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

Projekt navn: del2
Gruppe nr: 21.
Afleveringsfrist: Mandag den 12/11-2012 kl. 05:00
Denne rapport er afleveret via Campusnet (der skrives ikke under)
Denne rapport indeholder 23 sider incl. denne side.

Studie nr, Efternavn, Fornavne

s122996, Caspersen, Martin
Kontakt person (Projektleder)

s100182, Baltzersen, Jesper Engholm

CDIO del 2							
Time-regnskab	Ver. 2012-11-09						
Dato	Deltager	Design	Impl.	Test	Dok.	Andet	Ialt
2012-10-23	Martin	1	2				3
2012-10-23	Jesper	1	1				2
2012-10-25	Martin			1	1		2
2012-10-25	Jesper			1			1
2012-10-26	Martin	1	4				5
2012-10-26	Jesper	1	3				4
2012-10-30	Martin		3				3
2012-10-30	Jesper		1				1
2012-10-31	Martin		2		1		3
2012-11-02	Martin	2	1	1	2		6
2012-11-02	Jesper			2	3		5
2012-11-03	Martin	4	1	1	2		8
2012-11-04	Jesper				2		2
2012-11-05	Jesper				3		3
2012-11-06	Jesper				2		2
2012-11-06	Martin				2		2
2012-11-07	Martin				3		3
2012-11-08	Martin				3		3
2012-11-09	Jesper				2		2
2012-11-09	Martin				2		2
	Sum	10	18	6	28	0	62

Indhold

1 Indledning[1]

Denne rapport er udarbejdet i kurserne 02313 Udviklingsmetoder til IT-systemer, 02312 Indledende programmering på første semester af Diplomigneniør IT. Opgaven er del to 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 klasserne *Konto* og *Player* og lave et lille spil. Den nærmere kravspecificering til opgaven findes i "CDIO_opgave_del2.pdf"[Nyb12], kravene bliver behandlet i kapitel 2 på den følgende side. Opgaven bygger på undervisning modtaget i ovenstående kurser hvor der blev benyttet bøgerne [Lar04] og [LL12]. Igennem opgaverne vil klasser blive vist på følgende måde *KlasseNavn*, metoder `metodeNavn()` og design patterns DesignPattern.

Rapporten er sat med LaTeX, UML diagrammer er lavet med Ti*k*Z-UML og andre figurer er lavet med Ti*k*Z og PGF.

Bemærk venligst at Martin Caspersen tidligere var med i gruppe 14 og Jesper Engholm Baltzersen tidligere var med i gruppe 17. Begge skrev størstedelen af rapporterne til CDIO del 1 og derfor kan der forekomme passager i denne rapport som minder om passager fra gruppe 14 og 17s rapporter til CDIO del 1. Kodebasen der er benyttet som udgangspunkt for denne del er således den samme kode som gruppe 14 afleverede i del 1.

1.1 Ansvarfordeling

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

1. Martin Caspersen
2. Jesper Engholm Baltzersen

Ansvarsfordelingen 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.

Alle figurer og diagrammerne er udarbejdet af Martin, men både Martin og Jesper er ansvarlige for indholdet af disse.

Ansvaret for kodningen af de forskellige klasser er beskrevet herunder:

- Die [1, 2]
- MatadorRafleBaeger [1, 2]
- Game [1, 2]
- Player [1, 2]
- BoundaryToGUI [1, 2]
- BoundaryToPlayer [1, 2]

- Main [1, 2]
- Board [1, 2]
- Field og underklasser [1, 2]
- TestDie [1, 2]

2 Krav[1]

2.1 Udgangspunkt

Som udgangspunkt for starten af dette projekt bruges koden som Martin havde skrevet i forbindelse med CDIO del 1 i gruppe 14. Derfor var det kodemæssige udgangspunkt at kodebasen levede op til kravene fra CDIO del 1 som fortolket af gruppe 14.

En del af designet af løsningen til CDIO del 2 er altså at finde ud af hvilke dele af koden til del 1 der kan bruges i del 2. Martin valgte under programmeringen af del 1 af opdele spillet i forskellige klasser som havde forskelligt ansvar i forhold til at implementere spillet i del 1. På grund af den måde spillet var kodet på kunne de fleste af klasserne genbruges med få modifikationer. Selve spillogikken skal sandsynligvis undergå ændringer eftersom spillet i del 2 ikke er det samme som i del 1.

I del 1 havde Martin kodet følgende klasser som havde med selve spillet at gøre:

- `Die`
- `MatadorRafleBaeger`
- `Player`
- `Game`
- `GameController`
- `BoundaryToPlayer`
- `BoundaryToGUI`
- `Main`

Af disse klasser skal især `Game` undergå store ændringer da det blev valgt at starte på ny med spillogikken i del 2. Det er dog mere interessant at overveje om de andre klasser i systemet kan bruges i dette spil. Af speciel interesse er især klasserne `Die`, `MatadorRafleBaeger` og `Player`. Hvis disse klasser ikke afhænger af `Game` for deres funktion vil de sandsynligvis kunne anvendes mere eller mindre uændret i dette spil. Heldigvis blev disse klasser kodet med hensyn til lav kobling (Low Coupling, se afsnit 3.2 på side 9) og ingen af klasserne kender overhovedet til `Game`. Klasserne fra del 1 kan derfor genbruges i dette spil hvor `Game` sandsynligvis vil være ændret. `Player`

indeholder informationer der ikke er nødvendige i dette spil, ændringerne til *Player* vil blive behandlet senere i kapitel 4 på side 10.

Med hensyn til resten af klasserne vil Boundary klasserne i stor stil kunne genbruges. Dette spil kommunikerer nemlig også med den samme udleverede GUI og konsollen.

Udgangspunktet for projektet er nu mere klart og kravene til det nye spil undersøges i det følgende afsnit.

2.2 Kravspecifikation for del 2

Kravene til CDIO del 2 bliver stillet op ved hjælp af UP modellen for en kravspecifikation som præsenteret i kursus 02313. Den udarbejdede kravspecifikation bliver præsenteret i dette afsnit, hvor der er uklarheder i den udleverede tekst er de tolket efter bedste evne. I en virkelig opgave skulle uklarheder selvfølgelig afklares i samarbejde med kunden i Incpetion fasen. Kilde til dette afsnit er [Nyb12], som er opgavebeskrivelsen for CDIO del 2.

2.2.1 Formål med systemet

Formålet med systemet er at der skal implementeres to klasser *Player* og *Konto*, som henholdsvis beskriver en matadorspiller og en spillers kontantbeholdning. Kravene til begge klasser er at de skal indeholde passende `get`, `set` og `toString` metoder. Derudover skal *Konto* indeholde to metoder: `deposit(amount)` og `withdraw(amount)`.

`deposit(amount)` skal tilføje amount til spillerens kontantbeholdning.

`withdraw(amount)` skal fjerne amount fra spillerens kontantbeholdning dog med den lille tilføjelse at kontantbeholdningen ikke må kunne gå i minus og metode skal fortælle om dette.

Ved at bruge disse klasser samt *MatadorRafleBaeger* skal designes et lille spil. I spillet ønskes der benyttet en udleveret GUI som viser status af spillet.

2.2.2 Funktionelle krav

Use case diagram

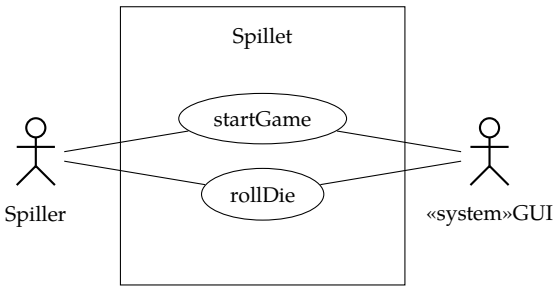
I forbindelse med dette spil er der kun en menneskelig og en system aktør, henholdsvis spilleren og den udleverede GUI. Der er identificeret to use cases, startGame og rollDie som bruges til henholdsvis at starte spillet og for at rulle terningerne i spillet. Begge use cases bruges af spilleren og kommunikere med GUI. Use case diagrammet kan ses i figur 2.1 på næste side.

Use case beskrivelser

startGame

Deltagende aktører: Initieres af spilleren og interagerer med GUI.

Figur 2.1: *Use case diagram.*



1. Spilleren starter spillet.
2. Systemet sætter systemet op til at køre spillet.
3. Systemet giver kontrollen tilbage til spilleren og han kan nu rulle terningerne, se use case rollDie.

rollDie

Deltagende aktører: Initieres af spilleren, interagerer med GUI.

Pre konditioner: Spillet er startet jævnfør use case startGame.
Standard forløb uden vinder:

1. Spilleren har kontrollen og systemet er sat op og klar til at spille, se use case startGame.
2. Spilleren ruller terningerne.
3. Systemet ruller terningerne som repræsenteret i systemet.
4. Systemet kontrollerer hvilket felt spilleren er landet på.
5. Systemet undersøger hvordan spillerens kontantbeholdning påvirkes.
6. Systemet opdaterer spillerens kontantbeholdning.
7. Systemet præsenterer status af spillet for spilleren.
8. Systemet kontrollerer om der er en vinder.

Trin 2 - 8 gentages indtil der bliver fundet en vinder, hver gang trin 2 nås skiftes til den anden spiller.

8a. Der findes en vinder.

1. Systemet præsenterer vinderen af spillet.
2. Systemet lukker spillet ned.

Aktørbeskrivelser

Spilleren Det forventes at spilleren starter spillet og at spilleren ruller med terningerne når systemet anmoder om dette. Derudover præsenterer systemet oplysninger for spilleren for at oplyse om status af spillet.

GUI Det forventes at GUI er i stand til at præsentere de nødvendige oplysninger for at vise spillets status til spilleren.

3 Design patterns[2]

I denne iteration af CDIO-projektet [Nyb12] har en del af udviklernes opmærksomhed været rettet mod "godt design"på kode-niveau. Diskutioner og beslutninger er hovedsageligt foregået/taget på baggrund af første halvdel af GRASP-principperne som de er præsenteret til og med kapitel 24 i [Lar04]. Dette omfatter følgende design patterns/principper:

- High Cohesion
- Low Coupling
- Information Expert
- Creator
- Controller

Hvert af principperne gennemgås yderligere i de følgende afsnit.

Til grund for al Objekt Orienteret (herefter OO) udvikling ligger principperne High Cohesion og Low Coupling. Dette skyldes at vi på denne måde "nemt"får grupperet kodedele der hører logisk sammen, altså OOD (Objekt Orienteret Design). Dette gøres, fordi det resulterer i et mere overskueligt og lettere vedligeholdt system.

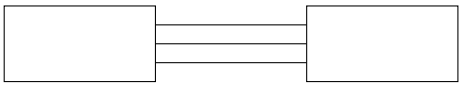
3.1 High Cohesion

Da High Cohesion nærmere er et overordnet princip end et egentligt design pattern er det også anvendt gennem hele systemets design. High Cohesion handler om at samle dele af et program, der logisk har tætte relationer. På denne måde opnår man, at hver klasse kun tager sig af en type operationer. Dermed bliver hele programmets struktur lettere at overskue. I *matador* systemet har hver klasse sin specifikke opgave. `Die` er kun terning og indeholder således kun metoder der hænger tæt sammen med det "at være terning". `MatadorRafleBaeger` indeholder også kun metoder der har med et raflebægers funktion at gøre. Raflebægeret har tætte relationer med terningerne i det, men de er hver deres entitet og er således opdelt i hver deres klasse. Dog har raflebægeret kendskab til terningerne hvilket uddybes mere i afsnit 3.4 på side 10.

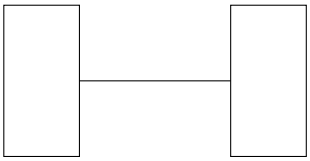
3.2 Low Coupling

I forlængelse af og samspil med High Cohesion følger Low Coupling. Hvor High Cohesion handler om samling af samhørende elementer, siger Low Coupling i modsætning noget om hvordan disse elemtenters interne kendskab er. Altså hvor og hvordan de enkelte elementer "hænger"sammen. Her er fokus bl.a. på hvem der skal have kendskab til hvem. Der er fx ingen grund til at lade *Die* have kendskab til *matadorRafleBaeger*. En terning bruges af andre og der er dermed grundlag for at andre skal kende terningen, men ikke omvendt. Ved at lade disse ideer gennemsyre et helt system opnås Low Coupling nemmere og således bliver systemet også lettere at vedligeholde, da hvert element har mindst mulig indflydelse på andre elementer. Et andet eksempel på Low Coupling er *BoundaryToPlayer*. Denne klasse overholder High Cohesion ved kun at have med præsentation af information i konsollen at gøre. Samtidig overholdes også Low Coupling da den ikke har kendskab til andet end de objekter den henter information ud af. Man kan, med rette, argumentere for, at der brydes med Low Coupling ved, at overføre objekter istedet for kun at overføre den specifikke data *BoundaryToPlayer* skal bruge. Der er taget et valg om at gå fra det ene design til det andet for *matador* systemets vedkommende. Valget skyldes, at det at overføre data gennem flere klasser giver mange redundante metoder gennem systemet. Derfor, hvis et objekt har mere end en attribut der skal overføres til *BoundaryToPlayer* kan der spares en del kode ved, at placere logikken til at hente informationerne ud, i *BoundaryToPlayer*. I figur 3.1 på denne side og figur 3.2 på denne side ses henholdsvis High Coupling, Low Cohesion og Low Coupling, High Cohesion skitseret.

Figur 3.1: *High Coupling og Low Cohesion*



Figur 3.2: *Low Coupling og High Cohesion*



3.3 Information Expert

Dette pattern hænger meget tæt sammen med, og fremkommer af, begrebet "responsibility assignment". Det kan tolkes som "den der ved det, formidler det!". Indehaveren af en given information bliver altså tildelt ansvaret for også at videreformidle den. Et godt eksempel på anvendelse af Information Expert kan findes i *Die*. Her er det oplagt at instanser af *Die* er eksperter på egne *facevalues*. Dermed består "responsibility assignment", for *Die*, i metoder som `getFaceValue` og `rollDie`.

3.4 Creator

Navnet fortæller meget godt hvad dette pattern gør. Nemlig at skabe. Creator pattern anvendes for at finde ud af hvilken klasse der skal skal skabe instanser af andre klasser. Iflg. Creator skal en klasse A, instansiere en anden klasse B, hvis A: omslutter, aggregerer, optager, tæt bruger eller har initialiseringsdata for B [Lar04]. Således er det et meget benyttet pattern simplethen fordi dets brug ligger i kernen af OO udvikling. Et eksempel på en situation hvor dette pattern bør anvendes findes i `matadorRafleBaeger`. Her bør Creator benyttes til instansiering af `Die` fordi netop entiteten raflebæger indeholder og bruger terninger.

3.5 Controller

Brugen af Controller pattern kan udmønte sig i flere forskellige løsninger på samme problem. Dette skyldes at implementationen af Controller hænger tæt sammen med det resterende programs udformning, størrelse og formål. I denne iteration af CDIO-projektet er noget af spillets logik og "styringen"af spillets flow lagt i samme Controller/logik klasse, `Game`. Denne løsning kan fx andvendes, som her, til mindre systemer, hvor en opdeling af klasserne til håndtering af logik og flow synes overflødig pga. systemets relativt lille og overskuelige størrelse. I denne iteration af projektet er det blevet valgt at følge netop denne model. Resultatet er en klasse `Game` der er en smule "bloated"og med en smule mindre samhørighed (afsnit 3.1 på side 8) samt lidt højere kobling (afsnit 3.2 på forrige side) sammenlignet med resten af systemets klasser. Det understreges, at denne model er et særtilfælde og ikke tydeligt understøtter gængse OO-principper. Dens anvendelse kan dog forsvares med systemts relativt lille kompleksitet. Det nævnes at [Lar04] behandler dette nøjagtige eksempel i kap. 17.

De andre muligheder for Controller som præsenteret i [Lar04] er tilfælde for systemet enten opererer i en lukket enhed (tlf. og bank er brugt som eksempler) og den anden, og for dette projekt relevante, tager udgangspunkt i bce/mvc-modellerne. Denne løsningsmodel har til formål at adskille kommunikation og logik. Her ved at lade `Game` blive opdelt i 2 klasser. Heraf vil den ene fungere som en "ren"Controller og den anden som en "ren"entity-klasse. Ved brug af denne model ligner systemets design i højere grad bce-modellen og Controller klassens "ansvarsområde"ville således kunne overføres direkte fra Use Case Modellen.

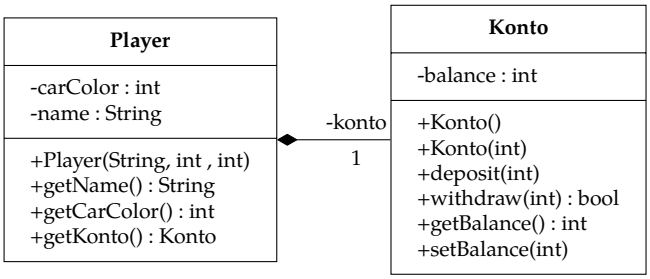
4 Design[1]

Hvor sidste afsnit omhandlede design på et mere teoretisk plan ved gennemgang af design patterns vil dette afsnit afsnit fokuserer på den praktiske udførsel i dette system. Hovedsaligt vil dette ske ved hjælp af diagrammer.

4.1 Diagrammer for `Player` og `Konto`

I figur 4.1 på denne side ses designklassediagrammet for *`Player`* og *`Konto`* som blev designet til CDIO del 2. *`Player`* eksisterede i kodebasen som blev brugt som udgangspunkt for dette projekt, den er dog modificeret hovedsaligt ved at fjerne unødige metoder og variabler. Ændringerne i forhold til udgangspunktet vil kort blive gennemgået:

Figur 4.1: *Designklassediagram der viser `Player` og `Konto` klasserne.*



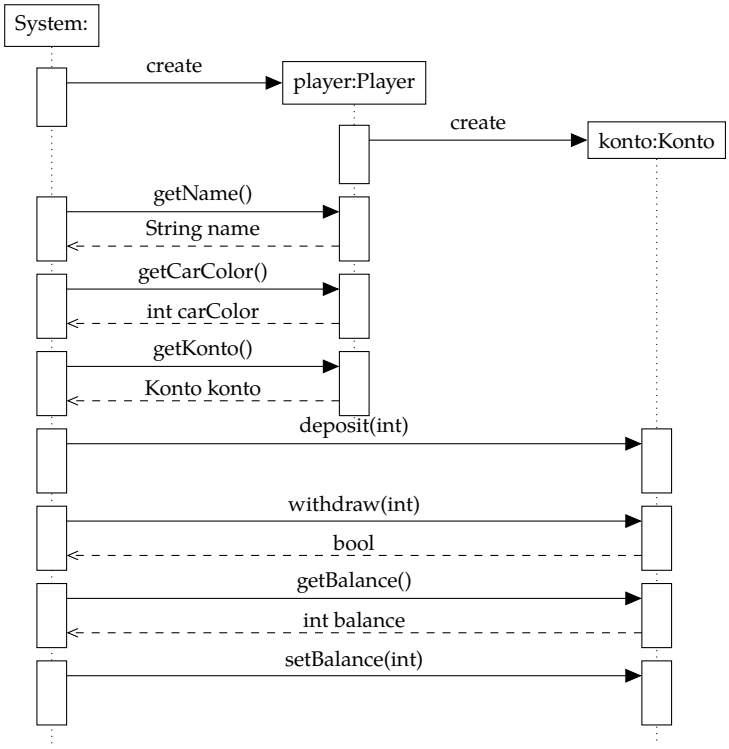
- Fjernet variablerne: point, balance og twelveLastTime.
- Fjernet metoderne: `getPoint`, `getTwelveLastTime`, `setTwelveLastTime`, `addPoint`, `setPoint`, `getBalance` og `setBalance`.
- Tilføjet variabel: konto.
- Tilføjet metode: `getKonto`

Variablerne blev fjernet fordi de ikke skulle bruges i implementeringen af spillet til del 2, balance bliver i stedet repræsenteret af *`Konto`*. *`Player`* indeholder nu en instans af klassen *`Konto`* som har en variabel balance af typen int. Klassen bruges til at repræsentere en spillers kontantbeholdning og har nødvendige metoder til at udføre operationer på denne. To af metoderne er givet fra opgaven af: `deposit(amount)` og `withdraw(amount)`. *`Player`* repræsenter en fysisk spiller som deltager i spillet.

`Player` er ansvarlig for at konstruere sin egen instans af *`Konto`*. Dette er i overenstemmelse med design pattern Creator, se afsnit 3.4 på forrige side, da *`Player`* aggregerer *`Konto`* som også ses af designklassediagrammet.

I figur 4.2 på næste side ses hvordan interaktionen mellem resten af systemet og *`Player`* og *`Konto`* foregår. Det ses af diagrammet at når *`Player`* konstrueres, konstruerer *`Player`* en instans af *`Konto`*. Efter begge klasser er konstrueret af systemet kan alle *`Player`*s public metoder kaldes direkte fra systemet. For at få adgang til hver spillers kontantbeholdning skal *`Konto`* bruges. Denne henter systemet ved at kalde `getKonto()` på den enkelte spiller, denne metode returnerer spillerens instans af *`Konto`*. Når systemet har fået denne instans kan alle public metoder i *`Konto`* kaldes direkte fra systemet. Denne fremgangsmåde er illustreret i figur 4.2 på den følgende side.

Figur 4.2: *Sekvensdiagram der viser `Player` og `Konto` klasserne.*



4.2 Designklassediagrammer for det samlede system

I figur 4.3 på næste side ses designklassediagrammet for det samlede system, for overskuelighedens skyld er undladt `Field` og dens underklasser, som bruges til at repræsentere felter i matadorspillet. Disse klasser er vist i figur 4.4 på side 14 i afsnit 4.3 på næste side hvor de vil blive behandlet nærmere.

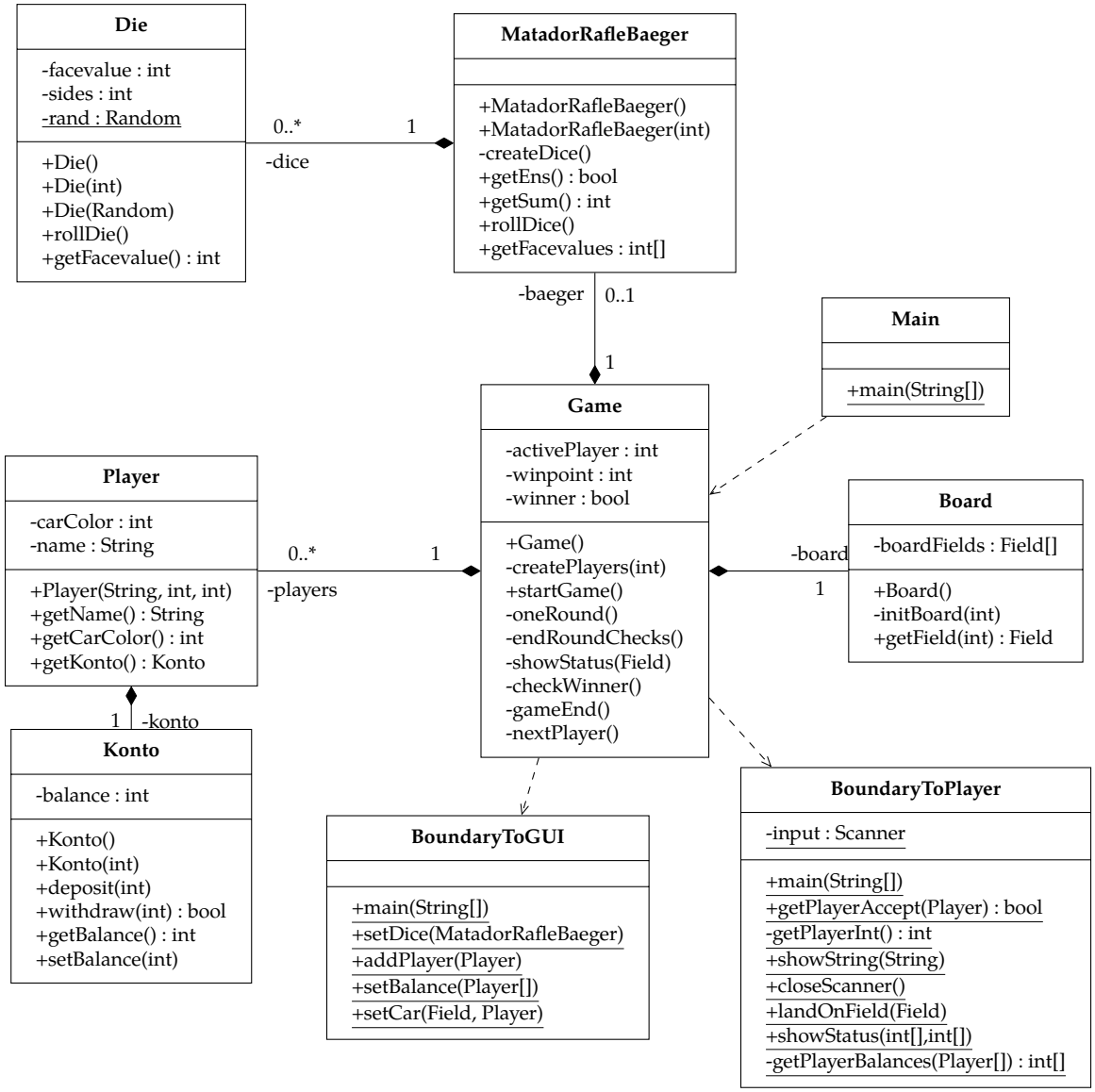
I designklassediagrammet i figur 4.4 på side 14 ses at `Game` aggregerer en instans af `Board` og `MatadorRafleBaeger`, derudover aggregerer `Game` også 0 eller flere instanser af `Player`, for dette spil vil det være 2 instanser af `Player`. `MatadorRafleBaeger` er ansvarlig for at konstruere instanserne af disse klasser fordi den aggregerer dem, i overenstemmelse med design pattern Creator. Hver af disse klasser aggregerer selv andre klasser og de vil kort blive gennemgået.

`MatadorRafleBaeger` aggregerer 0 til flere instanser af `Die` som det ses af diagrammet, for vores spil vil dette være 2 instanser. `Die` repræsenterer en fysisk terning og indholder metoder for at rulle og hente værdien af slaget ud. `MatadorRafleBaeger` repræsenterer et fysisk raflebæger som indeholder et antal terninger, den indeholder metoder til at slå med alle terningerne og hente værdien af terningerne ud. `MatadorRafleBaeger` er ansvarlig for at konstruere instanser af `Die` i overenstemmelse med design pattern Creator da `MatadorRafleBaeger` aggregerer `Die`.

Hver instans af `Player` repræsenterer en fysisk spiller og aggregerer hver en instans af `Konto` som repræsenterer en spillers kontantbeholdning. Disse klasser blev nærmere beskrevet i afsnit 4.1 på foregående side.

`Board` repræsenterer matadors spilleplade og aggregerer en række felter, disse omtales i afsnit 4.3 på den følgende side.

Figur 4.3: *Designklassediagram der viser det meste af systemet. `Field` og dennes underklasser vises i figur figur 4.4 på næste side.*



Udover de ovennævnte klasser indeholder systemet også `Main`, `BoundaryToGUI` og `BoundaryToPlayer`. `Main` bruges til at starte spillet. `BoundaryToGUI` er ansvarlig for al kommunikationen mellem systemet og den udleverede GUI. `BoundaryToPlayer` er ansvarlig for kommunikationen mellem systemet og den fysiske spiller.

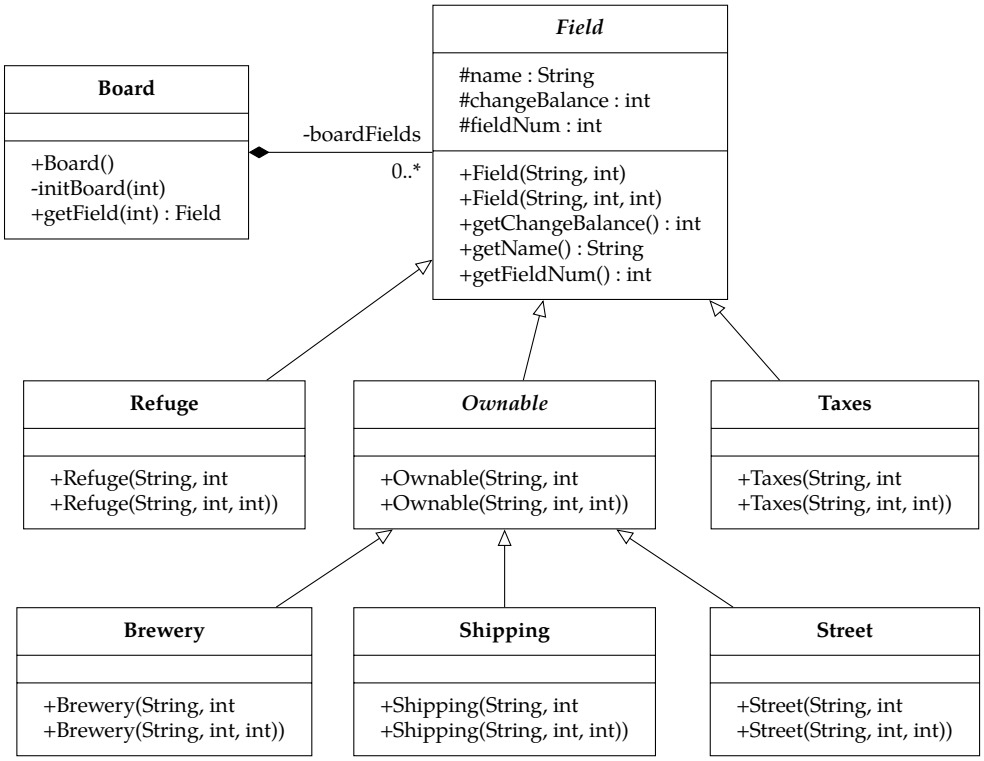
Det bemærkes at de dependencies der stammer fra at boundaries modtager objekter er undladt. Boundaries afhænger derfor af de objekter de modtager som parametre, dependencies er undladt for ikke at forstyrre diagrammet unødigt.

4.3 Designklassediagram for `Field` og underklasser

Som beskrevet i afsnit 4.2 på foregående side er designklassediagrammet for det samlede systemet delt i to. Dette afsnit beskriver designklassediagrammet for `Board`, `Field` og `Field`s underklasser, diagrammet ses i figur 4.4 på den følgende side. Repræsenta-

tionen af felterne i systemet er delt op i forskellige klasser som hver især repræsenterer en type af felter som findes i et matadorspil.

Figur 4.4: *Designklassediagram der viser `Board`, `Field` og underklasser af `Field`.*



`Board` aggregerer 0 eller flere instanser af `Field`s underklasser. Ved konstruktionen af `Board` er `Board` ansvarlig for at konstruerer alle instanserne som nødvendigt. Dette i overenstemmelse med design pattern CREATOR, da `Board` aggregerer instanserne.

`Field` er klassen som alle felter arver fra, i denne version af spillet indeholder `Field` alle variabler og metoder som bruges. Det vil sige at alle metoderne som bliver brugt til del 2s felter er implementeret i `Field`. `Field` er abstract da det ikke giver mening at have en instans af `Field` eftersom der er underklasser til at repræsenterer alle de typer af felter der findes i et matadorspil.

Af direkte underklasser til `Field` er der `Refuge`, `Ownable` og `Taxes`, hvor `Ownable` er abstract mens de to andre er normale klasser. `Refuge` og `Taxes` repræsenterer to typer af felter i matadorspillet og under starten af spillet vil der blive konstrueret instanser af disse klasser som nødvendigt.

`Ownable` er som tidligere nævnt en abstract klasse ligesom `Field` da den ikke repræsenterer én type af matadorfelter. I stedet repræsenterer den fællestræk ved flere typer af matadorfelter, nemlig alle de felter som kan ejes af en spiller. `Ownable` har tre underklasser, `Brewery`, `Shipping` og `Street` som alle tre har det til fælles at de kan ejes af en matadorspiller. Der vil ved starten af spillet blive konstrueret instanser af disse klasser som nødvendigt.

Som tidligere nævnt er alle metoder og variabler implementeret i `Field` og ingen af underklasserne overskriver implementeringen. Derfor ville der isoleret set i denne del af CDIO projektet ikke være noget godt argument for at lave denne opbygning. Under-

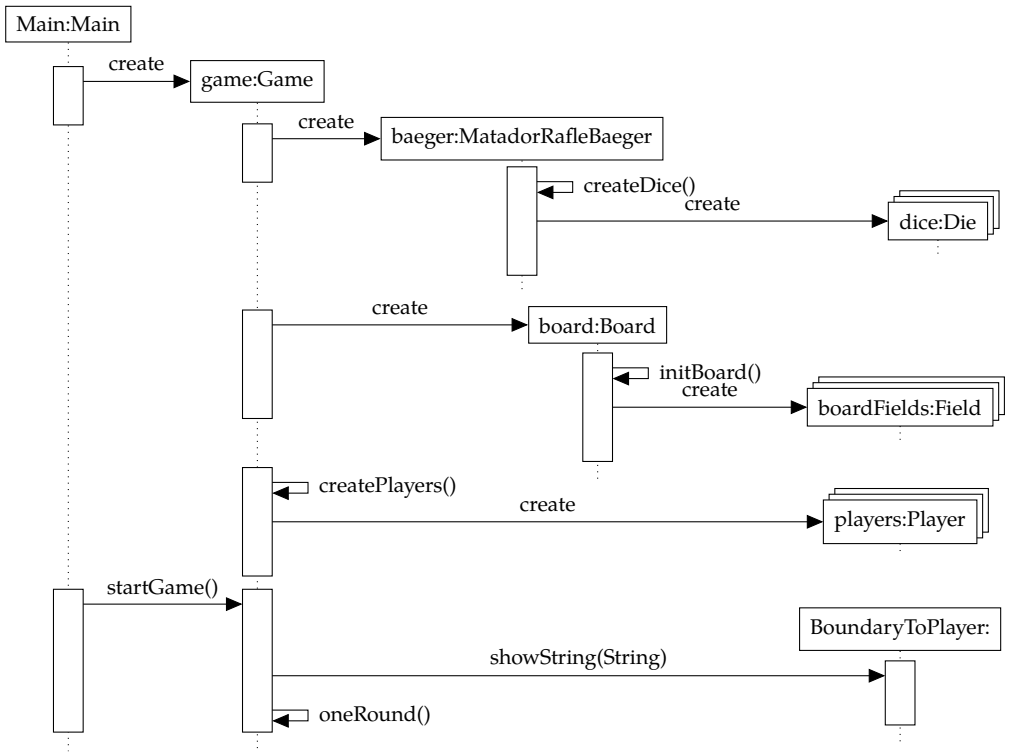
klasserne er alle implementeret sådan at de bare kalder deres superklasses konstruktør med `super()`.

Denne opbygning er da også hovedsaligt valgt for at gøre systemet klar til at implementere mere avancerede versioner af et matadorspil. Det er nemlig ikke det samme der sker når man lander på de forskellige typer af matadorfelter ifølge de rigtige matadorregler. Derfor skal de forskellige typer af felter have forskellige implementering af nogle af de metoder der kaldes når man lander på felterne. Alligevel har alle felterne det til fælles at man kan sige at de er et *Field*. Det er netop et af kravene for at man bruger arv, samtidig har felterne også andre fælles træk. Alle typer af felter har et navn og et fieldNum som repræsenterer deres id i den udleverede GUI.

4.4 Sekevensdiagrammer for det samlede system

I forbindelse med at opsætte sekvensdiagrammerne er det svært at præsenterer hele systemet i et samlet sekvensdiagram. Derfor er sekvensdiagrammet over forløbet af et spil delt op i to forskellige diagrammer. Et der viser hvordan spillet startes op og et der viser forløbet af en runde af spillet, disse kan ses i henholdsvis figur 4.5 på denne side og figur 4.6 på side 18.

Figur 4.5: *Sekvensdiagram som viser opstart af spillet.*



I figur 4.5 på denne side ses sekvensdiagrammet for opstart af spillet, dette forløb skal køres inden der kan spilles en runde. Forløbet af sekvensdiagrammet vil herunder blive gennemgået på punktform:

1. *Main* konstruerer en instans af *Game*.

2. *Game* konstruerer en instans af *MatadorRafleBaeger* som konstruerer to instanser af *Die*.
3. *Game* konstruerer en instans af *Board* som konstruerer de 11 instanser af underklasser til *Field*.
4. *Game* konstruerer to instanser af *Player*, denne konstruktion sker i hjælpemetoden `createPlayers()`.
5. *Main* kalder metoden `startGame()` i *Game*.
6. *Game* printer en besked til spilleren og kalder `oneRound()` i sig selv.

Efter disse sekvenser er gennemløbet er spillet klar til at der kan spilles en runde. Dette vil blive gennemgået herunder.

I figur figur 4.6 på side 18 ses sekvensdiagrammet for forløbet af en runde af spillet og spillet er som gennemgået ovenfor klar til at der kan spilles en runde. Forløbet af sekvensdiagrammet bliver gennemgået herunder:

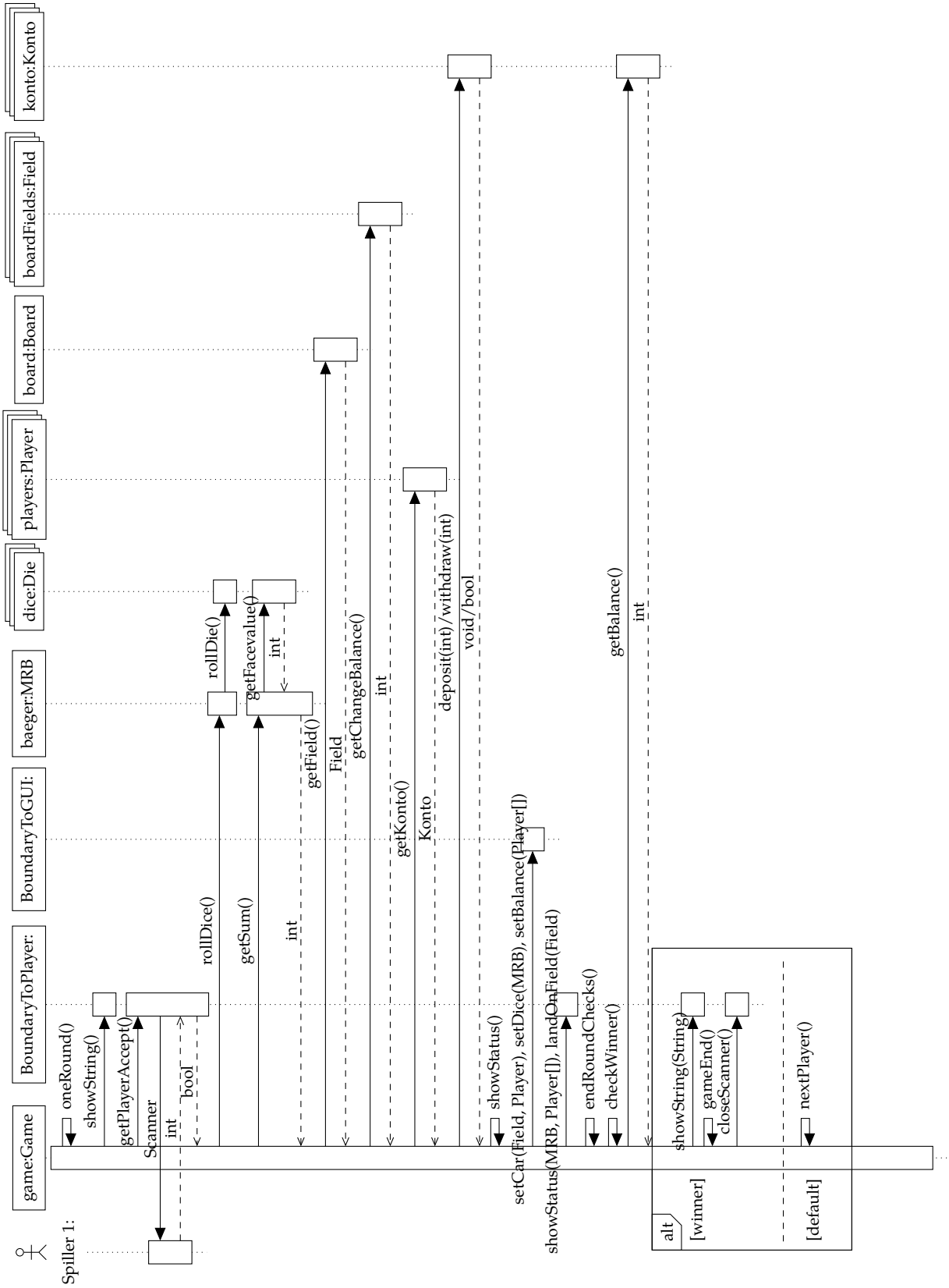
1. *Game* printer en besked til spilleren og spilleren anmodes via *BoundaryToPlayer* om at indtaste en integer for at slå med terningerne.
2. *Game* ruller med terningerne *Die* ved hjælp af *MatadorRafleBaeger*.
3. *Game* henter værdierne af terningerne *Die* ved hjælp af *MatadorRafleBaeger*.
4. På baggrund af det slåede henter *Game* et *Field* ved hjælp af *Board*.
5. *Game* kalder `getChangeBalance()` på det hentede *Field*, dette returnerer en integer som beskriver hvilken ændring der skal ske med spillerens kontantbeholdning.
6. *Game* henter en *Konto* fra den aktive *Player*.
7. *Game* kalder enten `deposit(int)` eller `withdraw(int)` på den hentede *Konto*.
    a) Hvis `withdraw(int)` kaldes og der returneres false har den anden spiller vundet.
8. *Game* kalder `showStatus()` på sig selv.
    a) *Game* kalder *BoundaryToGUI* med metoderne `setCar(Field, Player)`, `setDice(MRB)` og `setBalance(Player[])`.
    b) *Game* kalder *BoundaryToPlayer* med metoderne `showStatus(MRB, Player[])` og `landOnField(Field)`.
9. *Game* kalder endRoundChecks() på sig selv.
    a) *Game* kalder checkWinner() på sig selv.
        i. *Game* kalder `getBalance()` på *Konto*.

a) Hvis der er en vinder:
   i. *Game* kalder `showString(String)` på *BoundaryToPlayer* for at printe en besked til spilleren.
   ii. *Game* slutter spillet ved at kalde `gameEnd()` på sig selv som kalder `closeScanner()` på *BoundaryToPlayer*

b) Hvis der ikke er en vinder:
   i. *Game* kalder `nextPlayer()` på sig selv, som avancerer turen til næste spiller.

Efter forløbet af en runde vil hele sekvensdiagrammet for en runde blive gennemløbet igen nu med den anden spiller. Dette gentages indtil der er fundet en vinder.

Figur 4.6: *Sekvensdiagram som viser en runde af spillet, MRB betyder MatadorRafleBaeger.*



5 Implementering[1]

For præsentation af implementeringen af spillet i kode er valgt nogle interessante elementer af koden ud. Disse elementer vi blive behandlet i dette afsnit. Al koden til spillet kan findes i det vedlagte Eclipse projekt. I forbindelse med implementeringen af spillet har vi valgt nogle områder af vores kode ud som vi finder specielt interessant. Disse områder vil blive behandlet i dette afsnit. Al koden til spillet kan ses i det vedlagte Eclipse projekt.

For at køre spillet køres Main.java som starter spillet.

5.1 Objekter sendes til boundaries

BoundaryToPlayer og *BoundaryToGUI* er boundaries til henholdsvis den fysiske spiller og GUI. Boundaries er ansvarlige for at præsentere data fra spillet og det er praktisk at have al koden der har med denne præsentation at gøre i selve boundaries. På denne måde kan præsentationen af dataene ændres ved blot at ændre i boundary klasserne. Samtidig kan spillet tilpasses andre systemer, f.eks. en anden GUI, ved blot at ændre i den tilsvarende boundary.

Dette er realiseret ved at objekter sendes direkte til boundaries, boundaries henter på denne måde de nødvendige data ud ved at kalde get metoderne på objekterne. På denne måde holdes al koden i spillet der har med præsentation af data overfor GUI og den fysiske spiller via konsollen i boundaries.

Eksempel på en metode i *BoundaryToPlayer* som tager imod et objekt som parameter er `landOnField(Field)`, koden for metoden ses i figur 5.1 på denne side. Denne metode tager imod et objekt af typen *Field*, der er her tale om det felt der netop er landet på. På feltet kaldes der `getName()` og `getChangeBalance()` for at få henholdsvis navnet fra feltet og ændringen til spillerens konto. `landOnField(Field)` bruger oplysninger til at printe information om feltet til den fysiske spiller i konsollen.

Figur 5.1: *Eksempel på metoden `landOnField(Field)` i `BoundaryToPlayer`*

```
1  public static void landOnField(Field field) {
2    String fieldName = field.getName();
3    String result = "Det␣";
4
5    showString("Du␣landede␣paa:␣" + fieldName + ".");
6
7    result += (field.getChangeBalance() >= 0) ? "giver␣" + ↩
       ↪ field.getChangeBalance(): "koster␣" + ↩
       ↪ (-field.getChangeBalance());
8    result += ".";
9    showString(result);
10 }
```

5.2 Exception handling

I implementeringen af spillet benyttes et *Scanner* objekt konstrueret med parameteren System.in til at hente indtastninger fra den fysiske spiller via konsollen. På dette objekt benyttes metoden `nextInt()` for at hente en integer fra den fysiske spiller. At benytte `nextInt()` har nogle fordele og ulemper, en fordel er at man får en int med det samme uden at skulle omdanne en String til int. En af ulemperne er at `nextInt()` thrower en exception hvis der tastes andet end en int ind. Den exception der throwes er af typen *InputMismatchException*, exceptionen betyder at den token der blev modtaget af scanner ikke er i den range der blev angivet[Ora12]. Denne exception fremkommer for eksempel hvis der indtastes en String.

Det at der kommer en exception er ikke i sig selv dårligt, hvis der ikke kom en exception ville det være umuligt for Java at afgøre hvad der skulle ske eftersom koden antager at den modtager en int. Problemet er at spillet stopper når det møder en exception der ikke bliver håndteret. For at undgå dette er der indført exception håndtering i koden der er ansvarlig for at modtage en int fra den fysiske spiller.

I figur 5.2 på denne side ses `getPlayerInt()`, det ses at metoden `nextInt()` kaldes på input, input er en *Scanner*, inde i en try blok. Koden i try blokken forsøges kørt hvis der kommer en exception på grund af koden i try blokken, vil den blive fanget af catch (Exception e). Her er den sat op til at fange alle exceptions, normal praksis er at man sættet den op til at fange specifikke exceptions, på den måde kan man håndtere forskellige typer exceptions på den forskellige måder. For at forenkle koden lidt er her valgt at fange alle exceptions. Hvis en exception bliver fanget printes en besked og man får mulighed for at forsøge at taste en int ind endnu en gang.

Figur 5.2: *Exception handling i `getPlayerInt()` i `BoundaryToPlayer`*

```
1  private static int getPlayerInt() {
2    int inputInt;
3
4    try  {
5      inputInt = input.nextInt();
6    } catch (Exception e) {
7      showString("Kun integers er tilladt.");
8      input.nextLine();
9      inputInt = getPlayerInt();
10   }
11   return inputInt;
12 }
```

6 Test[1]

I forbindelse med CDIO del 2 er det meste af testningen foregået ved at spillet er blevet kørt. Mens spillet kørte er det kontrolleret at spillet udfører de rigtige handlinger i forhold til de værdier der bliver slået med terningerne.

Det er valgt at lave en unittest for en klasse ai systemet for at få lidt erfaring med udførelsen af unittests.

6.1 Unittest af `Die`[2]

Til unit test er JUnit anvendt som plugin til Eclipse. JUnit giver en række muligheder for at teste hvordan et program opfører sig. Til at lave mock's benyttes mockito som er et mocking framework. Til tests af *Die* er anvendt *assertions*. Alle tests ligger i *TestDie*. Der er foretaget 2 typer af tests. I den første verificeres det, med *assertTrue*, først at terningens værdi ligger i intervallet 1-6 (figur 6.1 på denne side) og efterfølgende at terningen også kan oprettes med 200 sider.

Figur 6.1: *Test terning facevalue har interval 1-6*

```
1 @Test
2 public void testDie() {
3   die1 = new Die();
4   assertTrue(die1.getFacevalue() >=1 && die1.getFacevalue() ↩
      ↪ <= 6);
5 }
```

Den anden del af testen illustrerer en anden måde at opsætte testen på. Her laves et mock objekt af *Random* fra Java's standard API. Dette mock objekt bliver kodet til at returnere 3 når metoden `nextInt()` kaldes med parameteren 6. Efterfølgende injiceres det mock'ede objekt i en instansiering af *Die*. Det er således nu muligt, at bestemme udfaldet af et kald til *Random*. Dermed kan det testes at den *Die* rent faktisk benytter *Random* til at bestemme værdien af *facevalue*. Yderliere bruges *verify* til at sikre at der har fundet et kald af `Random.nextInt(6)` sted. Hele denne del af testen kan ses på figur 6.2 på denne side.

Figur 6.2: *Test `rollDie()`*

```
1  @Test
2  public void testRollDie() {
3    Random mockRand = mock(Random.class);
4    when(mockRand.nextInt(6)).thenReturn(3);
5    Die die = new Die(mockRand);
6
7    die.rollDie();
8
9    assertTrue(die.getFacevalue() == 3+1);
10   verify(mockRand).nextInt(6);
11 }
```

7 Konklusion[1, 2]

Formålet med CDIO del 2 var at implementere klasserne *Player* og *Konto* og to krævede metoder. Med disse klasser samt den tidligere *MatadorRafleBaeger*, ønskedes udviklet et lille spil. Systemet skal desuden dokumenteres med UML designklassediagrammer og designsekvensdiagrammer.

For at udvide systemet med den nye funktionalitet ses i kapitel 2 på side 5, på først hvilket udgangspunkt der er fra det tidligere projekt, dernæst hvilke krav der er til dette projekt. De fleste af klasserne fra det tidligere projekt kunne genbruges i dette projekt, hvilket har mindsket arbejdet med implementeringen. Dette har givet mulig for at forberede systemet til senere iterationer.

Designet af systemet blev gennemgået ved hjælp af designklassediagrammer og designsekvensdiagrammer.

Koden i Eclipse indeholder nu et spil der lever op til kravene stillet i opgavebeskrivelsen [Nyb12] med alle udvidelsesmulighederne tilføjet. Derudover er udviklet en unittest for en enkelt klasse for at få erfaring med udviklingen af unittests.

Referencer

[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 Nybord. *Del opgave 2. Anvendelse af klasser og relationer*. Vers. 2012-08-08. Opgavebeskrivelse udleveret i forbindelse med projektet. DTU, 8. aug. 2012.

[Ora12] Oracle. *Class InputMismatchException*. 8. nov. 2012. URL: `http://docs.oracle.com/javase/7/docs/api/java/util/InputMismatchException.html` (sidst set 08.11.2012).