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

Projekt navn: del3
Gruppe nr: 21.
Afleveringsfrist: Mandag den 03/12-2012 kl. 05:00
Denne rapport er afleveret via Campusnet (der skrives ikke under)
Denne rapport indeholder 23 sider incl. denne side.
Denne rapport indeholder 15 sider excl. forside, indholdsfortegnelse og bilag.

Studie nr, Efternavn, Fornavne

s122996, Caspersen, Martin
Kontakt person (Projektleder)

s100182, Baltzersen, Jesper Engholm

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 tre af tre CDIO opgaver på første semester, det overordnede formål med opgaverne på semesteret er at lave et Matador spil. Denne tredje opgave fokuserer på at implementere et klassehieraki over felterne i et matadorspil og modificere spillet fra del 2. Den nærmere kravspecificering til opgaven findes i "CDIO_opgave_del3.pdf"[Nyb12], kravene bliver behandlet i kapitel 2 på denne 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 del 2 er den kode som Gruppe 14 afleverede til del 1.

Det antages i opgaven at læseren har kendskab til UML 2.0 syntaks, GRASP.

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.

Begge gruppemedlemmer er ansvarlige for alle dele af koden.

2 Krav[2]

Kravene til CDIO-opgave del 3 bliver behandlet gennem UP-modellen for en kravspecifikation som præsenteret i kursus 02313. Hvor der er uklarheder i den udleverede tekst er de tolket efter bedste evne. Uklarheder i beskrivelsen bør naturligvis afklares med kunden i et virkeligt projekt. Kilde til dette afsnit er [Nyb12], som er opgavebeskrivelsen for CDIO-opgave del 3.

2.1 Formål

Kendskab til arv og polymorfisme skal demonstreres gennem implementering af fields i et matadorspil. Yderligere skal det demonstreres at gruppen kan anvende arrays i forbindelse med instanser af egne typer. Derudover skal det illustreres at der kan arbejdes på et mere abstrakt niveau gennem udarbejdelse af en domænemodel over hele spillet.

2.2 Specificering af krav

Udfra den udleverede opgavebeskrivelse [Nyb12] er der udarbejdet en længere liste med specificering af kravene. De områder der fremgår klare og utvetydige i opgavebeskrivelsen er blot overført, mens andre dele er uddybet og afklaret. Da gruppen forsøger at arbejde efter UP modellen og denne er Use Case driven, er de dele af kravene der kunne overføres til Use Case's indført i afsnit 2.3.1 på næste side og afsnit 2.3.1 på den følgende side. De krav der ikke kunne overføres til Use Case's er placeret i appendiks A på side 18 som er UP's artefakt til krav der ikke kan placeres i Use Case's.

2.3 Funktionelle krav

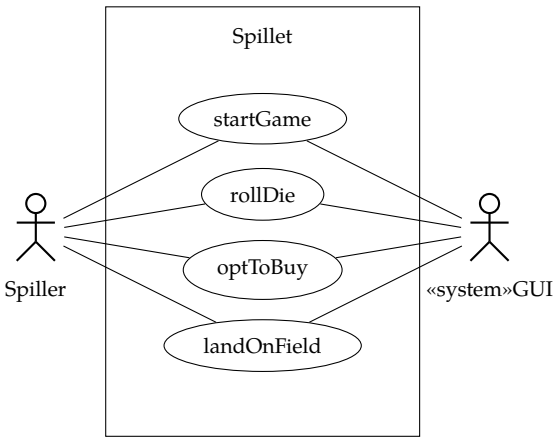
2.3.1 Use Case Diagram

Til udarbejdelsen af et Use Case Diagram er identificeret 4 Use Cases.

- startGame
- rollDie
- optToBuy
- landOnField

Spillet interagerer med to aktører, spilleren og den udleverede GUI. Således fremgår begge disse af Use Case Diagrammet i figur 2.1 på denne side.

Figur 2.1: *Usecasediagram for spillet.*



Use Case beskrivelser

Der henvises til CDIO-opgave del 2 for de to Use Case's: *startGame* og *rollDie*. I afsnit 2.3.1 på denne side behandles Use Case UC1 - optToBuy og UC2 - landOnField i afsnit 2.3.1 på denne side.

Use Case UC1 : optToBuy

Primær aktør: Spiller

Beskrivelse af aktør: Spiller, en person der interagerer med systemet gennem konsol dialog.

Ansvar: Skal handle ud fra om han vil købe det field han er landet på. Dette gøres via konsol dialog.

Prekonditioner: Spilleren er landet på et felt der ikke ejes af nogen af de andre spillere og spilleren har penge nok på sin konto til at kunne betale for feltet.

Postkondition/Succeskriterie: Spilleren foretager positivt valg gennem konsol dialog. Hvis han ønsker at købe feltet registreres dette som ejet af spilleren og penge trækkes fra hans konto. Turen overgår til næste spiller.

Main Success Scenario:

1. Konsollen spørger spiller om han ønsker at købe feltet han er landet på. Konsollen oplyser om mulige svar.
2. Spilleren svarer ved at indtaste positivt svar i konsollen.
3. Systemet trækker et beløb svarende til købsprisen fra spillerens konto.
4. Systemet registrerer at feltet nu ejes af den pågældende spiller.
5. Konsollen oplyser spilleren om at han har købt feltet.
6. Konsollen oplyser om at det er næste spillers tur

2a Spiller ønsker ikke at købe felt.

1. Spilleren indtaster negativt svar
2. Konsollen oplyser om at det er næste spillers tur

Use Case UC2 : landOnField

Primær aktør: System

Beskrivelse af aktør: System foretager opgaver i forbindelse med at spiller lander på et felt.

Ansvar: Skal anvende korrekt algoritme i forhold til det felt der er landet på.

Prekonditioner: Spilleren er landet på et felt.

Postkondition/Succeskriterie: System har enten foretaget en transaktion eller intet gjort afhængig af hvad reglerne foreskriver for det enkelte felt.

Main Success Scenario:

1. Spiller lander på felttypen Street, Shipping eller Brewery.
2. System finder ud af at spiller ikke ejer feltet.
3. System finder ud af at en anden spiller ejer feltet.
4. System trækker leje fra spillers konto og overfører til ejers konto.
5. System giver turen videre til næste spiller

Alternative forløb:

1a Spiller lander på felttypen Refuge.

1. System overfører bonus til spillers konto
2. Tilbage til MSS: 5

1b Spiller lander på felttypen Tax.

1. System trækker tax fra spillers konto
2. Tilbage til MSS: 5

2a System finder ud af at spiller ejer feltet.

1. System foretager ingen transaktioner
2. Tilbage til MSS: 5

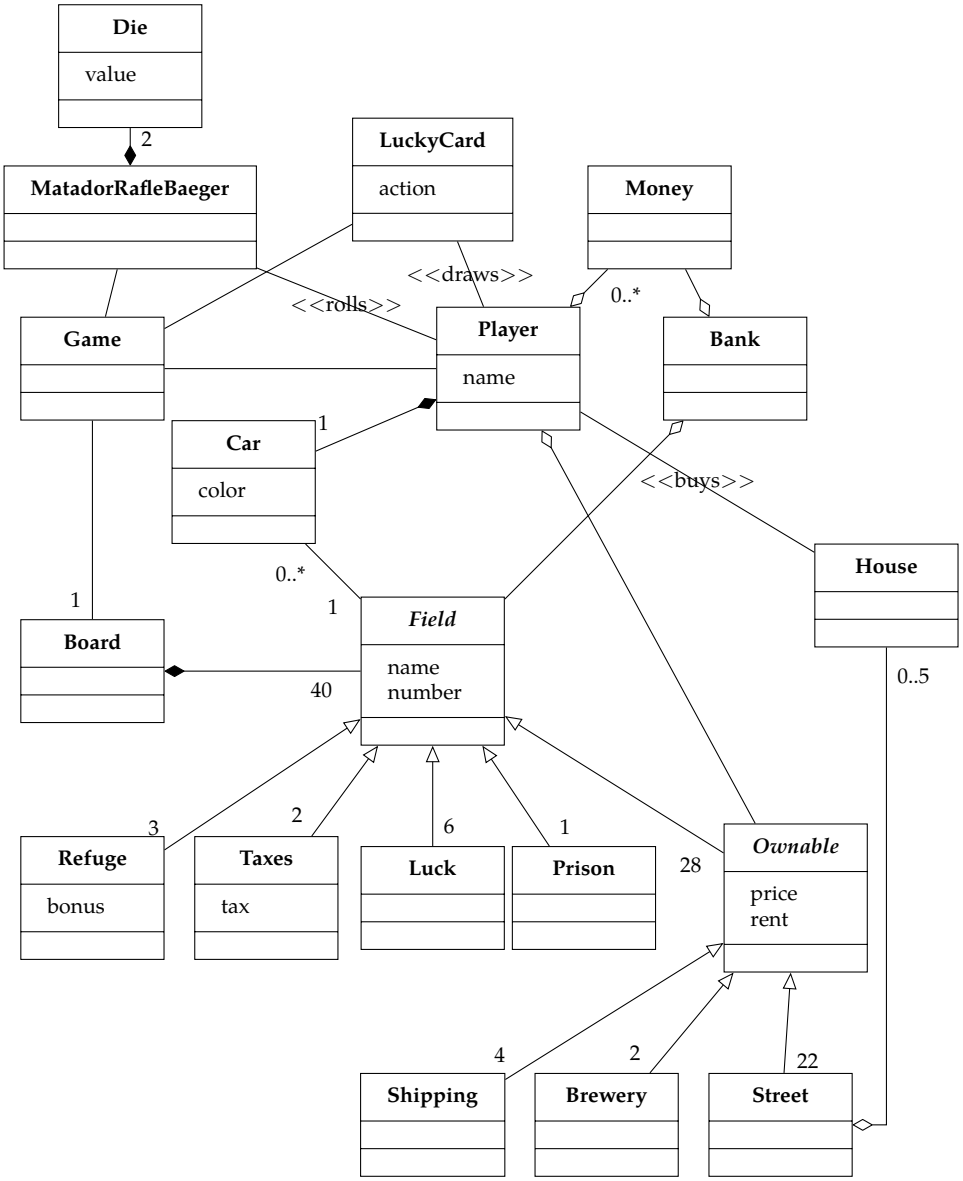
3a System finder ud af at ingen ejer feltet.

1. System foretager ingen transaktioner
2. Tilbage til MSS: 5

4a Spiller har ikke penge nok på konto til at betale leje.

1. System sætter spiller som loser
2. Tilbage til MSS: 5

Figur 3.1: *Domæneklassediagram over matadorspillet.*



3 Design

3.1 Domænemodel[2]

3.1.1 Overblik

Domænemodellen i figur 3.1 på denne side består af en række kasser repræsenterende matadorspillets forskellige elementer (herefter klasser). Dvs. alle de enkeltgrupperinger gruppen fandt nødvendige til at kunne beskrive et helt matadorspil. I udarbejdelsen af domænemodellen er der trukket meget på den objektorienterede viden opnået gennem kurserne 02312 og 02313. Herunder særligt den anvendte UML syntaks til illustration af multipliciteter, associationer og generelt indbyrdes forhold klasserne imellem. Hver klasses multiplicitet, dvs. antal af forekomster, er angivet med et lille tal uden for klassens navn. Alle klasser har forbindelser til andre klasser. Måden disse forbindelser

er tegnet på varierer efter hvordan klassernes indbyrdes forhold er. De steder hvor der står et ord på tværs af en forbindelse angiver dette ord en specificering af denne forbindelses type. Der anvendes en række forskellige syntakser i domænemodellen, hvoraf en af hver vil blive beskrevet i det følgende, samt en mere detaljeret gennemgang af modellen.

3.1.2 Modelforklaring

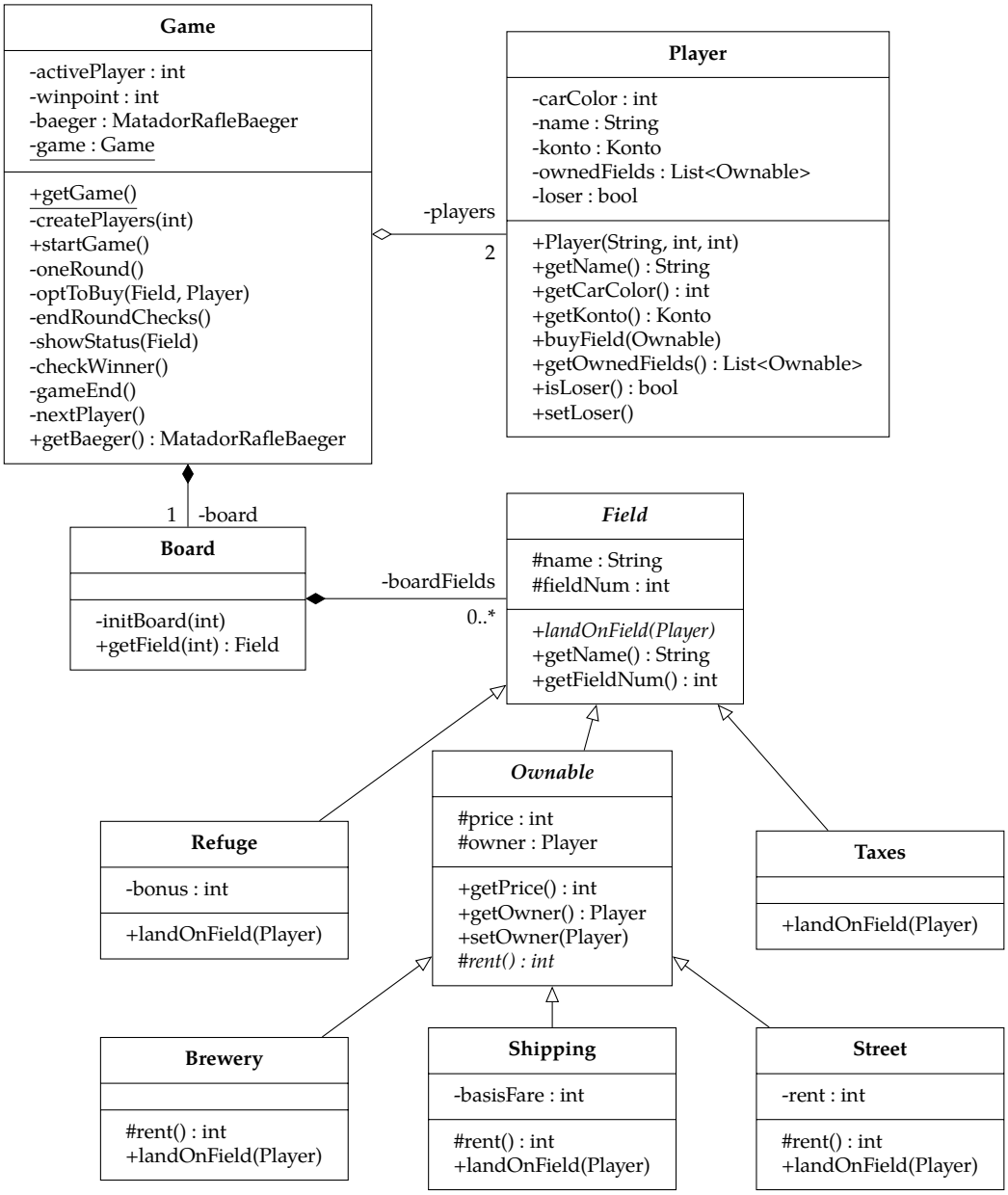
Der er 3 centrale elementer i matadorspillet. Disse er den enkelte spiller `Player`, klassen `Game` eller spillet som helhed og brættet som udgør den fysiske dimension af spillet. `Game` skal forstås som en slags abstraktion over regelsættet i matador. Spillet spilles med 2 terninger `Die` som `Player` ruller med `MatadorRafleBaeger`. Det ses at der er 2 terninger i spillet ved det lille 2 tal under klassen `Die`. Med andre ord dikterer `Game` at `Player` ruller `MatadorRafleBaeger` med 2 terninger i. Derfor er der indbyrdes forbindelser og «rolls» på forbindelsen fra `Player` til `MatadorRafleBaeger`.
Hvis der tages udgangspunkt i 1 `Player` ses det at han har 1 `Car` som har en farve *color*. Denne *color* står inden i kassen der angiver `Car` og er således en attribut. Altså en egenskab som `Car` har. Det ses også at `Player` har `Money` og multipliciteten "0..*"betyder at spilleren kan have fra 0 til uendeligt mange penge. Der er selvfølgeligt nogle fysiske begrænsninger på dette og * kunne også skiftes ud med den sum man skal have for at vinde spillet. `Player` har også forbindelse til `LuckyCard` hvorpå der står «draw». `Player` trækker altså et `LuckyCard`. Dette sker når der landes på feltet luck. Der er taget beslutning om ikke at tilføje en forbindelse fra `LuckyCard` til `Luck` af hensyn til overskueligheden, men denne kunne meget vel også tilføjes. `Player` har også en forbindlese til `Game`. Denne skyldes det faktum at `Game` er den regeldefinerende klasser og således bestemmer restriktioner for alt i spillet. Således kunne man med rette argumentere for at `Game` skulle have forbindelse til alle dele af spillet, Det er dog valgt kun at medtage de vigtigste forbindelser. Ydermere har `Player` forbindelse til `House` med «buys» på. Dette indikerer at `Player` kan købe `House`. Den videre forbindelse fra `House` til `Street` viser at disse hører til på en `Street`. Under `Game` er en forbindelse til `Board`. Dette er selve spillepladen. Der er kun 1 spilleplade per spil og denne består endvidere af 40 `Fields`. Hvert `Field` har et *name* og *number*. Der kan stå flere biler på samme `Field`, men hver bil kan kun stå på et `Field` ad gangen. Dette ses ved multipliciteterne på forbindelsen mellem de 2 klasser. De 40 `Field`'s som spillepladen består af er som nævnt inddelt i underklasser. Hver underklasse har en speciel funktion. Multipliciteterne angiver hvor mange der er af hvert felt. Underklassen `Ownable` er en abstraktion over de `Field`'s der har den fælles egenskab at de kan ejes. Udover at have *name* og *number* attributterne som tidligere angivet har de også attributterne *price* og *rent*. I standard versionen af et matadorspil findes der også et skøde til hver `Ownable`. Der er taget beslutning om at udelade dette i modellen da skøder ikke er nødvendige for at illustrere spillet, men de kunne naturligvis indføres.

3.2 Designklassediagram[2]

3.2.1 Generelt

figur 3.2 på denne side viser et udsnit af det totale spil. Det er således kun de, for denne 3. del af CDIO projektet særligt interessante, klasser der er afbildet. For designklassediagram og beskrivelse for de resterende dele henvises til [2112]. I det følgende vil overordnede designbeslutninger beroende på gruppens viden om designpatterns og principper blive behandlet.

Figur 3.2: *Designklassediagram over ændrede klasser i matadorspillet.*



3.2.2 Arv og polymorfi

I denne del af pojektet er det særligt interssant at kigge på forholdet mellem *Field* og de klasser der arver fra denne.*Field* er en abstrakt klasse og det ses at klasserne *Refuge*, *Taxes* og *Ownable* arver fra *Field*. *Ownable* er også en abstrakt klasse som så igen har underklasserne *Brewery*, *Shipping* og *Street*. Alle underklasserne i dette arvehieraki har hver deres polymofiske implementation af de, i de abstrakte klasser definerede, abstrakte metoder.

3.2.3 Singleton

I spillet er der kun behov for en instans af klassen *Game* og *Brewery* har behov for at kunne hente værdien af terningerne, til udregning af leje, er der taget beslutning om at anvende SINGLETON ved oprettelse af *Game*. Dette ses også i figur 3.2 på foregående side ved at angivelserne af atributten *game* og `getGame` begge er markeret som statiske. På denne måde kan *Brewery* hente summen af terningerne ud gennem den statiske adgang til *Game*.

3.3 Sekvensdiagrammer[1]

For at vise hvordan spillet fungerer er der produceret to sekvensdiagrammer. Disse to sekvensdiagrammer viser de dele af systemet som har undergået betydningsfulde ændringer siden del 2. Der er tale om implementeringen af metoderne `optToBuy(Field, Player)` og `landOnField(Player)` i henholdsvis *Game* og *Field*s underklasser.

`optToBuy(Field, Player)` kaldes hver gang en spiller lander på et felt, sekvensdiagrammet ses i figur 3.3 på den følgende side. Metoden undersøger ved hjælp af operatøren instanceof, beskrevet i afsnit 4.4 på side 14, om feltet kan købes. Derefter undersøges ved et kald til `getOwner()` om der allerede er en ejer.

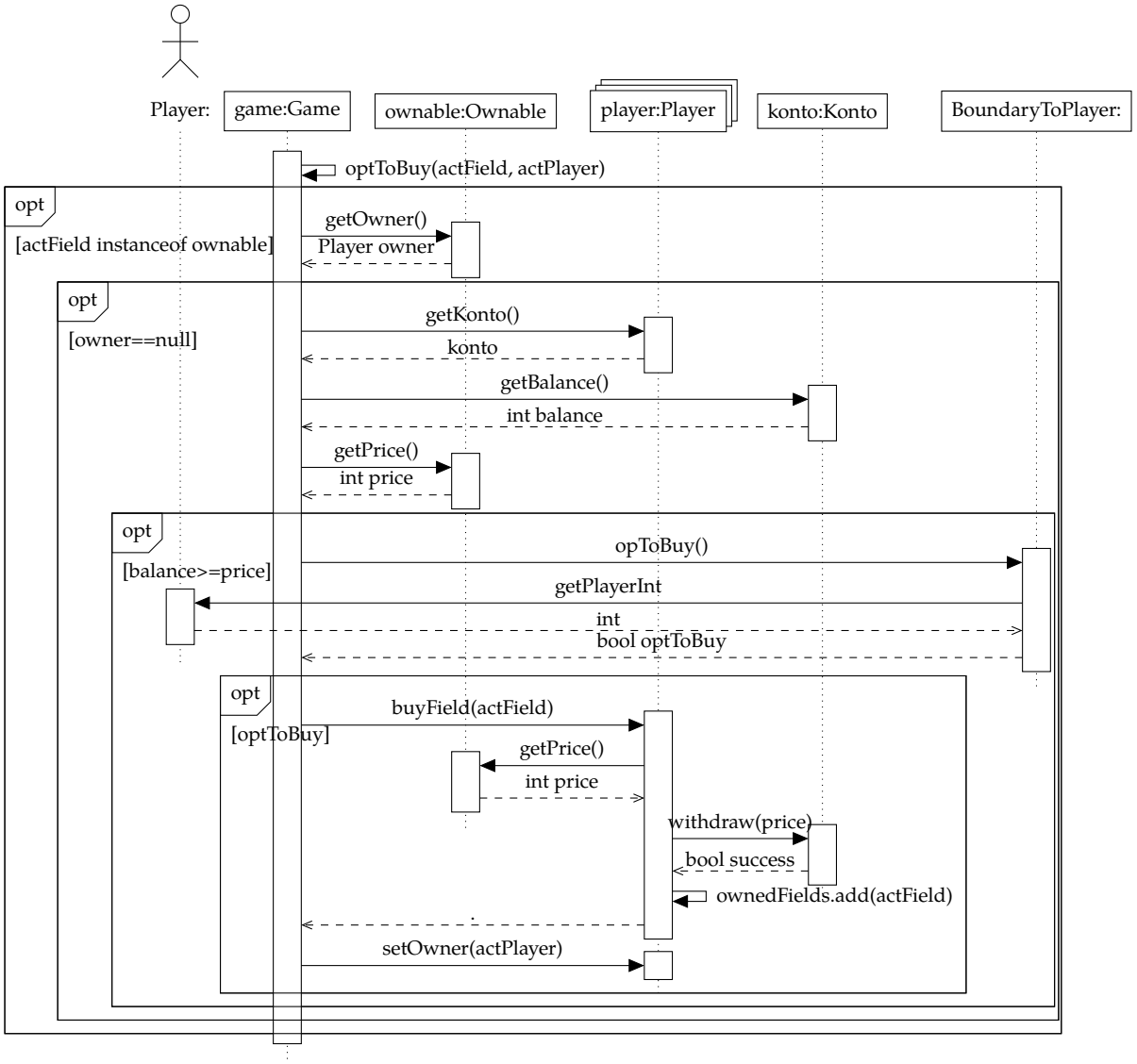
Hvis feltet kan købes undersøges om spilleren har nok penge i kontoen til at købe feltet, hvis spilleren har nok penge gives muligheden for at købe feltet.

Hvis spilleren accepterer at købe feltet trækkes pengene fra spilleren konto og feltet tilføjes til spillerens liste over ejede felter. Derudover sættes spilleren også som ejer af feltet i feltet.

`landOnField(Player)` håndterer den begivenhed at en spillet lander på et felt. Implementeringen af `landOnField(Player)` varierer for de forskellige typer af felter. Fælles er dog at de alle sammen sørger for at lave de nødvendige transaktioner med spillerens konto som resultat af at lande på feltet. Som eksempel er vist metoden for felter af typen *Ownable* da metoden er mere kompliceret for disse felter da der eventuelt skal overføres penge mellem spillere.

Sekvensdiagrammet ses i figur 3.4 på side 12, metoden starter med et kald fra *Game*. Først og fremmest ses at hvis spilleren der lander på feltet også ejer feltet sker der intet. Ellers beregnes lejen(rent), prisen for at lande på feltet. Lejen trækkkes fra spillerens konto og hvis spilleren har nok penge til at betale lejen vil lejen blive overført til ejeren af feltet, hvis der er en sådan. Hvis spilleren ikke har nok penge til at betale lejen sættes spilleren som taber i spillet.

Figur 3.3: *Sekvensdiagram over `optToBuy(Field, Player)`.*

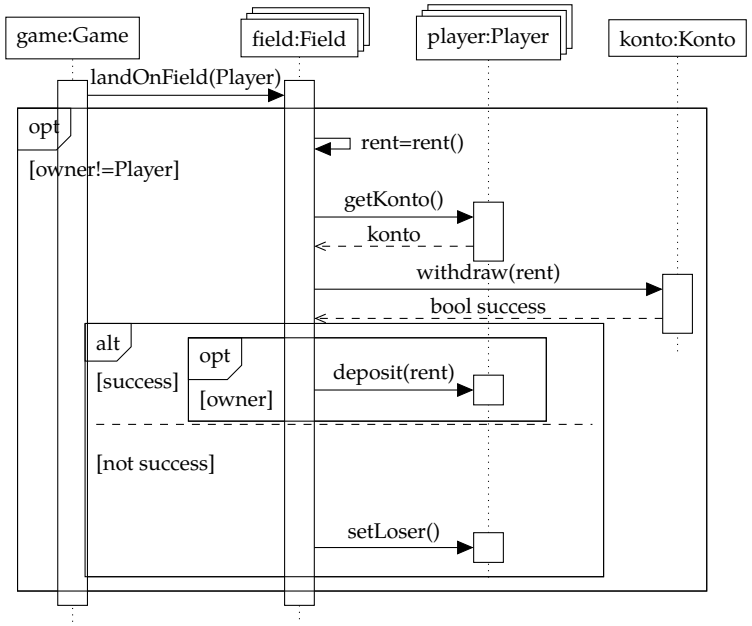


Efter dette gives kontrollen tilbage til `Game`.

4 Implementering[1]

Dette afsnit vil gennemgå dele af koden som er særligt interessante, al koden kan ses i Eclipse projektet. Implementeringen af Singleton design pattern, brug af `ArrayList`, `Iterator` og instanceof. Derudover er der brugt exception håndtering i boundaryen, dette blev behandlet i rapporten til del 2. Eneste ændring der er blevet lavet i forbindelse med exception håndteringen i forhold til del 2 er at kun exception af typen `InputMismatchException` fanges. Det er nemlig denne type exceptions der throwes hvis der indtastes noget der ikke kan matches som en `Integer`. [Ora12a]

Figur 3.4: *Sekvensdiagram over* `landOnField(Player)`.



4.1 Singleton Pattern

Som beskrevet i afsnit 3.2.3 på side 10, anvender implementeringen af spillet Singleton design pattern for at sikre at der kun findes en instans af *Game*. Derudover giver implementeringen mulighed for at andre klasser kan tilgå denne instans af *Game* ved at kalde `getGame()` som er en statisk metode i *Game*. Singleton blev valgt for at sikre at *Brewery* kunne tilgå *Game* for at få adgang til terningerne som skal bruges til at regne ud hvad det koster at lande på feltet.

Figur 4.1: *Koden som implementerer* Singleton *i* *Game*.

```
1  public class Game {
2     (...)
3     private static Game game = new Game();
4
5     private Game() {
6        (...)
7     }
8
9     public static Game getGame() {
10       return game ;
11    }
12 }
```

Koden som implementerer Singleton ses i figur 4.1 på denne side. Implementeringen bygger på at der defineres en statisk variabel Kaldet game som bliver initialiseret til at indeholder en instans af *Game*. Initialiseringen af variablen kunne også være udført i getGame() hvor man kunne kontrollere om der allerede fandtes en instans. Som hjælpelæren påpegede og litteratur omkring emnet også påpeger kan dette give problemer

i forbindelse med multithreaded applikationer. I sådanne tilfælde er det muligt at to kald til `getGame()` det sker simultant giver to forskellige instanser af *Game*. Vores applikation er ikke multithreaded så vi kunne ligeså godt have valgt den implementering.

Konstruktøren sættes til private sådan at andre klasser ikke kan lave en ny instans af *Game* ved at kalde `new Game()`.

Metoden `getGame()` er ansvarlig for at returnere instansen af *Game*.

4.2 ArrayList

Player som repræsenterer den fysiske spiller holder styr på hvilke felter en spiller ejer. Denne information bliver holdt i en *ArrayList* af *Ownable* objekter. Fordelen ved at bruge *ArrayList* er at *ArrayList* er et sandt objekt som har metoder til at lave forskellige operationer. Implementeringen i Java bruger *Array* men størrelsen af listen vokser automatisk efter behov, fra starten af har arrayet en størrelse på 10. I spillet vides det ikke fra starten af hvor mange felter en spiller kommer til at eje derfor er det en fordel at bruge *ArrayList*.[SB05]

Figur 4.2: *Koden som implementerer ArrayList i Player.*

```
1  public class Player {
2    (...)
3    private List<Ownable> ownedFields = new ArrayList<Ownable>();
4    (...)
5    public void buyField(Ownable field){
6      this.konto.withdraw(field.getPrice());
7      this.ownedFields.add(field);
8    }
9    (...)
10 }
```

Koden som implementerer *ArrayList* ses i figur 4.2 på denne side. Det ses at den private variabel ownedFields peger på den *ArrayList* som holder de felter som den pågældende spiller ejer, altså felter af typen *Ownable*. Det ses at ownedFields er af typen *List*, *List* er en interface some *ArrayList* og andre lister implementerer. Fordelen ved dette er at en anden type liste senere kan vælges så længe den også implementerer *List* interfacen.[Ora12d]

Når en spiller køber et felt tilføjes det pågældende felt til listen af ownedFields ved at kalde metoden `add(field)` på *ArrayList*en.

4.3 Brug af Iterator

Listen af ejede felter som omtales i afsnit 4.2 på denne side over ejede felter bruges for at undersøge hvor mange felter af en type en given spiller ejer. Dette er nødvendigt fordi felter af typen *Brewery* og *Shipping* har en leje der afhænger af hvor mange felter af den givne type ejeren har.

Figur 4.3: *Koden som implementerer `Iterator` i `Shipping`.*

```
1  protected int rent() {
2    if (owner != null) {
3      List<Ownable> ownedFields = owner.getOwnedFields();
4      int numFields = 0;
5      Iterator<Ownable> ownIter = ownedFields.iterator();
6
7      while (ownIter.hasNext()) {
8        if (ownIter.next() instanceof Shipping) {
9          numFields++;
10       }
11     }
12     (...)
13   }
```

Koden i figur 4.2 på foregående side viser implementeringen af *`Iterator`* i *`Shipping`*, næsten ens kode bruges i *`Brewery`*. Det ses at listen af ejede felter hentes ved at kalde `getOwnedFields()` på owner som er en henvisning til ejeren af det pågældende felt. Der laves en *`Iterator`* af ejede felter ved at kalde `iterator()` på listen af ejede felter. *`Iterator`* har tre metoder, men vi bruger kun to: `hasNext()` og `next()`. `hasNext()` returnerer true hvis der er et næste element som iteratoren kan returnere, vi bruger denne metode til kun at kalde `next()` så længe der er flere elementer.[Ora12c]

`next()` returnerer det næste element fra iteratoren, det undersøges herefter om det returnerede element er en instans af den pågældende klasse *`Shipping`* eller *`Brewery`*. Hvis det er tilfældet øges numFields med 1. Når iteratoren er færdig vil numFields indeholde en int svarende til mængden af samme type felter som ejeren af feltet ejer.

I den udeladte kode som er forskellig for de to klasser bruges numFields til at regne ud hvor dyrt det er at lande på feltet.

4.4 Brug af instanceof operatøren

Til at regne ud hvad det skal koste at lande på *`Shipping`* og *`Brewery`* er der brug for at vide hvor mange felter af samme type ejeren af det pågældende felt ejer. Udregningen af dette er implementeret ved hjælp af instanceof operatøren og *`Iterator`* som er omtalt i afsnit 4.3 på forrige side.

instanceof operatøren kontrollerer om et givent objekt er en instans af en given klasse, eller underklasser af den givne klasse. Koden som implementerer dette kan ses i figur 4.3 på denne side i linie 8. <instans> instanceof <klasse> vil altså returnerer true hvis <instans> er en instans af klasse <klasse>. Vedhjælp af *`Iterator`* gåes alle felterne som spilleren ejer igennem og antallet af felter der er en instans af det pågældende felt tælles. På denne måde har vi mulighed for at tælle hvor mange felter af en given type en spiller ejer.[Ora12b]

5 Test[1]

Test blev behandlet i kurset 02313 Development methods for IT-Systems, kravet var at der skulle laves bruger-/systemtest af systemet til spillet. Denne skulle ydermere laves uden at testeren kender til implementeringen.

5.1 Brugertest

Det eneste eksterne system som matadorspillet modtager input fra er den fysiske spiller som sender input til systemet igennem konsollen. Systemet tager imod input fra spilleren to gange, når en spiller skal slå med terningerne og når en spiller skal vælge om vedkommende vil købe en grund.

5.1.1 Slå med terninger

Med hensyn til at slå med terningerne skal spilleren indtaste sit spillernummer for at slå med terningerne, dvs spiller 1 skal indtaste 1 for at slå med terningerne når det er vedkommendes tur. „Kontrakten“ med den fysiske spiller ligger i den tekst som bliver printet til spilleren når vedkommende skal taste: „Det er spiller 1’s tur. Tast 1 for at slå:“.

Som tester ses det altså at der kun er én gyldig ækvivalensklasse, nemlig 1. Der er dog adskillige ugyldige ækvivalensklasser, ækvivalensklasserne kan ses i figur 5.1 på denne side.

Figur 5.1: *Ækvivalensklasser for „Slå med terningerne“ for Spiller 1.*

Ækvivalensklasser	Gyldighed	Input	Forventet Resultat
input == 1	gyldig	1	"slår med terninger"
input != 1	ugydlig	2	"Slår ikke"
input != int	ugydlig	"test"	"Slår ikke"
input != int	ugyldig	0.5	"Slår ikke"

Tilsvarende ækvivalensklasser vil selvfølgelig findes for spiller 2 hvor det gyldige input i stedet for er 2.

Selve testning af de forskellige ækvivalensklasser blev udført ved at spillet blev kørt i Eclipse og de forskellige indtastninger forsøgt. De forskellige indtastninger og resultater er listet i figur 5.2 på denne side.

Figur 5.2: *Test af ækvivalensklasser for „Slå med terningerne“ for Spiller 1.*

Ækvivalensklasser	Gyldighed	Input	Resultat
input == 1	gyldig	1	"Status, Terning 1: 5, Terning 2: 6"
input != 1	ugyldig	2	"Det er spiller 1’s tur. Tast 1 for at slå:"
input != int	ugyldig	"test"	"Kun integers er tilladt, prøv igen:"
input != int	ugyldig	0.5	"Kun integers er tilladt, prøv igen:"

Det ses fra testen at når det gyldige input 1 bliver tastet ind bliver terningerne slået.

Hvis der tastes en int som ikke er lig med spillerens nummer så får spilleren igen en mulighed for at taste sit nummer ind. Dette gentager sig indtil spilleren taster sit nummer ind, spilleren har altså ikke mulighed for at undlade at slå.

Hvis der derimod tastes noget som ikke kan fortolkes som en int, som bogstaver eller kommatal, så vil resultatet være at spillet vil anmode spilleren om at taste en integer ind i stedet for.

5.1.2 Køb af grund

Hvis spilleren ønsker at købe en grund skal spilleren indtaste 1 for at købe grunden. Hvis spilleren ikke ønsker at købe grunden skal spilleren indtaste en anden int end 1. Kun ints er gyldige inputs. „Kontrakten“ med den fysiske spiller er givet i tekst som bliver printet til konsollen. Teksten der bliver printet lyder som følger: „Du landede på: <feltNavn>. Ønsker du at købe feltet for <Pris>?, Tryk 1, derefter Enter for at købe, tryk en anden int og derefter Enter for ikke at købe.“.

I modsætning til tilfældet hvor der skal slås med terninger er alle ints nu gyldige ækvivalensklasser, den eneste int der forårsager et køb af et felt er dog 1. Ækvivalensklasserne ses i figur 5.3 på denne side.

Figur 5.3: *Ækvivalensklasser for „Køb af grund“.*

Ækvivalensklasser	Gyldighed	Input	Forventet Resultat
input == 1	gyldig	1	"Køber grund"
input != 1	gyldig	2	"Køber ikke"
input != int	ugyldig	"test"	"Køber ikke"
input != int	ugyldig	0.5	"Køber ikke"

Selve testningen blev foretaget ved at spillet blev kørt og de forskellige input testet manuelt. De forskellige indtastnigner og resultater er listet i figur 5.4 på denne side.

Figur 5.4: *Ækvivalensklasser for „Køb af grund“.*

Ækvivalensklasser	Gyldighed	Input	Resultat
input == 1	gyldig	1	"Du har købt <feltNavn>"
input != 1	gyldig	2	"Du købte ikke <feltNavn>"
input != int	ugyldig	"test"	"Kun integers er tilladt, prøv igen:"
input != int	ugyldig	0.5	"Kun integers er tilladt, prøv igen:"

Det ses fra resultatet af testningen at hvis der tastes 1 så købes feltet som kontrakten også foreskrev.

Hvis der indtastes andre ints end 1 så købes feltet ikke men spillet fortsætter.

Hvis der tastes noget ind som ikke kan fortolkes som en int så vil spilleren blive anmodet om at taste en int ind inden at spillet vil fortsætte.

5.1.3 Test af spillets funktion

Udover testning af de forskellige input som blev gennemgået i de foregående afsnit blev spillet også testet ved at køre spillet gentagende gange. Hvad der sker i spillet

er tilfældigt ved hjælp Javas java.util.Random. Derfor fungerede denne del af testning ved at det blev kontrolleret at spillet reagerede korrekt på de forskellige værdier som terningerne antog under testningen.

På samme måde blev det kontrolleret at spillet overførte prisen for at lande på et felt til ejeren af feltet hvis feltet var ejet.

6 Konklusion[1, 2]

Formålet med CDIO del 3 var at implementere et klassehierarki over felterne i et matadorspil, med tilhørende abstrakte metoder, til brug for videreudvikling af matadorspillet fra CDIO del 2. De implementerede felter skulle repræsentere de fysiske matadorfelter i funktionalitet. Desuden skulle udvikles en fuld domænemodel over et matadorspil.

De udleverede krav i [Nyb12] blev tolket og opstillet i henhold til UP modellen for kravspecificering i kapitel 2 på side 3.

Domænemodellen over matadorspillet blev gennemgået og kommenteret i kapitel 3 på side 7. I samme kapitel blev de vigtigste ændringer i spillet siden CDIO del 2 gennemgået ved hjælp af designklasse- og sekvensdiagrammer.

I kapitel 5 på side 15 blev black box testning af spillet gennemgået som en bruger ville opleve systemet.

Koden i Eclipse lever nu op til kravene i [Nyb12] og er altså en videreudvikling af CDIO del 2 som implementerer det nye klassehierarki over felterne. I kapitel 4 på side 11 bliver de mest interessante dele af koden gennemgået, med fokus på emner som ikke har været gennemgået i undervisningen.

Referencer

[2112] Gruppe 21. *CDIO del 2*. DTU, 12. nov. 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 Nybord. *Del opgave 3. Matador Felter - polymorfisme og domænemodel*. Vers. 2012-08-08. Opgavebeskrivelse udleveret i forbindelse med projektet. DTU, 8. aug. 2012.

[Ora12a] Oracle. *Class InputMismatchException*. 8. nov. 2012. URL: `http://docs.oracle.com/javase/7/docs/api/java/util/InputMismatchException.html` (sidst set 08.11.2012).

[Ora12b] Oracle. *Equality, Relational, and Conditional Operators*. 28. nov. 2012. URL: `http://docs.oracle.com/javase/tutorial/java/nutsandbolts/op2.html` (sidst set 28.11.2012).

[Ora12c] Oracle. *Iterator*. 24. nov. 2012. URL: `http://docs.oracle.com/javase/7/docs/api/java/util/Iterator.html` (sidst set 24.11.2012).

[Ora12d] Oracle. *List*. 24. nov. 2012. URL: `http://docs.oracle.com/javase/7/docs/api/java/util/List.html` (sidst set 24.11.2012).

[SB05] Kathy Sierra og Bert Bates. *Head First Java*. 2. udg. O'Reilly Media, 2005.

A Supplementary Specification artefakt[2]

1. `toString` føjes til alle klasser
2. *`Field`* skal have attributterne *number* og *name* samt `landOnField()` skal tage *Player* som argument.
3. *`Refuge`* skal have attribut *bonus* (kontant beløb).
4. *`Tax`* skal have attribut *tax* (kontant beløb).
5. *`Ownable`* implementeres som abstrakt klasse der definerer alle felter der kan ejes.
6. *`Street`* simplificeres til kun at have en leje.
7. Hvis der landes på *`Shipping`* betales der leje til ejeren af feltet alt efter hvor mange *`Shipping`*-felter (*rederier* i formel) denne ejer. Lejen udregnes som:

$$leje = 2^{(rederier)} * 500$$

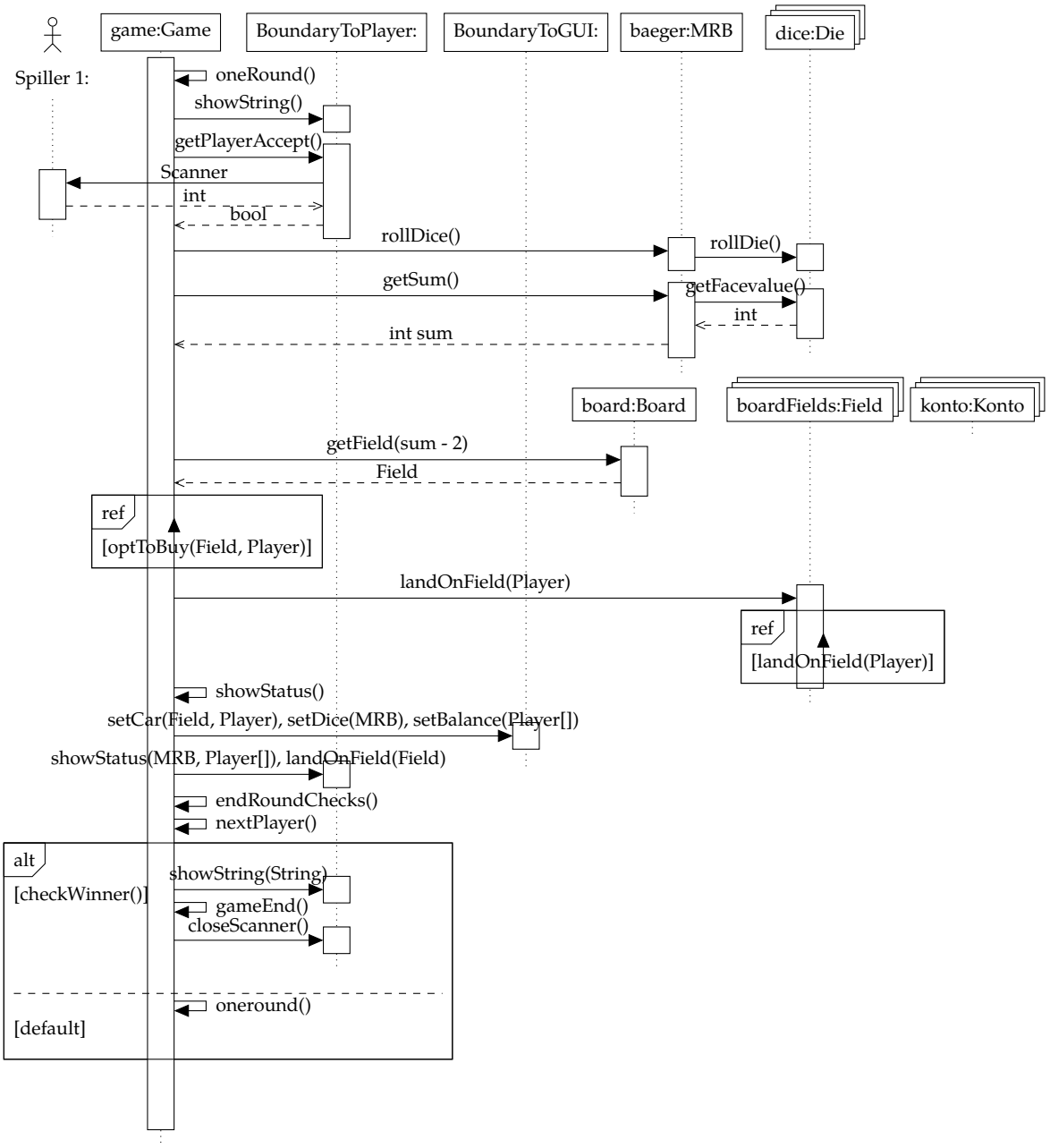
8. Hvis der landes på *Brewery* betales der leje til ejeren udfra antallet af øjne i slaget der bragte ham til feltet (*slag* i formel) samt hvor mange bryggerier denne ejer (*bryggerier* i formel. Lejen udregnes som:

$$leje = (slag) * (bryggerier) * 100$$

9. Modificer CDIO-opgave del 2 så denne anvender det nye arveheiraki.
10. Deklarér klasserne *Field* og *Ownable* som abstrakte og deres metoder `landOnField()` og `rent()` skal ligeledes være abstrakte.
11. Lav et program som indeholder et array af typen Field, som udskriver indholdet af hvert objekt i array'et. Du skal minimum oprette en instans af hver klasse.
12. Domænemodel skal fuldt udvikles og kommenteres.
13. De beskrevne udvidelser blev implementeret i CDIO-opgave del 2 og er således stadig en del af systemet.

B Sekvensdiagram over hele spillet[1]

Figur B.1: *Sekvensdiagram som viser hele spillet.*



Rettelser