bruges af klassens konstruktører til at oprette *Player* instanser i players arrayet. Metoden er private da den kun bruges internt i klassen.

`public void startGame()` bruges af *GameController* til at sætte spillet i gang, metoden er public da den skal kaldes udefra. Metoden starter spillet ved at kalde den private metode `oneRound()`.

`private void oneRound()` er ansvarlig for at spille en runde af spillet. Metoden er private da den kun kaldes internt fra klassen. Det første metoden gør er at få input fra den fysiske spiller ved hjælp af klasserne *GameController* og *BoundaryToPlayer* med static kald. Når den fysiske spiller i *BoundaryToPlayer* har indtastet den rigtige int vil `oneRound()` gå videre. Alle *Die* instanserne i *MatadorRafleBaeger* instansen bliver rullet og værdierne sendt til *GameController* som eventuelt sender dem videre til *BoundaryToGUI*. Efter dette tilføjes summen af *Die* instansernes facevalue til den aktive *Player* instans' point, `endRoundChecks()` kaldes nu som tager over for at kontrolleren hvad der skal ske efter turen.

`private void endRoundChecks()` kontrollerer alle udvidelsesmulighederne som er slået til. Metoden er også ansvarlig får at sende *Die* instansernes facevalues og *Player* instansernes point videre til *GameController* ved hjælp af `showStatus(int[],int[])`. GUI'en opdateres på lignende vis gennem kald til *GameController*. Det kontrolleres om der er en vinder og i så fald slutter spillet. Hvis der ingen vinder er findes den næste spiller som kan være enten den samme *Player* instans eller den næste i arrayet, kald til `nextPlayer()` er ansvarlig for at avancere turen.

`private void checkWinner()` kontrollerer om der er en vinder med de givne regler for spillet. Hvis der er en vinder sættes attributten winner til true. Denne attribut kontrolleres af `endRoundChecks()`.

`private void gameEnd()` anmoder *GameController* om at printe vinderen ud og ende spillet.

`private void nextPlayer()` sørger for at avancere activePlayer så den peger på den næste *Player* instans som skal have turen.

`private int[] getPlayerPoints()` returnerer et array af int som indeholder alle *Player* instansernes point. Pointene i arrayet vil have samme rækkefølge som *Player* instanserne har i players arrayet.

`public String toString()` bruges ikke, men kan repræsenterer status af *Game* som en String.

3.5 Player

I vores designklassediagram som ses i afsnit 2.3 på side 9 svarer *Player* til *Spiller*. Inden vi begyndte at kode var vores tanke at *Player* instanser skulle indeholde to variabler, en int med *Player* instansens point og en String med navnet. Derudover skulle klassen indeholde følgende metoder: `getPoint()` som returnerer en int, `addPoint(int)`, `getName()` som returnerer en String og `setPoint(int)`. Det skulle dog vise sig at vi fik behov for både flere instans variabler og flere metoder, hovedsaligt skyldes dette at vi valgte at implementere alle udvidelsesmulighederne.

Klassens instans variabler vil kort blive beskrevet. point beskriver med en int hvor mange point *Player* instansen har. carColor angiver med en int carColor i GUI'en, denne int svarer også til det index som *Player* instansen har i players arrayet. balance angiver med en int hvor mange penge *Player* instansen har, denne bruges ikke i spillet og findes kun fordi balance skal bruges når en *Player* ska oprettes i GUIen. name angiver med en String *Player* instansens navn, denne bliver heller ikke umiddelbart brugt i spillet men af hensyn til senere CDIO opgaver er det en fordel at kunne sende et navn til GUIen. twelveLastTime angiver med en boolean om *Player* instansen slog tolv i sidste tur, man kunne overveje om denne ikke ligeså godt kunne ligge i *Game* instansen. Dog vil man komme i problemer hvis man spiller med at man vinder ved at slå 12 to gange i træk men at man ikke får en ekstra tur når man slår to ens. På denne måde ved at knytte den op på den pågældende *Player* instans har vi sikret os mod sådanne problemstillinger.

`public Player(String, int, int)` konstruktøren til klassen sætter instansvariablen name til String parameteren, carColor sættes til den første int parameter og balance til den sidste int parameter. Derudover sættes point til 0 da man starter med 0 point, twelveLastTime sættes til false.

`public int getPoint()` returnerer værdien af point som en int.

`public String getName()` returnerer værdien af name som en String. Denne metode bruges ikke i vores spil som det er nu men er lavet for at man senere vil kunne hente navnet for en *Player* instans ud.

`public boolean getTwelveLastTime()` returnerer værdien af twelveLastTime som en boolean. Denne angiver om *Player* instansen slog tolv sidste tur.

`public void setTwelveLastTime(boolean)` sætter twelveLastTime til den boolean parameter som metoden er kaldt med.

`public void addPoint(int)` tilføjer den int parameter den er kaldt med til point. Dette kunne også være opnået ved hjælp af blot at have `setPoint()` og `getPoint()` som vi også har. Det er dog så ofte at der tilføjes point til en *Player* instans at vi har valgt at lave en metode specifikt til det.

`public void setPoint(int)` sætter point til den givne int. Nødvendig når man har den udvidelsesmulighed at man når man slår to enere skal miste sine point.

`public int getBalance()` returnerer værdien af balance som en int. Denne metode bruges ikke i vores spil men er lavet fordi den kan være nyttig i fremtidige spil.

`public void setBalance(int)` sætter værdien af balance til den int parameter som den er kaldt med. Denne metode bruges ikke i vores spil men kan måske bruges i fremtidige spil.

`public String toString()` bruges ikke, men repræsenterer *Player* som en String.

3.6 GameController

Denne metode er i vores designklassediagram som ses i afsnit 2.3 på side 9 kaldet SpilController og har ingen variabler og kun en main metode. Denne klasse har ændret sig noget i forbindelse med programmeringen af spillet. Oprindeligt var det tænkt at denne klasse blot skulle starte spillet ved at oprette en instans af *Game* klassen. Nu har *GameController* dog udover at starte spillet fået den rolle at koble *Game* klassen samme med *BoundaryToGUI* og *BoundaryToPlayer*.

I vores endelige implementering af *GameController* indeholder den en instans variabel kaldet activeGame som holder den aktive instans af *Game*. Derudover har klassen en static variabel kaldet gui af typen boolean som bruges til at styre om GUIen skal være aktiveret.

Metoderne i klassen er som følger:

`public GameController() & public GameController(int,int,bool,bool,bool,bool,bool)` er klassens to konstruktører. Standardkonstruktøren starter et spil med 2 *Die* og 2 *Player* instanser, alle udvidelsesmuligheder og GUIen er slået til. Med den anden konstruktør har man med de to første int parametre mulighed for henholdsvis at bestemme antallet af *Player* instanser og hvilken spiller der skal starte. De fem boolean parametre er som standard true, hvis de sættes til false kan man slå henholdsvis udvidelsesmulighed A-D og GUIen fra. Vi har valgt at man ikke skal kunne stille antallet af terninger da det ikke giver nogen mening for dette spil.

`public void startGame()` kalder blot `startGame()` i activeGame og derved startes spillet.

`public static void addPlayer(String,int,int)` sætter en *Player* instans på GUIen hvis gui er true. String parameteren er *Player* instansens navn, den første int parameter er balance og den sidste int parameter er carColor.

`public static void setCar(int,int)` sætter en *Player* instans' bil på GUIen hvis gui er true, *Player* instansen skal være oprettet i GUIen med addPlayer(String,int,int) beskrevet ovenfor. Den første int parameter beskriver *Player* instansens point, og den sidste int parameter beskriver carColor. carColor bruges af GUIen til at identificere en given *Player* instans.

`public static boolean getPlayerAccept(int)` videresender en int til *BoundaryToPlayer*. *BoundaryToPlayer* beder så spiller om at taste en int ind og hvis de matcher som de skal returnerer den true.

`public static void setDice(int[])` sender en array af int indeholdende *Die* instansernes facevalues til *BoundaryToGUI* såfremt gui er true.

`public static void showStatus(int[],int[]))` sender de to array af ints videre til *BoundaryToPlayer* som så printer status af spillet i konsollen til den fysiske spiller. Den første parameter er *Die* instansernes facevalues og den anden er *Player* instansernes point.

`public static void showString(String)` sender en String videre til *BoundaryToPlayer* som så printer denne String i konsollen til den fysiske spiller.

`public static void endGame()` kalder `closeScanner()` i *BoundaryToPlayer* som lukker scanneren der bruges til at få input fra den fysiske spiller.

3.7 BoundaryToGUI

Denne klasse figurerer ikke på vores designklassediagram fra inden vi begyndte at programmere som kan ses i afsnit 2.3 på side 9. Klassen har som formål at sende in-

formation videre til den udleverede GUI. Metoderne i denne klasse har samme navne som metoderne i boundaryToMatador fra den udleverede GUI. Alle metoderne i denne klasse er static og vil kort blive beskrevet herunder:

`public static void setDice(int[])` tager en array af ints som parameter som indeholder *Die* instansernes facevalues. Metoden sørger for at kalde setDice(int,int) metode i GUIen med to int som er de to *Die* instansers facevalue. Metoden vil kun udføre kaldet såfremt arrayen er præcis 2 lang.

`public static void addPlayer(String,int,int)` tilføjer en spiller til GUIen ved at kalde addPlayer(String,int,ing) i GUIen. String parameteren er navnet på *Player* instansen, den første int er balancen mens den sidste int er carColor.

`public static void setBalance(int,int)` sætter balancen i GUIen. Vi bruger ikke metoden i vores spil, derfor findes der ikke en tilsvarende metode i *GameController*.

`public static void setCar(int,int)` tager to ints som parametre, den første beskriver *Player* instansens point og den anden carColor. Pointene omregnes til et felt nummer i GUIen med følgende formel: point%40 + 1.

3.8 BoundaryToPlayer

Denne klasse figurerer ikke på vores designklassediagram fra inden vi begyndte at programmere som kan ses i afsnit 2.3 på side 9. Klassen bruges til at kommunikere med den fysiske spiller ved hjælp af konsollen. Klassen har en private static variabel kaldet input som er den Scanner der bruges til at modtage input fra den fysiske spiller. Metoderne beskrives herunder.

`public static boolean getPlayerAccept(int)` modtager en int som parameter, denne int repræsenterer den aktive *Player* instans index i players arrayet over *Player* instanser. Derfor adderes denne int med en og den fysiske spiller anmodes om at indtaste denne værdi. Metoden vil blive ved med at spørge efter input indtil den fysiske spiller indtaster den rigtige int hvorefter den returnerer true. Hvis der indtastes andet end en int vil man få en Exception.

`public static void showString(String)` printer String parameteren til konsollen.

`public static void closeScanner()` kalder `close()` på input som blev benyttet til at tage input fra den fysiske spiller.

`public static void showStatus(int[],int[])` modtager to parametre af typen array af ints, første er *Die* instansers facevalue sidste er *Playaer* instansers point. Med de to parametre printer denne metode status af spillet til konsollen.

3.9 Main

Denne metode har til formål at starte spillet, den starter med at lave en instans af *GameController* kaldet game. Afhængig af hvilken konstruktør der bruges kan ekstra konditionerne som bliver omtalt i afsnit 3.4 på side 13 slås fra. Når instansen af *GameController* er initialiseret kalder klassen `startGame()` på game og dermed startes spillet.

3.10 TestMatadorRafleBaeger

TestMatadorRafleBaeger klassen er ikke beskrevet i nogen af de diagrammer som vi udarbejdede inden programmeringen. Kravene til *TestMatadorRafleBaeger* er givet i afsnit 2.2.2 på side 8

For at opnå dette har vi lavet *TestMatadorRafleBaeger* som kun indeholder en metode, `public static void main(String[] args)`. Det betyder at klassen direkte fra Eclipse kan køres og *MatadorRafleBaeger* kan dermed testes for at se om den er tilfældig. I implementeringen af klassen har vi valgt at slå 100000 slag i stedet for 1000 slag, dette giver et bedre billede af om den er tilfældig. Ved et lavt antal slag vil det være muligt for mange af slagene at have den samme værdi selvom klassen er tilfældig. Det er selvfølgelig også muligt lige meget hvor mange slag man slår men sandsynligheden for at en stor procentdel af slagene bliver det samme er mindre ved et større antal slag.

Til at holde alle de forskellige antal slag har vi valgt at benytte et array af ints med længden 11. Der er nemlig 11 forskellige muligheder for summen af øjnene af to terninger, (2-12). Derefter oprettes et *MatadorRafleBaeger* med to terninger og en final variabel som indeholder antallet af slag som skal slås.

Efter alle variablerne er initialiseret, initialiseres to forskellige *NumberFormat* til at formatere tallene vi får ud i sidste. Vi laver et *NumberFormat* til procenttal og et til at formatere summen af terningerne i formatet 02-12.

Dernæst slås de 100000 slag, for hvert slag øges tallet i index <summen af øjnene> - 2, det vil sige summen 2 giver index 0 og summen 12 giver index 10.

Til sidst printes procentfordelingen og noget tekst som er vores tanker omkring om klassen er tilfældig.

Hvis vores *TestMatadorRafleBaeger* køres ses det at tallene ser meget tilfældige ud. Til det niveau vi er på nu er de også tilfældige nok, de er dog kun pseduotilfældige. Computere er ikke i stand til at være tilfældige, den eneste måde at få et sandt tilfældigt tal på er ved at indføre en måling eller lignende af et tilfældigt fysisk fænomen. java.util.Random som vi benytter til at få vores "tilfældige"terningslag beskriver konstruktøren kaldet uden parametre således:

> *Creates a new random number generator. This constructor sets the seed of the random number generator to a value very likely to be distinct from any other invocation of this constructor. [Ora12]*

Som det ses er det kun "very likely"at det seed som vores tilfældige tal generes fra er forskelligt fra et andet kald til denne konstruktør. Seedet bruger java til at lave en række

matematiske operationer på for at finde et "tilfældigt"tal.

3.11 Overvejelser til fremtidige opgaver

En ting som vi tydeligt så i forbindelse med denne opgave var hvor meget de design vi havde inden vi gik i gang med at programmere ændrede sig. Det skyldtes i stor grad vores manglende erfaring men også det at vores oprindelige diagrammer ikke havde udvidelsesmulighederne med.

En anden ting er at hvis vi rigtigt skal benytte os af boundaries kunne det være en idé at sende instanserne som man gerne vil vise til boundaryen. På den måde kan boundaryen selv stå for at hente informationen ud og formatere den på den måde som den ønsker.

Hvis vi skal have printet strings fra andre Klasser ville en god måde at gøre det på nok være at sende et keyword til boundaryen. Boundaryen kunne så med dette keyword slå op og se hvad der skulle printes ud. På denne måde kunne boundaryen virkelig styre hvad der skulle printes. Det ville også gøres vores kode robust hvis man senere skulle få lyst til at tilføje for eksempel flere sprog.

4 UML efter programmering

Når I har løst programmeringsopgaven, bedes I lave UML-diagrammer som dokumenterer Jeres program. I bedes gennemføre følgende aktiviteter:

- *Udarbejde UML-diagrammer (designklassediagram og sekvensdiagram), som svarer til jeres program*
- *Finde en anden gruppe med hvilken I bytter jeres kode og UML-diagrammer*
  - *Vurdere om deres UML-diagrammer var tilstrækkeligt grundlag for vurdering af, om I kunne anvende deres Bæger klasse (MatadorRafleBaeger)[Nyb12]*

Ovenstående vil blive behandlet i dette afsnit.

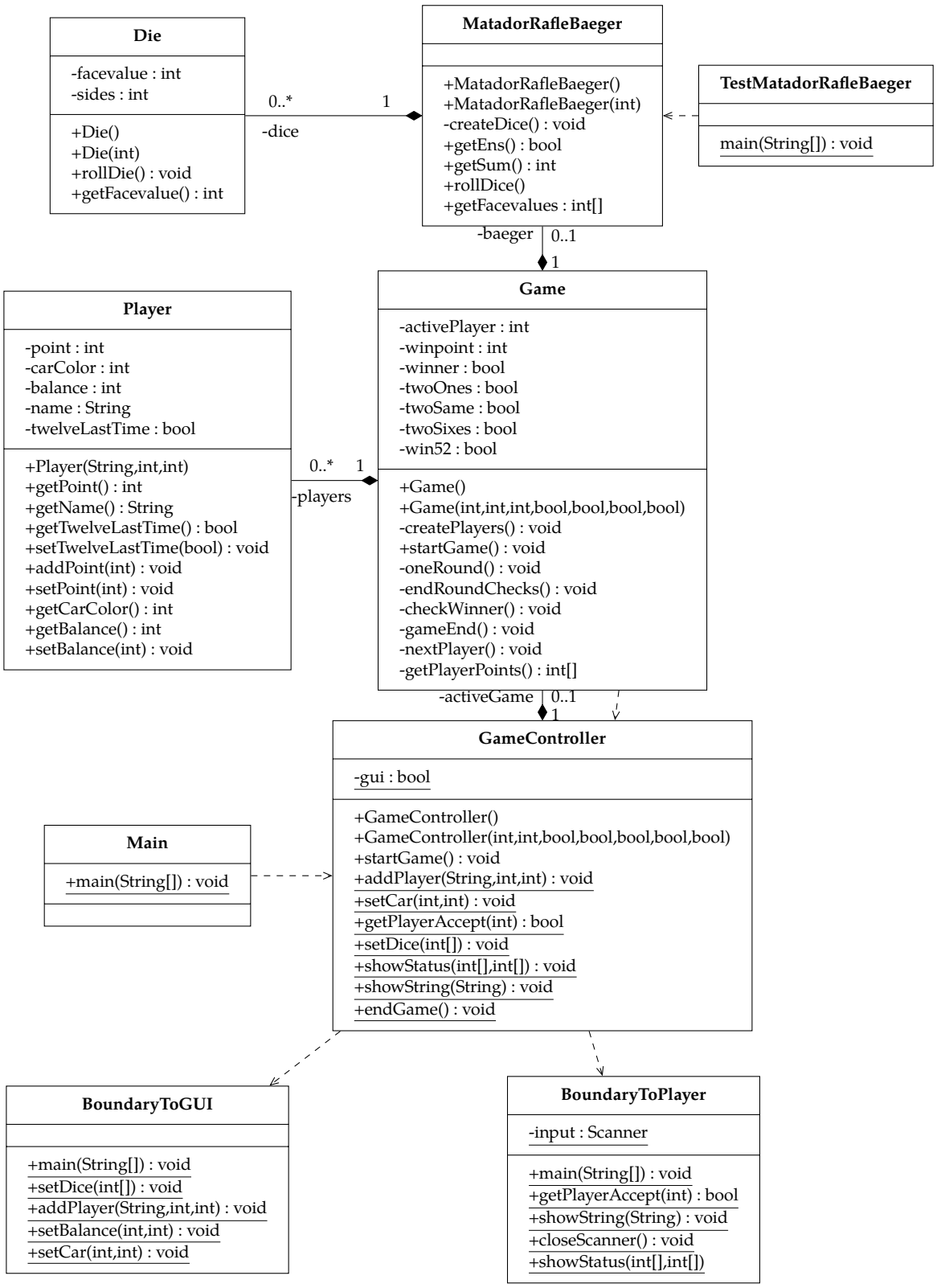
4.1 Designklassediagram[1]

I figur 4.1 på den følgende side ses designklassediagrammet som vi udarbejdede efter programmeringen var færdig. Dermed viser diagrammet relationerne mellem klasserne i vores færdige program. Det ses at der er store forskelle i forhold til diagrammet fra før programmeringen som kan ses i figur 2.1 på side 9. De største forskelle er at der er kommet flere klasser og metoder til, mange af disse skyldes at vi valgte at implementere alle udvidelsesmulighederne i vores kode.

Vi vil nu gennemgå diagrammet og beskrive relationerne. Kravene til koden som kan ses i afsnit 2.2 på side 7 stiller kun krav om at der er en `MatadorRafleBaeger` klasse. Vi fandt det dog mest logisk at repræsenterer de terninger som der skal simuleres i `MatadorRafleBaeger` ved instanser af en klasse vi har valgt at kalde `Die`. Vores

`MatadorRafleBaeger` holder disse instanser i et Array fordi vi på denne måde har mulighed for nemt at variere antallet af terninger som skal bruges.

Figur 4.1: *Designklassediagram efter programmeringen*



`MatadorRafleBaeger` bruges af klassen `TestMatadorRafleBaeger` og som navnet antyder er denne klasses formål at teste `MatadorRafleBaeger`. `TestMatadorRafleBaeg-`

er bruges ikke i spillet.

Vi har en *Game* som repræsenterer det spil vi skulle implementere, denne klasse indeholder en instans af *MatadorRafleBaeger*. Dermed har hver instans af *Game* adgang til to terninger som netop var hvad der skulle bruges i dette spil.

Derudover har vi valgt at repræsenterer de fysiske spillere i vores system med *Player* klassen. Hver instans af klassen har mulighed for at indeholde en række oplysninger til dette spil er det hovedsaligt pointene der er vigtige. Der vil typisk være to spillere med i spillet, men vi har mulighed for at variere dette da vi ligesom med *Die* holder vores *Player* instanser i et Array i *Game*.

Vi har en *GameController* klasse som står for at starte spillet og kommunikation mellem *Game* og de to boundary klasser vi har. *GameController* indeholder én instans af *Game* og der er ikke mulighed for at den kan indeholde flere instanser. Det ses også at *Game* afhænger af *GameController* det skyldes at *Game* kalder statiske metoder i *GameController*.

Selve spillet bliver startet af *Main* klassen som i main metode opretter en instans af *GameController* og kalder de nødvendige metoder for at starte spillet.

GameController afhænger af *BoundaryToPlayer* og *BoundaryToGUI* fordi den kalder metoder i klasserne for at kommunikere med henholdsvis den fysiske spiller og GUIen.

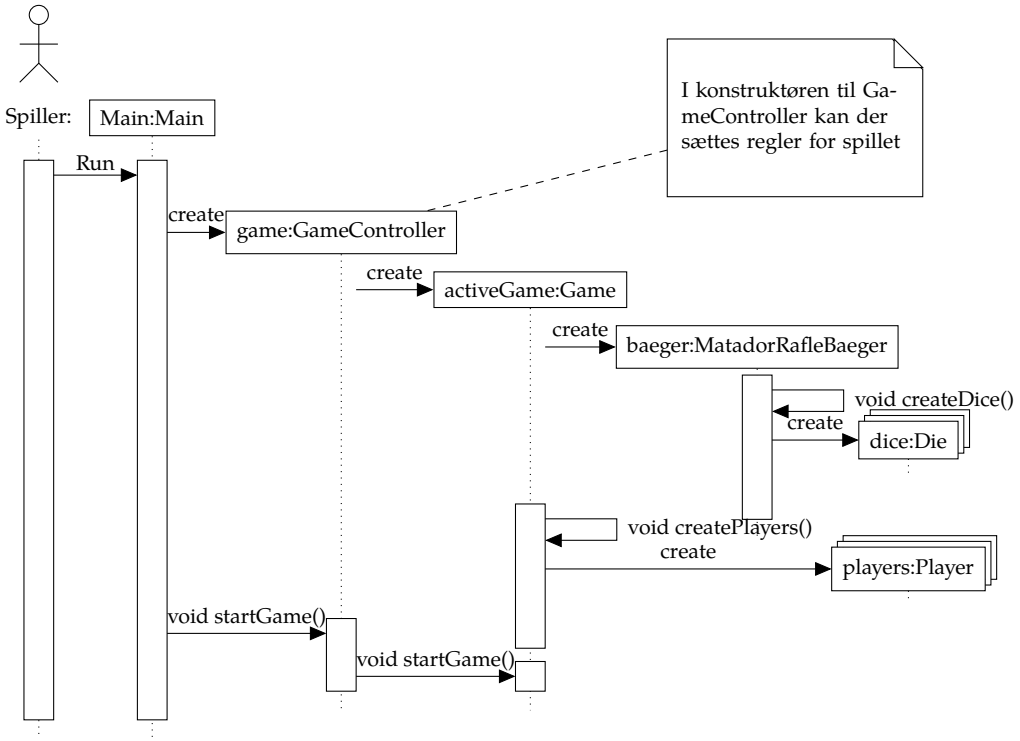
4.2 Sekvensdiagrammer[5]

For at beskrive interaktionen mellem klasserne som bliver vist i figur 4.1 på foregående side. Har vi udarbejdet sekvensdiagrammer for vores kode efter programmeringen. Vi har for overskuelighedens skyld valgt at opdele sekvensdiagrammet i to dele. Én sekvensdiagram for starten af spillet som beskrives i afsnit 4.2.1 og ét diagram for spil af en runde som beskrives i afsnit 4.2.2 på næste side.

4.2.1 Start af spillet

Som sekvensdiagrammet i figur 4.2 på den følgende side viser; startes programmet ved at den fysiske spiller kører *Main*. *Main* den skal opretter en instans af *GameController*, her bestemmes hvilke regler der skal sættes for spillet. Videre opretter *GameController* en instans af *Game*, som opretter en instans af *MatadorRafleBaeger*, som opretter det ønskede antal instanser af *Die*. Dette udgør hoved ingredienserne i selve ternings spillet og skal danne grundlag for selve spillets udførelse. Dog mangler repræsentationen af spillerne, *Game* opretter det ønskede antal instanser af *Player*. Når alle de nødvendige instanser er oprettet går kontrollen tilbage til *Main* som kalder metoden `startGame()` på instansen af *GameController*. *GameController* kalder så `startGame()` på instansen af *Game*.

Figur 4.2: *Sekvensdiagram som viser starten af spillet.*



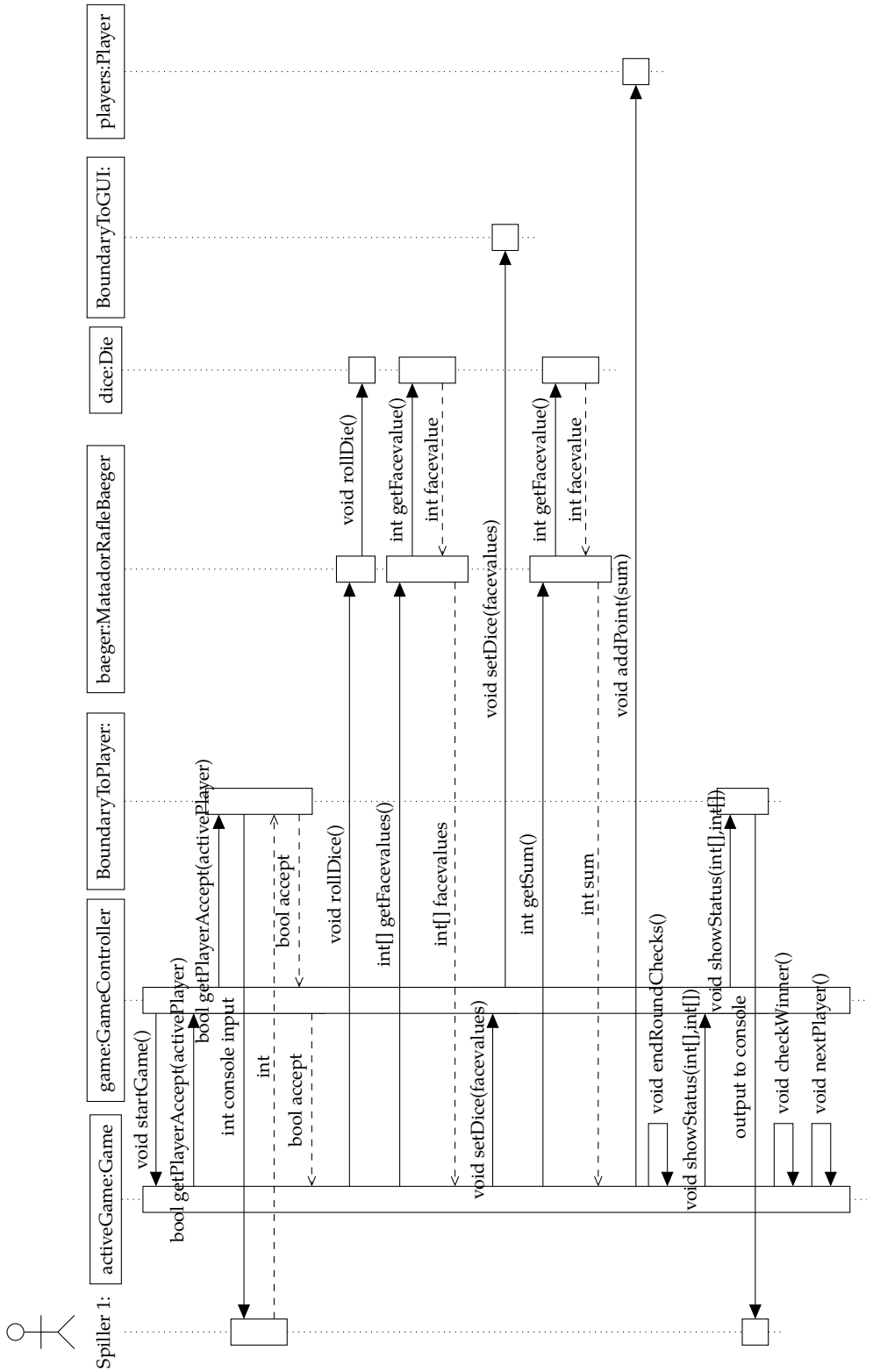
4.2.2 En runde

Inden vi gennemgår sekvensdiagrammet for en runde af spillet er der nogle betingelser der skal stilles op. Først og fremmest skal spillet være startet, sekvensdiagram for dette bliver behandlet i afsnit 4.2 på foregående side. Derudover viser vores sekvensdiagram for en runde det forløb hvor der er Spiller 1 der har turen. Ingen af udvidelsesmulighederne vi har implementeret bliver nærmere behandlet i sekvensdiagrammet men kontrollen af disse sker i *Game* klassens `endRoundChecks()` metode. Vi har også i koden koblet GUI på men viser kun hvordan terningerne bliver vist, ikke bilerne. Til sidst antager vi at der ikke bliver fundet en vinder og at turen skal gå videre til Spiller 2. Sekvensdiagrammet starter med det sidste kald fra sekvensdiagrammet for start af spillet nemlig at *GameController* instansen kalder `startGame()` på *Game* instansen. Diagrammet ses i figur 4.3 på den følgende side.

Sekvensdiagrammet viser hvordan én runde for spiller 1 forløber; Først anmoder *Game* via *GameController* og *BoundaryToPlayer* den fysiske spiller, om i dette tilfælde at trykke "1" for at slå terningerne. Så sender *Game* anmodningen om at slå med terningerne via `rollDice()` til baeger instansen, som kalder `rollDie()` på instanserne af *Die*. *Game* anmoder herefter baeger instansen om værdierne for slaget via `getFacevalues()`, baeger instansen tjekker herefter dette i instanserne af *Die*. Disse informationer sendes til *GameController* og derfra videre til *BoundaryToGUI* med `setDice(int[])`. Herefter kalder *Game* `getSum()` på baeger instansen for at hente værdien af terningslaget, baeger instansen henter de nødvendige oplysninger fra instanserne af *Die*. *Game* bruger så `addPoint()` for at lægge summen af terningerne til den instans af *Player* der repræsenterer spiller 1. Til sidst udfører *Game* 3 opgaver med hjælpemetoder, `endRoundChecks()`

kontroller udvidelsesmuligheder og hvem der skal have næste tur, i `endRoundChecks()` kaldes `checkWinner()` for at kontrollerer om der er en vinder. Om nødvendigt kaldes `nextPlayer()` for at avancere turen til næste instans af *`Player`*. Efter dette starter runden forfra med den instans af *`Player`* som repræsenterer spiller 2 i stedet.

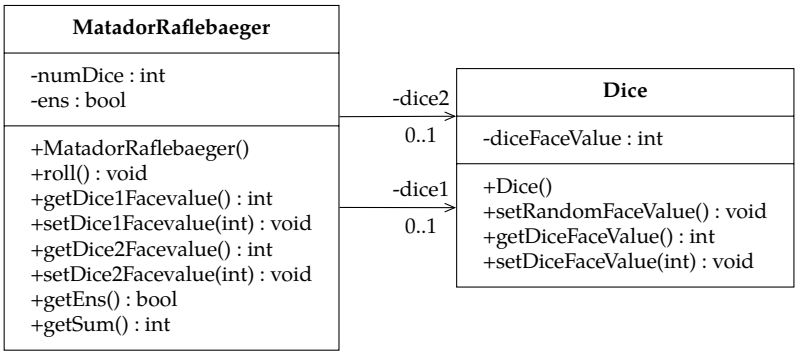
Figur 4.3: *Sekvensdiagram som viser en runde af spillet.*



4.3 Vurdering af anden gruppes UML og kode[1]

For at vurdere en anden gruppes UML og kode har vi lånt et designklassediagram fra gruppe 17 som viser deres *MatadorRaflebaeger* og *Dice* klasser. I figur 4.4 ses designklassediagrammet. Af diagrammet kan det ses at de har valgt at implementere deres *MatadorRaflebaeger* med separate variabler for de to terninger som er nødvendige for dette spil. Vi bemærker også at deres klasse ikke hedder *MatadorRafleBaeger* som vores gør, deres klasse hedder *MatadorRaflebaeger* (bemærk lille b i baeger). Vi vil starte med at kigge på metoderne der findes i deres *MatadorRaflebaeger*.

Figur 4.4: *Designklassediagram fra Gruppe 17 som viser MatadorRaflebaeger og Dice klasserne.[1712]*



Det ses at Gruppe 17 har en konstruktør som ikke tager nogen parametre ligesom vi har. Koden for deres konstruktører for *MatadorRaflebaeger* og *Dice* ses i figur 4.5. Det ses at deres konstruktør for *MatadorRaflebaeger* som forventet opretter to terningers objekter. I konstruktøren for deres *Dice* klasse ses dog en forskel fra vores *Die* klasse, de giver ikke terningen en værdi når den bliver konstrueret. Det ændrer dog ikke på vores kode eftersom vi alle steder i vores kode ruller terningerne inden vi bruger værdier fra dem. Vi vil altså uden ændringer i vores kode kunne bruge deres konstruktør `MatadorRaflebaeger()`.

Figur 4.5: *Konstruktør for MatadorRaflebaeger og Dice[1712]*

```
public MatadorRaflebaeger(){
    this.dice1 = new Dice();
    this.dice2 = new Dice();
}
```

```
public Dice(){
}
```

Næste metode vi vil kigge på er `roll()` som vi formoder svarer til vores `rollDice`. De ses at ingen af metoderne returnerer en værdi og derfor er der indtil videre ingen problemer med at bruge deres metode. Det ses i figur 4.6 på næste side at `roll()` i *MatadorRaflebaeger* kalder `setRandomFaceValue()` på begge terningerne i deres *MatadorRaflebaeger*. For at vi kan være sikre på at vi kan bruge deres `roll()` skal vi lige sikre os at `setRandomFaceValue()` nu også gør som navnet antyder og giver terningen en tilfældig facevalue.

Koden for gruppe 17s `setRandomFaceValue()` ses i figur 4.7 på den følgende side og det ses at de bruger sammen metode som os til at give deres terninger en ny tilfældig

Figur 4.6: *`roll()` metoden fra `MatadorRaflebaeger`[1712]*

```
public void roll() {
    dice1.setRandomFaceValue();
    dice2.setRandomFaceValue();
}
```

facevalue. Vi vil altså kunne bruge deres `roll()` i stedet for vores `rollDice()` ved at ændre alle vores kald til `rollDice()` så de i stedet går til `roll()`.

Figur 4.7: *`setRandomFaceValue()` fra `Dice`[1712]*

```
public void setRandomFaceValue(){
    Random rand = new Random();
    this.diceFaceValue = rand.nextInt(6)+1;
}
```

Næste metode vi vil kigge på er deres `getEns()` som svarer til vores `getEns()`. Begge metoder returnerer en boolean så på overfladen er der ingen problemer. Koden for både `getEns()` og `getDiceFaceValue()` som `getEns()` kalder ses i figur 4.8. Fra koden ses det at `getDiceFaceValue()` som den skal returnerer den givne ternings facevalue. `getEns()` vil så returnerer true hvis de to terningers facevalue er ens. Det er sådan vi forventer at `getEns()` virker og vi vil uden ændringer i vores kode kunne bruge denne metode.

Figur 4.8: *`getEns()` og `getDiceFaceValue()` metoderne fra henholdsvis `MatadorRaflebaeger` og `Dice`.)[1712]*

```
public Boolean getEns() {
    return dice1.getDiceFaceValue() == dice2.getDiceFaceValue();
}

public int getDiceFaceValue(){
    return this.diceFaceValue;
}
```

Så er turen kommet til at kigge på `getSum()` hvor vi igen i begge grupper har metoder af samme navn. Vi forventer at denne metode returnerer summen af de to terningers facevalue, fra diagrammet ser det godt ud da den er sat til at returnerer en int, ligesom vores metode gør. Et lille kig i koden og vi kan med sikkerhed afgøre om koden gør som vi antager, koden for `getSum()` kan ses i figur 4.9 på den følgende side og `getDiceFaceValue()` kan ses i figur 4.8. Det ses hurtigt at gruppe 17s `getSum()` metode som forventet korrekt returnerer summen af de to terningers facevalue. Vi vil derfor uden ændringer i vores kode kunne benytte os at `getSum()` metoden i vores program.

Figur 4.9: *getSum() fra MatadorRaflebaeger.)[1712]*

```
public int getSum(){
    return dice1.getDiceFaceValue() + dice2.getDiceFaceValue();
}
```

Til sidst kommer så den forskel at vi har valgt at implementere vores *MatadorRafleBaeger* sådan at vores terninger (*Die*) bliver holdt i et Array af *Die* objekter. Dette giver nogle forskelle i hvordan *MatadorRafleBaeger* har implementeret de forskellige metoder. Det betyder dog ikke noget for de funktioner som afhænger af *MatadorRafleBaeger*. Sålænge alle metoderne i *MatadorRafleBaeger* er implementeret korrekt og virker som de skal vil brugerne af *MatadorRafleBaeger* ikke mærke forskel. Her er der dog en forskel i den måde som terningerne facevalue hentes på. I vores *MatadorRafleBaeger* hentes terningernes facevalue som en Array af ints, i gruppe 17s tilfælde hentes de to facevalues med `getDice1Facevalue()` og `getDice2Facevalue()`. Den nemmeste måde at undersøge hvilke ændringer dette medfører i vores kode, er i Eclipse at benytte sig af funktionen hvor man kan se hvor i koden der er referencer til en given metode. Det ses at vi har to referencer til `getFacevalues()` i Game klassen. De to referencer er i `endRoundChecks()` og `oneRound()`, disse to metoder sender blot informationerne videre til *GameController* som så sender dem videre til *BoundaryToGUI* og *BoundaryToPlayer*. Man kan ændre i de berørte metoder i *GameController*, *BoundaryToGUI* og *BoundaryToPlayer* og i *Game*. Vi var sluppet uden om dette problem hvis vi havde sendt baeger objektet til *GameController* og senere til de to boundaries. På den måde ville en ændring i *MatadorRafleBaeger* kun have ført til en ændring i den måde som boundaryen skulle hente infomationerne ud af *MatadorRafleBaeger*. I figur 4.10 ses gruppe 17s kode for at hente de to facevalues ud. Selvom koden er lidt forskellig virker begge måder at skrive koden på og den virker som vi forventer.

Figur 4.10: *getDice<num>FaceValue fra MatadorRafleBaeger.[1712]*

```
public int getDice1FaceValue() {
    return this.dice1.getDiceFaceValue();
}

public int getDice2FaceValue() {
    return dice2.getDiceFaceValue();
}
```

Til at slutte af med kan man sige at for langt de fleste af metoderne vil det være simpelt at benytte gruppe 17s *MatadorRaflebaeger* i stedet for vores eget. I ét tilfælde, `roll()`, skal alle vores kald til `rollDice()` laves om til i stedet at kalde `roll()`, dette klares hurtigt i Eclipse. Lidt større vanskeligheder er der når terningernes facevalue skal hentes ud af *MatadorRafleBaeger*. Det skyldes nok også at med `roll()`, `getEns()` og `getSum()` ligger der i navnet hvordan de skal virke. En metode som skal hente face-

values ud er tilgengæld meget åben for fortolkning og kan implementeres på mange forskellige måder.

5 Konklusion[1]

Vi har i løbet af opgaven fået implementeret spillet som kravspecifikationen krævede. Derudover implementererede vi alle udvidelsesmulighederne til spillet. Vi fandt hurtigt ud af at de UML diagrammer vi havde lavet inden programmeringen ikke gav noget særligt godt billede af hvordan koden endte med at se ud. Vores UML diagrammer efter programmeringen afspejler selvfølgelig koden som den endte med at se ud.

Vi implementererede også `TestMatadorRafleBaeger` for at teste om `MatadorRafleBaeger` er tilfældig som den skal være. Det viste sig vores `MatadorRafleBaeger` er tilfældigt nok til det behov vi har lige nu. Vi er dog også klar over at den måde vi har implementeret `Die` på betyder at det ikke er sande tilfældige tal vi får.

Referencer

[1712] Gruppe 17. *CDIO del 1*. 12. okt. 2012.

[Lar04] Craig Larman. *Applying UML and Patterns. An Introduction to Object-Oriented Analysis and Design and Iterative Development*. 3. udg. Prentice Hall, 2004.

[LL12] John Lewis og William Loftus. *Java Software Solutions Foundations of Program Design*. 7. udg. Peason Education, 2012.

[Nyb12] Mads Nyborg. *Del opgave 1. Simpel anvendelse af klasser i Java og UML*. Vers. 2012-08-08. Opgavebeskrivelse udleveret i forbindelse med projektet. DTU, 8. aug. 2012.

[Ora12] Oracle. *Class Random*. 11. okt. 2012. URL: `http://docs.oracle.com/javase/7/docs/api/java/util/Random.html` (sidst set 11.10.2012).