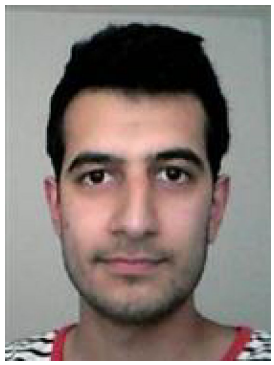
Projektnavn: CDIO Final
Gruppenummer: *18*
Afleveringsfrist: *Mandag den 16/01 2017 Kl. 12:00*
Fag og retning: Diplom Softwareteknologi
Denne rapport er afleveret via Campusnet (der skrives ikke under).
Denne rapport indeholder **52** sider inklusiv denne.

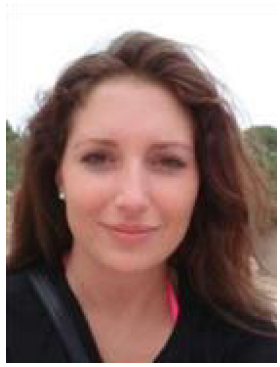
Kasper Leiszner
s165218

Frederik Von Scholten s145005



Bijan Negari
s144261

Troels Lund
s161791



Helene Zgaya
s104745

Resumé

Vores projekt går ud på at udvikle et Matador spil for to til seks spillere. Ligesom brætspillet Matador består vores spil af en spilleplade på 40 felter hvoraf der findes 8 forskellige typer: Brewery, Fleet, Plot, Chance, Parking, Tax og Prison, hver med deres egen unikke egenskaber.
Felterne Brewery, Fleet og Plot kan ejes af en spiller. Hvis en spiller lander på et felt af disse typer og det ikke er ejet af en anden spiller, kan spilleren som landte på feltet vælge at købe feltet, og skal derefter modtage et beløb hvis en anden spiller lander på dit eget felt.
Derudover hvis en spiller 1 ejer alle Plots i samme farvegruppe, skal en spiller 2 der lander på et af disse Plots, betale 2 gange normalt beløb.
Lander du på et Chance-felt, trækker du et tilfældigt kort fra vores kortbunke på 40 kort.
Kortet bestemmer om spilleren skal modtage eller betale et beløb, rykke sin brik eller få muligheden at slippe fri fra fængsel hvis man senere skulle havne der.
På feltet Parking, sker der ingenting. F.eks. er "START"-feltet et Parking-felt.
Der findes to Tax-feltet. Lander man på det ene betaler spilleren altid et fast beløb. På det andet kan man vælge et fastlagt beløb, eller 10% af sine penge i banken.
Feltet Prison har den egenskab at hvis man ender i fængslet får man 3 valgmuligheder. Enten at bruge et get-out-of-jail card, hvis man i løbet af spillet har modtaget sådan et, forsøge at slå 2 ens med terningerne eller betale et beløb for at komme ud igen.
Hver spiller starter med 1.500 kr. Spillet fortsætter indtil der kun er en enkelt spiller tilbage, som så er vinderen. Der er brugt flere UML diagrammer til både analyse- og design delen af programmet. Desuden er programmet blevet testet.
Vi har formået at opfylde kravene og vores program er testet. Alt i alt har vi udviklet et velfungerende program.

Timeregnskab

Dato*	Deltager	Design & Analyse	Impl.	Test	Dok.	Andet	Ialt
02-01-2017	Troels					3	3
02-01-2017	Kasper	1				2	3
02-01-2017	Helene					3	3
02-01-2017	Bijan		1			2	3
02-01-2017	Frederik					3	3
03-01-2017	Troels	3				2	5
03-01-2017	Kasper	1			1	3	5
03-01-2017	Frederik	3				2	5
03-01-2017	Bijan	3				2	5
03-01-2017	Helene	3				2	5
03-01-2017	Troels	2	2	1			5
04-01-2017	Bijan					1	1
04-01-2017	Frederik					2	2
05-01-2017	Helene		5				5
06-01-2017	Helene	4	3				7
05-01-20	Troels		4				4

17							
05-01-2017	Bijan		4				
05-01-2017	Frederik		5				5
06-01-2017	Troels	1	3			1	5
04-01-2017	Kasper		2			1	3
05-01-2017	Kasper		3				3
06-01-2017	Bijan		4				
06-01-2017	Kasper		2			1	3
06-01-2017	Frederik		5				5
06-01-2017	Bijan		3				
08-01-2017	Kasper		1				1
09-01-2017	Bijan		3			1	
09-01-2017	Kasper	1	2			1	4
09-01-2017	Frederik		5				5
09-01-2017	Helene	2	3				
10-01-2017	Bijan		4				
10-01-2017	Kasper			4		1	5
10-01-2017	Frederik	5					

11-01-2017	Helene	5					5
11-01-2017	Kasper		3	1		1	5
11-01-2017	Bijan	1	3				4
11-01-2017	Frederik	3				2	
12-01-2017	Bijan		3				3
12-01-2017	Frederik					6	6
12-01-2017	Helene	5					
13-01-2017	Bijan				2	1	3
13-01-2017	Frederik	3,5			2		5,5
13-01-2017	Helene	2				2	4
7-1-2017	Troels	1	4				
14-01-2017	Frederik					2	
15-01-2017	Frederik					2	
16-01-2017	Kasper	0,5			1		1,5
16-01-2017	Helene					2	2
17-01-2017	Frederik					2	2
10-01-2017	Troels		5			1	6
11-01-2017	Troels				5		5

08-01-2017	Troels		4				
12-01-2017	Troels				5		
13-01-2017	Troels				2		
14-01-2017	Troels		1		3	1	
16-01-2017	Troels	1			2	2	
	SUM	50	82	6	11	54	155

Indholdsfortegnelse

Indledning

Denne rapport er udarbejdet som led i undervisningen i fagene Indledende programmering (02312), Udviklingsmetoder til IT-systemer (02313) og Versionsstyring og testmetoder (02315) på første semester på diplom software.
Rapporten dokumenterer planlægning og udvikling af et program der skal simulere et matador brætspil mellem 2-6 spillere. Vi har skrevet i det objektorienteret programmeringssprog Java.
Først og fremmest har vi udfærdiget en kravspecifikation lavet på baggrund af spillereglerne for matador, dog med modificeringer. Denne beskrives mere udførligt i hovedafsnittet.
Derudover har vi benyttet flere modeller til at visualisere ved analyse samt dokumentation for udvikling af vores program.
Det primære mål er altså at udvikle matador spillet ved hjælp af java og implementere udvalgte features samt implementere en GUI der er blevet udviklet på forhånd.
Brætspillet foregår altså mellem 2-6 spillere der skiftes til at slå med to terninger. Værdien af disse placerer spilleren på et af 40 felter der har individuel effekt for spilleren (se feltliste i bilag).
Spillerens pengebeholdning starter på 1500. Går en spiller bankerot, bliver spillerens ejede felter ledige igen for salg såfremt der er flere end 2 spillere. Når alle spillere er bankerot på nær en, slutter spillet.
En beskrivelse af implementeringen generelt samt uddybende beskrivelse af dele af koden vil være at finde i denne rapport

Opsummering
Følgende er altså beskrevet i denne rapports hovedafsnit:
Krav og analyse (dertilhørende modeller)
Design-dokumentation (dertilhørende modeller)
Beskrivelse af koden
Test samt dertilhørende beskrivelser

MoSCoW (findes som bilag 6)

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

Hovedafsnit

Analyse

- Kravspecifikation
- Domænemodel
- Use-case diagram
- Use-case beskrivelser
- System Sekvens Diagrammer

Kravspecifikation

Functionality

- R1: Spillet skal foregå mellem 2 til 6 spillere.
- R2: Spillerne slår på skift, med de 2 terninger.
  - R2.1: Hver terning har 6 sider som er lige sandsynlige.
    - R2.1.1: Det skal vær let at skifte terning.
  - R2.2: Spillerne lander på det felt (nr.) som er summen af de 2 terninger angiver.
- R3: Spillet skal have 40 felter nummeret 1 - 40.
- R4: Spillet skal have felterne af forskellig felttyper som skal have egenskaber som beskrevet i listen over felter.
- R5: Spilleren skal kunne lande på et felt og fortsætte fra det felt.
- R6: Hver spiller skal starte med kr. 1.500
- R7: Spillet slutter når der kun er en spiller tilbage som ikke er bankerot.
- R8: Hvert felt skal have en effekt som beskrevet i følgende liste over felt typer.
- R9: Efter bankerot bliver spillerens ejendomme givet tilbage til banken, såfremt de er pantsatte

Usability

- R10: Der skal findes en brugervejledning til hentning, og afvikling af spillet.
- R11: Undervejs i spillet, skal spillernes slag og point vises på GUI'en gennem hele spillet.

Reliability

Performance

- R11: Spillet skal kunne køre uden forsinkelser på over 1 sek.
- R12: Spillet skal kunne køres på DTU's maskiner i databarene.

Supportability

- R13: Spillet skal endvidere kunne skiftes til andre sprog af en udvikler.
- R14: Spillet skal være så generelt skrevet at, spilleren og hans pengebeholdning uden videre kan bruges til andre spil.

Liste over felt typer og deres egenskaber:

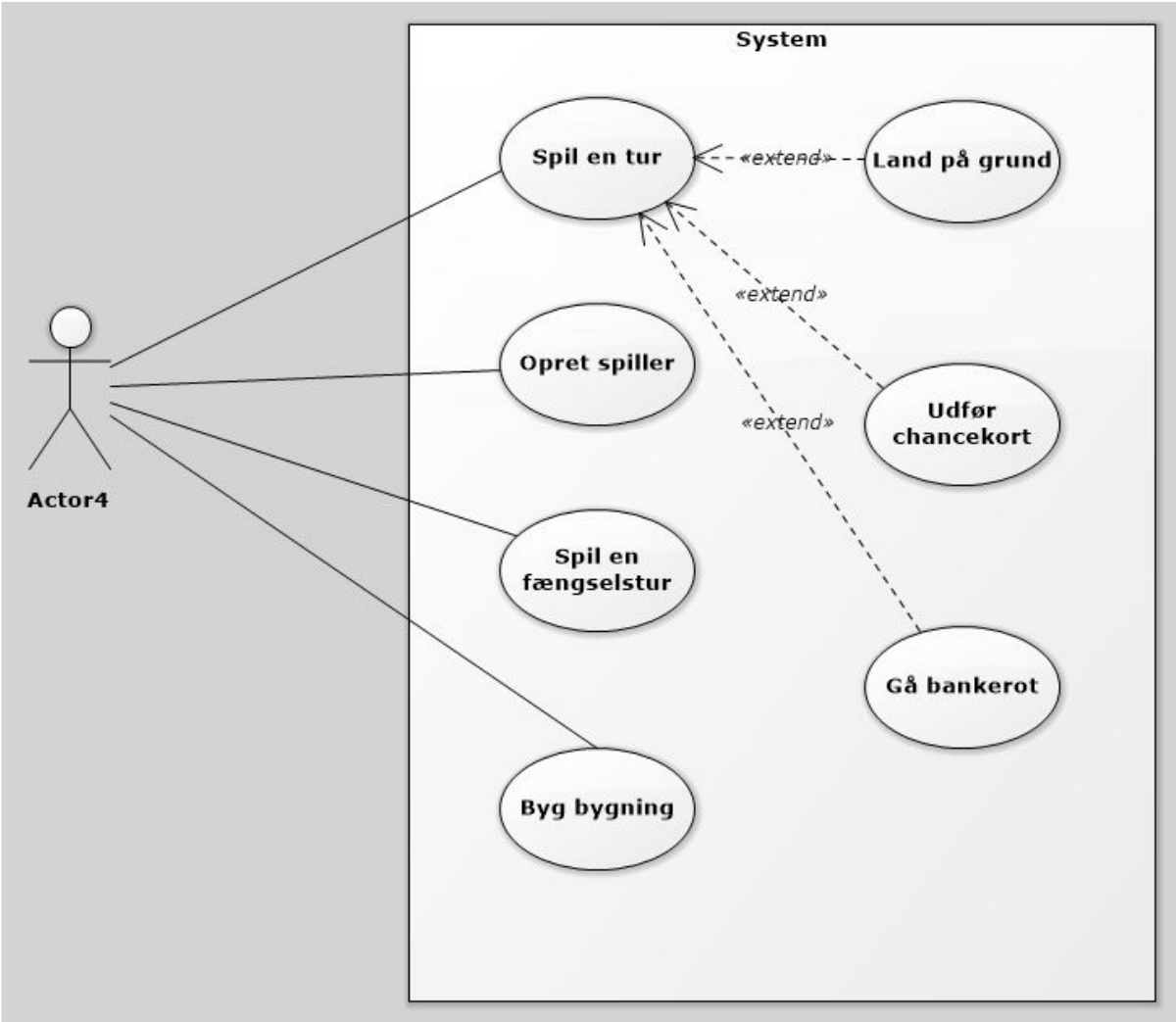
Felt-type	Egenskaber
Start	● Ingen konsekvenser på feltet ● Ved spillets begyndelse placeres spillere her ● Spillere der krydser start modtager en bonus
Parkering	● Ingen konsekvenser på feltet
Ryk i fængsel	● Spilleren fængsles og flyttes til fængselscelle
Fængsel	● Der sker intet hvis en spiller lander på feltet uden at være i fængsel

	• Spillere placeres her hvis de sættes i fængsel indtil de "frifindes"
Grund	• Feltet skal have en pris og kunne købes hvis det ikke ejes af en spiller i forvejen, når en spiller lander på feltet. • Feltet skal have en leje. • Hvis en spiller lander på et felt der ikke er ejet af den spiller, men ejet af en anden spiller, skal han betale leje til ejeren. • Ejeren af grunden skal kunne bygge 4 huse på en grund. • Hvis der er 4 huse på en grund skal ejeren kunne udbygge til et hotel ved at betale med 4 huse plus • Der må maksimalt være 1 hotel på hver grund.
Redderi	• Feltet skal have en pris og kunne købes hvis det ikke ejes af en spiller i forvejen, når en spiller lander på feltet. • Alle redderier har samme pris. • Feltet skal have en leje • Lejen på rederier ejet af samme spiller skal øges hvis den samme spiller ejer flere redderier felter
Bryggeri	• Feltet skal have en pris og kunne købes hvis det ikke ejes af en spiller i forvejen, når en spiller lander på feltet. • Feltet skal have en leje • Lejen på bryggeri regnes ud ved at slå med terningerne, og gange antal øjne vist med hhv. 4 og 10. • Ejer ejeren af et bryggeri også et andet bryggeri er lejen 10, ellers 4.

Domænemodel

Se bilag 1

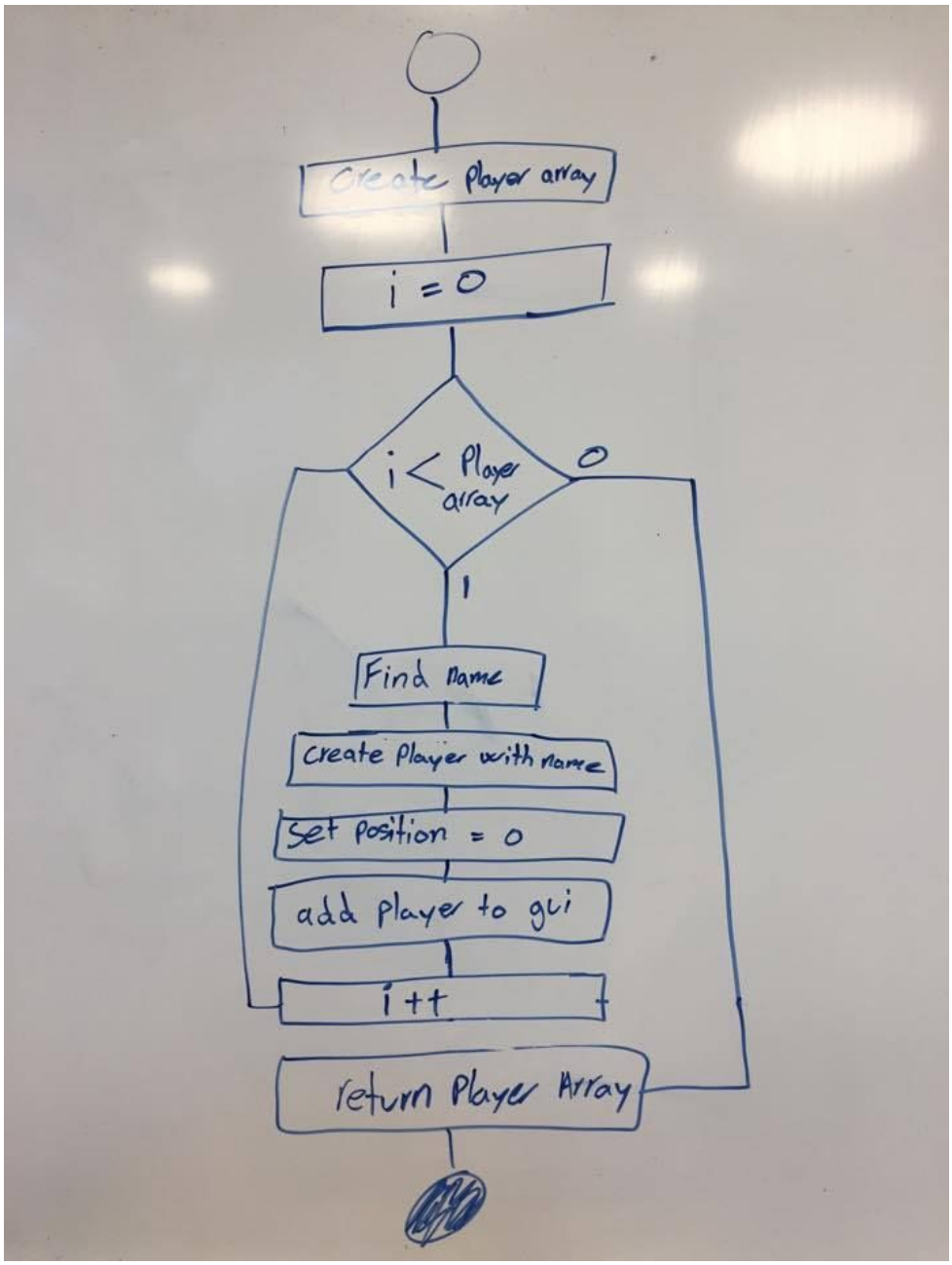
Use-case diagram



Use-case beskrivelser

Use case: Spil en tur
ID: 01
Brief description: En spiller udfører hele sin tur
Primary actors: Spiller
Secondary actors: Ingen
Precondition: 1. 3 spillerer har tilsluttet sig spillet 2. Alle spillere modtager 1500,- 3. Spillet er påbegyndt 4. Det er den første spillers tur
Main flow: 1. Spilleren kaster terningerne, terningernes øjne viser 2 og 3 2. Spilleren rykker 5 felter frem 3. Denne grund er ikke ejet af en anden spiller 4. Spilleren vælger at købe grunden til 200,- 5. Spillerens nye balance er nu 1.300,- 6. Feltet er nu ejet af spilleren
Alternativ flow: 1. Spilleren kaster terningerne, terningernes øjne viser 5 og 5 2. Spilleren rykker 10 felter frem 3. Denne grund er et bryggeri ejet af en anden spiller 4. Den anden spiller ejer kun 1 bryggeri 5. Spilleren slår med terningerne, terningernes øjne viser 4 og 5 6. Spilleren betaler til 36,- til bryggeriets ejer 7. Spillerens nye balance er 1464,- 8. Ejeren af bryggeriet får tilføjet 36,- til sin konto 9. Spilleren får en ekstra tur
Postcondition: 1. Turen går videre til den næste spiller

Use case: Opret spillere
ID: 02
Brief description: Oprettelse af 2 til 6 spillere
Primary actors: Spiller
Secondary actors: Ingen
Precondition: 1. Start program
Main flow: 1. Vælg antal spillere mellem 2 og 6 2. 3 antal spillere vælges 3. Spiller 1's navn angives som "Kasper" 4. Spiller 2's navn angives som "Bijan"
Postcondition: 1. 3 spillere er oprettet ved navn "Kasper", "Bijan" og "RonnieRonnie" og de får hver især 1500,- på deres bankkonto 2. Spillet starter



Flowchart over for loop der opretter player

Use case: Land på grund
ID: 03
Brief description: Spiller lander på en grund, mens det er hans tur
Primary actors: Spiller
Secondary actors: Ingen
Precondition: 1. 3 spillere er oprettet 2. Hver spiller har 1.500,- 3. Spillet er påbegyndt 4. Det er din tur
Main flow: 1. Du er landet på en grund 2. Grunden koster 350,- 3. Du vælger at købe grunden 4. Du betaler 350,- 5. Din nye balance er nu 1.150,-
Alternativ flow: 1. Du er landet på en grund 2. Grunden er ejet af en anden spiller 3. Du betaler 80,- i leje til grundens ejer 4. Din nye balance er 1.420,- 5. Ejerens nye balance er 1.580,-
Alternativ flow: 1. Du er landet på en grund 2. Grunden er ikke ejet 3. Du vælger ikke at købe grunden
Postcondition: 1. Det bliver næste spillers tur

Use case: Spil en fængselstur
ID: 04
Brief description: Land på “i fængsel”-felt, kom ud af fængsel.
Primary actors: Spiller
Secondary actors: Ingen
Precondition: Spilleren er i fængsel
Main flow: 1. Betal 50 kroner og kom ud af fængsel 2. Slå med terninger 3. Få en 5’er og en 4’er 4. Ryk 9 felter frem 5. Udfør normal tur
Alternative flow: 1. Slå med terninger 2. Få to 3’ere 3. Ryk 6 felter frem 4. Slå med terninger igen 5. Få en 1’er og en 6’er 6. Ryk 7 felter frem
Alternative flow: 1. Brug chancekort som tidligere er trukket i spillet “ryk ud af fængsel” 2. Slå med terninger 3. Få en 4’er og en 2’er 4. Ryk 6 felter frem
Alternative flow: 1. Få to forskellige værdier med terninger 2. Vent til næste tur 3. Få to forskellige værdier med terninger 4. Vent til næste tur 5. Få to forskellige værdier med terninger 6. Der trækkes automatisk 50 kroner fra konto 7. Slå med terninger 8. Få en 3’er og en 1’er 9. Ryk 4 felter frem

Postcondition Udfør feltets egenskab

Use case: Byg bygning
ID: 05
Brief description: Byg en bygning på et af dine ejet grunde
Primary actors: Spiller
Secondary actors: Ingen
Precondition: 1. 3 spillere er oprettet 2. Hver spiller har en balance på 1.500,- 3. Spillet er påbegyndt 4. Det er din tur
Main flow: 1. Du vælger at købe et hus til en af dine ubebygget grunde 2. Huset koster 200,- 3. Din balance er nu 1.300,-
Alternative flow: 1. Du vælger at købe et hus til en af dine grunde hvor der er 4 huse i forvejen 2. Husene bliver til et hotel 3. Huset kostede 200,- 4. Din balance er nu 1.300,-
Postcondition 1. Det er næste spillers tur

Use case: Gå bankerot
ID: 06
Brief description: Gå bankerot
Primary actors: Spiller
Secondary actors: Ingen
Precondition: Spillers konto går i 0
Main flow: 1. Du kan ikke sælge dine aktiver 2. Du går bankerot 3. Aktiver nulstilles
Postcondition: Spiller ude af spillet - næste spillers tur.

Use case: Udfør chancekort
ID: 07
Brief description: Udførelsen af et chancekort
Primary actors: Spiller
Secondary actors: Ingen
Precondition: Spilleren er landet på et “træk et chancekort” felt
Main flow: 1. Din balance er 650,- 2. Du trækker et chancekort 3. Der står at du modtager 100,- 4. Din balance er nu 750,-
Alternativ flow: 1. Du trækker et chancekort 2. Det er et løsladningskort til feltet fængsel 3. Du gemmer kortet til du kommer i fængsel
Postcondition: Næste spillers tur

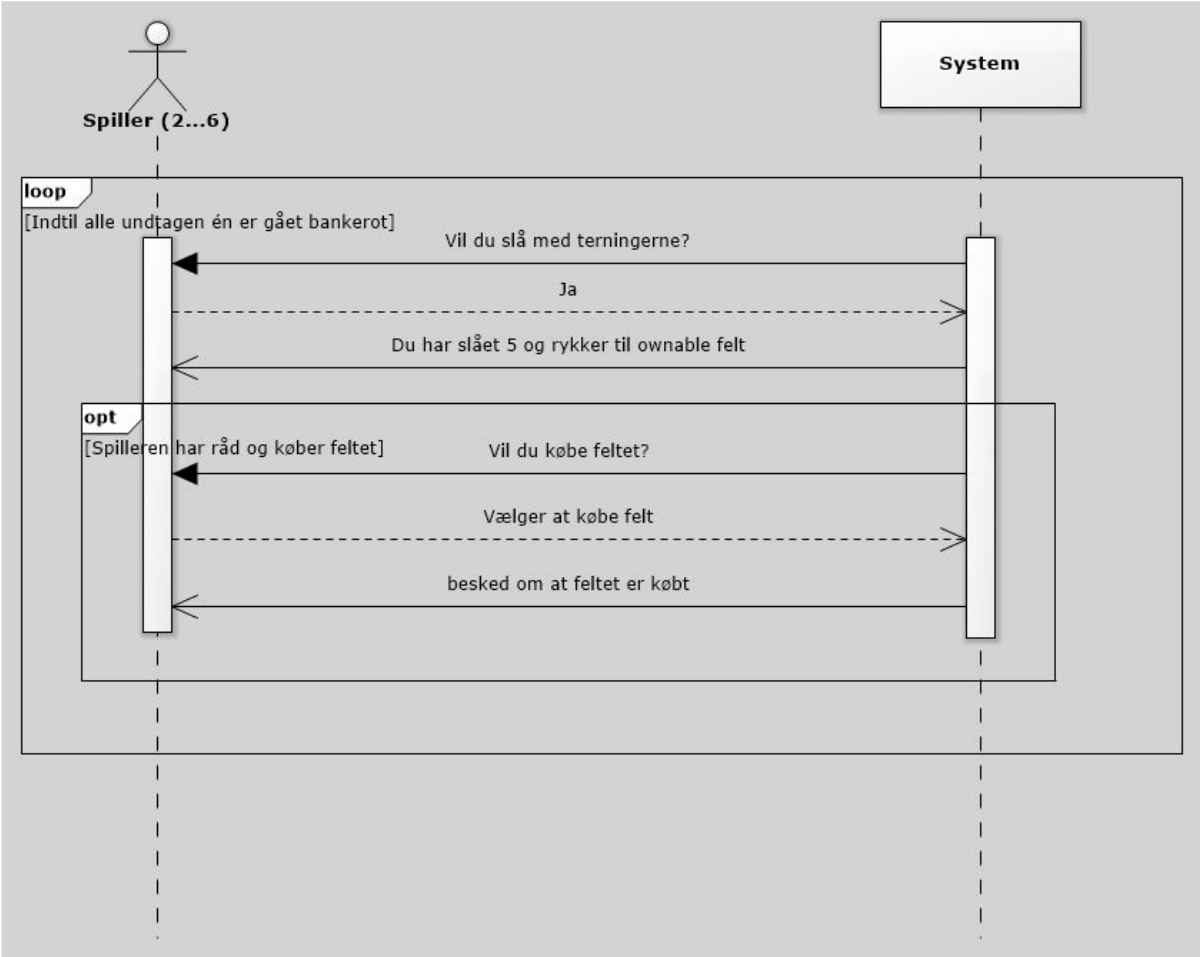
Traceability matrix

	R1	R2	R3	R4	R5	R6	R7	R8
01		x	x	x				
02	x				x			
03							x	
04							x	
05							x	
06						x		x

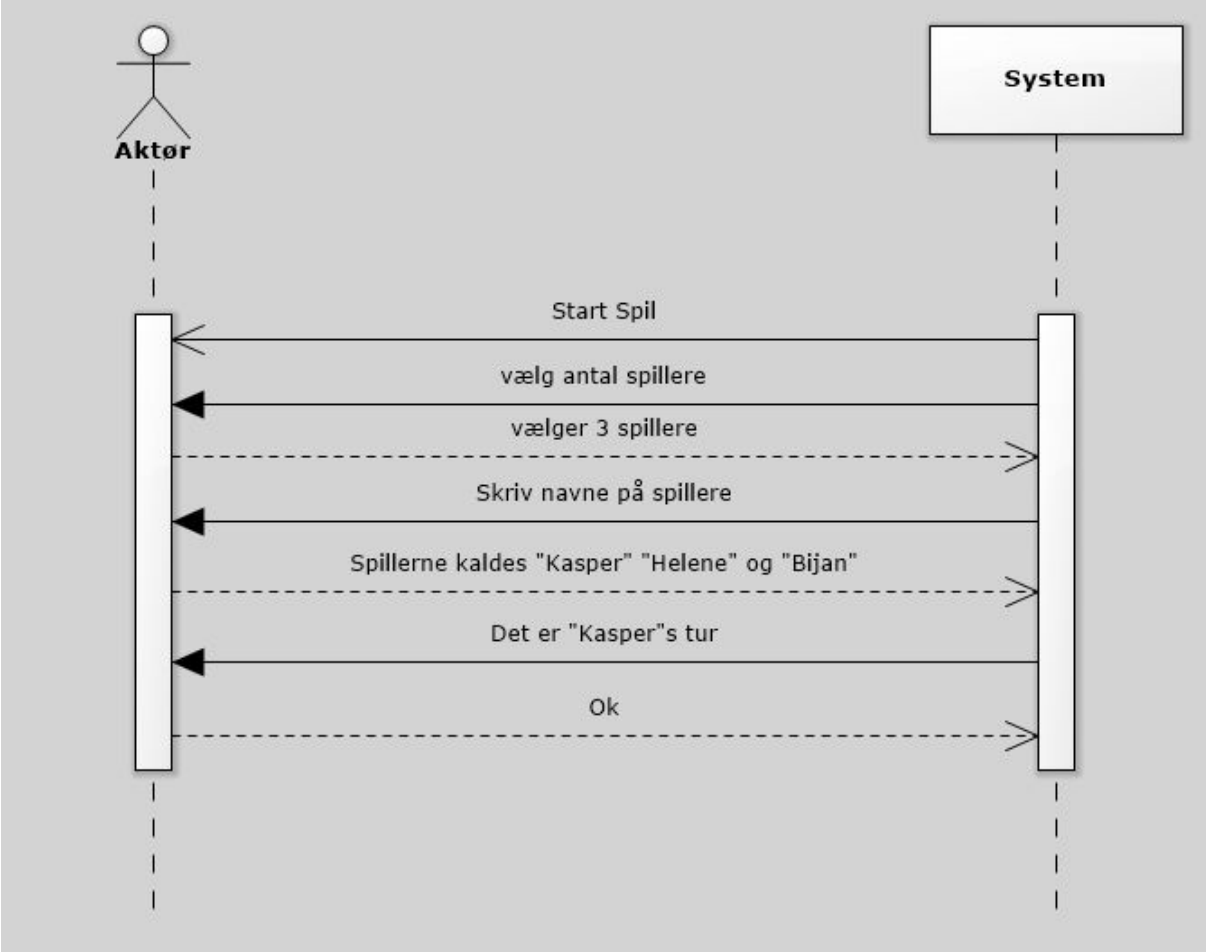
Dette diagram viser at alle funktionelle krav er medtaget i vores use-case beskrivelser.

SSD’er (System sekvens diagrammer)

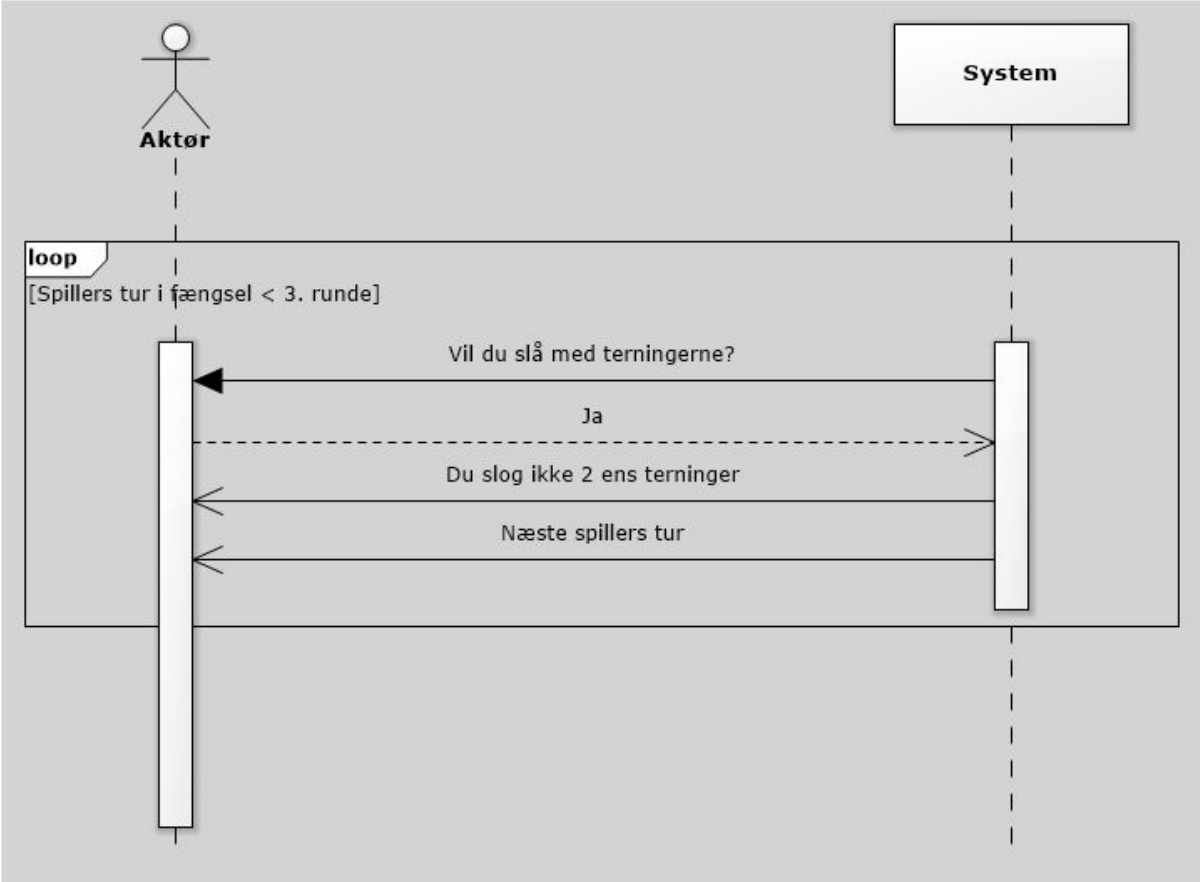
SSD Spil en tur



SSD Opret spillere



SSD Spil en fængselstur



Design

- Klassediagram
- DSD Design sekvens diagrammer

Klassediagram

Se bilag 2

DSD (Design sekvens diagrammer)

Vi har lavet tre DSD for henholdsvis “Spil en tur”, “Opret spillere” og “Spil en fængselstur”.
Se bilag

Implementering

Brug af arv og polymorfi i Felter

Vi har valgt at opbygge vores felter i et hierarki ved hjælp af nedarvning. Det vil sige vi først skriver den mest generaliseret klasse Field, for på den måde at sikre at alle felter har de generelle karakteristika, som bl.a. variablerne *Name og* description samt metoden landOn. Dette sikre at alle felter er ensartet så de kan behandles ens.

Generelt om arv

Det kan være smart at generalisere klasser så man kan nedarve og dermed genbruge større dele af koden og senere specialisere de enkelte dele hvorved subklassen kan adskille sig. Skal man bruge metoder og data fra en klasse, kan man i stedet for at kopiere koden fra en klasse til andre klasser, altså bruge nedarvning.
På denne måde kan man hente alle metoder og variabler fra en såkaldt superklasse ved at bruge extends og superklassenavnet.
Det gør det også langt nemmere at vedligeholde én kopi af en metode i stedet for at skulle rette den samme metode i flere klasser. Man kan desuden overskrive og tilføje nye metoder i subklasserne, samt tilføje nye variabler. Subklasserne er altså en specialiseret udgave at superklassen.
Ønsker man at bruge default metoden, kan man bruge super i underklassen. Det er vigtigt man bruger public som acces type i superklassen, eller protected for at opretholde indkapsling, da subklasserne ellers ikke kan tilgå metoderne. Man kan stadig godt have private variabler, men så skal man bruge en get metode for at kunne bruge dem i subklasserne. Access type "protected" indkapsler ikke helt som private, men gør at det kun er klasser i den pågældende package der har adgang, samt subklasser.
Konstruktøren nedarves ikke, men man kan tilgå metoden og variablerne ved at bruge super. Bruger man final, kan metoden ikke overskrives. Bruger man abstract, kan man ikke oprette objekter af denne klasse. Abstract markerer man med kursiv når man skriver den ind i et klassediagram.

Polymorfisk landOn metode

Field er gjort abstrakt da det i vores matadorspil ikke giver mening at have et felt som ikke tilhører en type, fordi dette felt ikke vil have en adfærd.
Alle felter har en LandOn metode via en abstrakt *LandOn* metode i klassen field. *LandOn* eksekveres når en spiller lander på et felt, og varierer alt efter hvilket felt spilleren lander på.

Denne variation i adfærd opnås via metoden *landOn,* som er en polymorfisk metode. Dvs. at de specialiserede subklasser overskriver *landOn* metoden med deres egen variant og derved ændrer adfærden samtidig med at den specialiseret felttype blot kan behandles som superklassen, nemlig som et felt.
Denne tilgang sikrer at vi i GameControlleren kan kalde *landOn* på hvilket som helst felt i vores Feltliste array, selvom vi kun tager højde for typen af felt.
Derudover gør denne tilgang det også nemt at udvide spillet med flere felttyper uden at skulle ændre i eksisterende kode.
Dette er også grunden til at samme overordnet struktur også er blevet brugt til at lave chancekort.
Ligesom i Field er der en superklasse som definerer de overordnet karaktertræk. Igen anvender vi polymorfi og nedarvning til at håndtere handlingen som skal ske når kortet bliver trukket op af bunken.

Arkitektur til optimering af testability med Out klassen.

Vi har ønsket at designe vores spil så det er muligt at teste med automatiseret test. For at opnå dette, har vi været nødt til at finde en måde at komme uden om GUI'en på, og erstatte det som GUI'en ellers normalt give af input med forudbestemte værdier.

Den løsning vi fandt frem til, er opbygget således at vi har en Out klasse øverst i et arve hierarki, Out og dens Subklasser der derved agerer som mellemlag mellem GUI'en og controllerne for således at kunne omdirigere til noget andet end GUI'en.

Vi anvender out som en interface. Dog har vi ikke lært om interfaces endnu og derfor er det ikke, til fulde, implementeret som en interface. Under out er subklasserne Output, FakeOutputTrue og FakeOutputFalse. Output er den klasse som vi bruger under det rigtige spil, hvor vi ønsker at spillet kommunikerer med GUI'en og viser spillerne hvad der sker visuelt på skærmen. Under testing ønsker vi ikke at skulle svare på spørgsmål undervejs i testen, da det så ikke længere vil være en automatiseret test. Det at have GUI'en med under test kan give uhensigtsmæssige fejl og sløver test processen væsentligt, hvorfor det var vigtigt for os at få et system, hvor det var muligt at komme uden om GUI'en. Det vil i praksis sige at når vi i fx. unit testing laver et testscenarie på en af vores logiske enheder, kan man nu komme helt udenom GUI'en.

Det formår vi ved at have to konstruktører. En uden parametre, og en som tager imod et Out objekt. Den konstruktør som ingen parameter tager kalder den anden konstruktør ved hjælp at

THIS, referencen til sig selv og sikrer at hvis konstruktøren uden parametre kaldes, vil spillet få et "normalt" Output objekt og dermed fungere med GUI'en.

```
public Gamecontroller()
{
    this(new Output()); //Kalder kontruktøren nedenunder.
}

public Gamecontroller(Out out) //Ved test bruges kun denne konstruktør og ikke den ovenstående.
{
    this.out = out; //
    new Fieldlist(out);
    new Deck(out);
}
```

Under test anvender vi i stedet den anden konstruktør som tager imod Out objektet, og alt efter hvilke svar vi ønsker tilbage til vores test, vælges *FakeOutputTrue* eller en *FakeOutputFalse.*
Disse to klasser dækker ikke over alle test scenarier, da ikke alle GUI'ens metoder returnerer True eller False. For at dække Fx. *prisonAction* metoden ville det være nødvendigt med en "*FakeOutput*" klasse mere, men med denne tilgang er det relativt nemt at skive en ny subklasse til og derved kan man håndtere mere avanceret svar.

Køb og salg af huse

I slutningen af en tur vil spillet tjekke om du har alle grunde i en farve kategori og i så fald give dig mulighed for at bygge eller sælge huse på de grunde som tilhører den farve kategori du ejer.
Det er implementeret med metoden *checkPlots.*

```
public void checkPlots(Player p, Out out){

    int[] gruppeNumre = GenereBuyedFieldArray(p); // udfylder gruppenumre med data om havd spilleren ejer
    List<Field> flist = addPlotsToList(gruppeNumre); // tjekker om spiller ejer alle grunde i en gruppe, og giver tilbage


    if(flist.size() < 1){ // vis der ikke er nogle grunde i flist - STOP her.
        return;
    }

    if(out.shopOrNot()){
        if(out.sellOrBuy()){

            String result = out.whereToBuild(GenereNameArray(flist)); // spørg i GUI

            int index = findeIndexofField(result); // finder index for placering af field der skal bygges på.

            buyhouse(p,result,index);

        }
        else{

            String result = out.whereToSell(GenereNameArraySell(flist));

            int index = findeIndexofField(result);

            sellhouse(p,result,index);

        }
    }

}
```

CheckPlots har en række private hjælpe metoder. Den første der bliver anvendt er *GenereBuyedFieldArray* som kører alle felter fra feltlisten igennem med et for-loop. Herefter tjekker vi med en if sætning hvert enkelt felt om det er en instans af en grund (af typen Plot). Hvis det er en grund, caster vi det til typen Plot, for at kunne tilgå den Plot-klassens *getOwner* metode og tjekker i et "nested if" om der overhovedet er en ejer, og hvis der er en ejer om det så er spilleren selv, som i så fald får mulighed for at bygge på sin grund.

Hvis alle disse betingelser er sande, vil der i Arrayet gruppeNumre blive lagt én til på den plads der tilhører den pågældende grunds farve. I arrayet er farven repræsenteret som et index, altså et heltal. Referencen til arrayet bliver returneret til variablen gruppeNumre.

Arrayet bliver dernæst brugt som parameter når der kaldes til addPlotsToList. Dette kunne sagtens have været lavet uden en mellem-variabel, men for overskuelighedens skyld er det gjort således.

addPlotsToList tjekker det array som *GenereBuyedFieldArray* har genereret, op mod et andet array med navnet *maxPlots* med et for-loop. maxPlots indeholder det max antal plots der er i hver gruppe. Hvis der er et match mellem det der er i maxPlots og gruppeNumre arrayerne på samme index-plads, manipuleres den data i arrayet canBuild boolean arrayet på det specifikke index til "true".

Dernæst anvendes endnu et for-loop til at køre gennem canBuild boolean arrayet. Inde i for-loopet er en if sætning som sørger for at hver gang den støder på et true *canBuild* tilføjer den, den gruppe grunde til en arraylist kaldt "flist". Efter eksekveringen af for-loopet returnerer metoden "flist" arraylisten og referencen bliver givet til variablen (lokal flist) som eksisterer inden for *CheckPlots* scoopet.

Tilbage i *CheckPlots* tjekker en if sætning om listens størrelse er mindre end 1. Hvis dette er sandt, returneres der ud af metoden og eksekveringen stoppes.

Ellers får den spiller først mulighed for at vælge om han vil handle eller ej, fremstillet grafisk i GUI'en. Hvis ja, bliver der dernæst spurgt om hvorvidt man vil købe eller sælge et hus. Vi har ikke valgt at implementere at man kan købe eller sælge flere huse på en gang, grundet tidsnød.

Vi anvender GenereNameArray() som parameter til kaldet til GUI'en gennem out klassen. I *GenereNameArray* Initialiserer og tildeler vi et array af strings med det antal pladser som svarer til *flist* størrelse. Dernæst kører vi gennem flist med et for-loop for at få hvert enkelt feltobjekts navn med getName metoden. Hvert af disse navne bliver tildelt en plads i String arrayet *FieldNames*. Til sidst returneres dette String array til out klassens metode *wereToBuild* som i GUI'en via en drop-down menu viser arrayet, så det er muligt at vælge et felt udfra navnet hvorpå man ønsker at bygge det pågældende hus.

```
private String[] GenereNameArray(List<Field> flist){

    String[] FieldNames = new String[flist.size()]; // opretter array.


    for (int i = 0; i < flist.size(); i++) { // sætter navne på felter ind i Navne-array
        FieldNames[i] = flist.get(i).getName();
    }
    return FieldNames;
}

public List<Field> getGroup(int x){
    List<Field> flist = new ArrayList<>();
    for (Field f : Fieldlist.getFields()) {
        if(f instanceof Plot ){
            Plot p = (Plot) f;
            if(p.getGroupNumber() == x){
                flist.add(p);
            }
        }
    }
    return flist;
}
```

Det valg spilleren foretager i GUI'en bliver returneret som et String objekt og gemt i den lokale variabel "result". findeIndexofField bliver dernæst kaldt med resultat som parameter. findeIndexofField finder, som navnet antyder, indexet på det field som brugeren har valgt i GUI'en ud fra navnet.

```
private int findeIndexofField(String result){
    for (int i = 0; i < Fieldlist.getFields().length; i++) {
        if(Fieldlist.getFields()[i].getName().equals(result)) {
            return i;
        }
    }
    return 0;
}
```

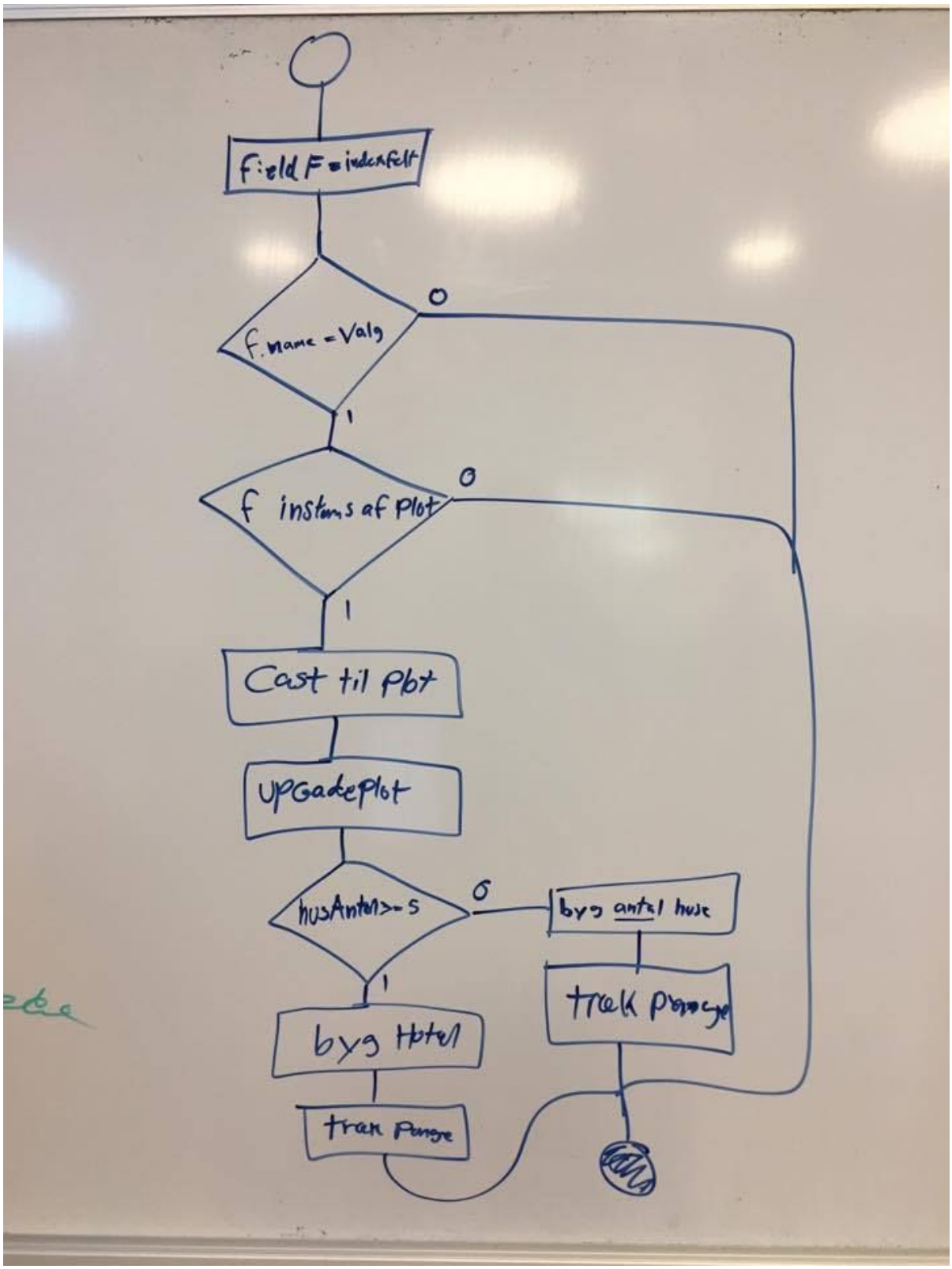
I praksis er det igen et for-loop med et "nested if". For-loopet kører hele feltlisten igennem indtil feltet er fundet. Hvis intet felt er fundet, returnerer den 0. Dette er i vores logiske indeksering start feltet og da der kun kan bygges på grunde (Plot), vil intet ske. Der kan

argumenteres for det kunne have været smart med en fejlmeddelelse. Vi afgører hvorvidt det er det rigtige felt ud fra det pågældende felt med index i's attribut *Name,* som bliver undersøgt vha. objektets standard Equals metode med parameteren result, som var Navnet (en String) på det valgte felt. Når matchet er fundet, returneres i og derved indexet på feltet tilbage til hovedmetoden.

```
private void buyHouse(Player p, String result, int index){

        Field f = Fieldlist.getFields()[index];
        if(f.getName().equals(result)){
            if(f instanceof Plot ){
                Plot plotCast =(Plot) f;
                plotCast.upgradePlot();
                if(plotCast.getHousecount() >= 5){ // bygger hotel vis der er 5 huse.
                    out.BuildHotel(index+1, true);
                    p.getAccount().withdraw(200);
                }
                else{
                    out.BuildHouse(index+1, plotCast.getHousecount()); // bygger huse vis der er mindre end 5 huse
                    p.getAccount().withdraw(200);
                }

            }

        }
}
```

I *buyHouse* metoden tager vi imod tre parametre, nemlig en Player, en String med navnet på feltet der skal bygges på og index hvorpå feltet er. Det første vi gør er at tildele f variablen referencen til vores specifikke felt. Herefter tjekker vi om der er overensstemmelse mellem navn og valget en gang til, så vi sikrer det er det rigtige felt. Så bruger vi et if-statement til at undersøge om feltet er af typen Plot, så vi er sikre på at kunne Caste sikkert. Herefter kan vi få fat på Plot objektets egne metoder og bruger det til at kalde *upgratePlot* som dernæst tjekker at int variablen house count, som tilhører det givne Plot, ikke bliver mindre end fem.

Metoden splitter sig herefter op i to forgreninger. En hvor *houseCount* er større eller lig med fem, til at bygge et hotel og en for at bygge normale huse. Som det kan ses er "index" plusset med én i begge metode kald til GUI'en gennem out. Dette skyldes at vores feltliste er 0 indekseret og GUI'ens er 1 indekseret. Dvs. de starter på henholdsvis 0 og 1.



Ovenstående viser et flowchart over buyHouse-metoden.

Overordnet struktur af Gamecontroller

Gamecontrolleren er den primære controller i vores spil. Det er gamecontrollerens opgave at styre programmets gang og begrænse spillets logik i resten programmet til en klasse. Dette afsnit beskriver strukturen af Gamecontrolleren og dens fire vigtigste metoder.

Programmet starter i main. *main()* metodens eneste ansvar er at starte programmet. Derfor opretter den kun et nyt objekt af *Gamecontroller* klassen og kalder metoden *setup()*.

setup skal gøre spillet klar til at køre. Først oprettes et nyt *player* array objekt vha. *addPlayer()* metoden som laver og returnerer et array af spillere (2 - 6) objekter. Derefter kaldes *runGame()* metoden.

Det er *runGame()*'s opgave at køre selve spillet indtil der er fundet en vinder og tildele hver spiller sin tur. Dette foregår i et *while()-loop* der kører indtil *endGame()* er sandt (se afsnittet om *winner()* metoden). I *while* kører et *for-loop* hver spiller igennem og tjekker vha. et *if* om spilleren er gået bankerot. Hvis han er bankerot springes denne over og loopet kører den næste i arrayet indtil der findes en spiller der ikke er bankerot og tildeler spilleren sin tur ved at kalde *turn()* metoden, som kører den aktuelle spillers individuelle tur.

En turs gang foregår i metoden *turn()*. Metoden består af nogle *if-* og *else- statements* og *for-loops*. Det første *if* tjekker om spilleren er i fængsel. Hvis dette er sandt, kaldes *prisonAction()* metoden så spilleren får "en fængselstur" i stedet for en almindelig tur. Ellers (*else*) køres en normal tur. Her kaldes en række metoder i henhold til turens gang - som vist i design sekvensdiagrammet "spil en tur" (bilag 3)
Når det er gjort, kører et *for-loop* alle spillerne i player arrayet igennem og opdaterer hver spillers balance i GUI'en.

Afslutningsvis kaldes metoderne i rækkefølgen *goBankrupt(), winner()* og *checkPlots()* - der alle er beskrevet i de nedenstående afsnit. Til sidst tjekker et *if-statement* om der blev slået to ens. Hvis der blev slået to ens, kører *turn()* igen for den samme spiller. Ellers er *turn()* færdig og *runGame()* repeteres så den næste spiller i arrayet får sin tur.

Om winner metoden

Overordnet tjekker winner metoden hvor mange spillere der stadig er "i live". Hvis der kun er én spiller tilbage, vil den tjekke og give besked om hvilken, hvorefter metoden også vil afslutte spillet.

Metoden består af nogle *for* loops og *if* statements. I det første *for*-loop tjekkes hver spillers "bankrupt" status om vedkommende er bankrupt eller ej. Dette gøres vha. if-statementet !Bankrupt. Hvis det er tilfældet (at spillerens status er *false*) lægges én til playersAlive count og loopet går videre til næste spiller i arrayet - ellers går loopet direkte videre til næste spiller.

Når *for-loopet* er færdig med at tælle, tjekkes der i et nyt *if-statement* om der kun er én spiller i live. Hvis der kun er en spiller i live, tjekker et *for-loop* vha. af et *if-statement* hvilken spiller der er tilbage hvorpå der vises en besked i GUI. Herefter lukkes GUI'en og spillet slutter.

Om goBankrupt

Det er goBankrupt metodens opgave i slutningen af hver spillers tur at tjekke om spilleren er gået bankerot og hvis det er tilfældet, at sætte spilleren bankerot og vha. hjælpemetoden resetOwnedFields() der nulstiller alle de grunde der er ejet af den spiller der er gået bankerot, så de kan købes påny af de tilbageværende spillere.

Metoden har et *for-loop* der kører igennem spiller arrayet, tjekker hver spiller vha. et *if-statement* om spilleren som loop'et er nået til har en balance der er mindre eller lig med nul og derfor er gået bankerot. Hvis spilleren er gået bankerot, sættes hans bankruptStatus til true, hans brik fjernes fra spillepladen og hans grunde nulstilles vha. hjælpe metoden *resetOwnedFields()*.

resetOwnedFields() tjekker først om den aktuelle spiller p's *bankruptStatus* er *true* vha. en *get-metode*. Hvis det er sandt, køres listen af felter igennem i et *for-loop* hvor det første *if-statement* tjekker om det pågældende felt i fieldlist arrayet er en instance af Ownable. Hvis feltet er en instance of Ownable, tjekker det næste *if-statement* om ejeren af det felt er den samme spiller som p. Hvis det er p's felt, sættes ejer med en set-metode til at være null og kan dermed frit på ny (ejeren fjernes samtidig fra feltet i GUI'en). Således køres alle *fields* igennem i *Fieldlist* på samme måde.

Test

Gennem softwareudviklingen har vi taget udgangspunkt i agile developments tilgang med iterationer som udgangspunkt i UP modellen, derfor har det været smart at have fokus på Test Driven Development, som et led i risiko formindskelses process.

Dette er i praksis gjort ved at teste hver enkelt klasse og dens tilhørende metoder via en dedikeret JUnit test til hver klasse. Alle JUnit testene er samlet i et test suite. Testene findes i test package.

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

Vores test dækker følgende:

- JUnit - af alle logiske enheder, klasser test af alle enheder (klasser)
- Integration - tester det samlede system
- White box test - debug på koden en linje af gangen.
- Black box test - user test af GUI (inkl. positiv og negativ test)

JUnit test

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

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

Dokumentation for alle JUnit test kan findes for hver enkelt specifik JUnit i kildekoden (i test-package).

Unit Testing af fleet

vi har anvendt JUnit som framework til at teste vores klasser i spillet. Vi har lavet en JUnit test på felttypen fleet. Måde vi anvender JUnit på er at oprette nogle spillere og lade dem lande på det specifikke felt, som i dette tilfælde er en Fleet. I JUnit testen kan vi opstille konkrete test scenarier som sørger for at teste de forskellige muligheder der kan forekommer når en spiller lander på feltet. Derudover kan vi tjekke at vores klasse fungere som forventet. Dette indbærer test af matematiske udregninger samt generel kodestruktur.

```
@Test
public void test()
{
        new Fieldlist(out);

        Fieldlist.getFields()[5].landOn(bijan, out);
        Fieldlist.getFields()[5].landOn(kasper, out);

        assertEquals(1500-200+25, bijan.getAccount().getSum());
        assertEquals(1500-25, kasper.getAccount().getSum());
        Fieldlist.getFields()[15].landOn(bijan, out);
        Fieldlist.getFields()[15].landOn(kasper, out);
        assertEquals(1500-200*2+75, bijan.getAccount().getSum());          // bijan køber endnu et fleet felt, og kasper lander uheldigvis på samme felt.
        assertEquals(1500-25-50, kasper.getAccount().getSum());   // nu må han betale 1000, da bijan ejer 2 fleets

        Fieldlist.getFields()[25].landOn(bijan, out);
        Fieldlist.getFields()[25].landOn(kasper, out);
        assertEquals(1500-200*3+175, bijan.getAccount().getSum());         // det samme bro, bare nu med 3 fleets
        assertEquals(1500-25-50-100, kasper.getAccount().getSum());       // betaler 2000

        Fieldlist.getFields()[35].landOn(bijan, out);
        Fieldlist.getFields()[35].landOn(kasper, out);
        assertEquals(1500-200*4+375, bijan.getAccount().getSum());         // det samme bro, bare nu med 3 fleets
        assertEquals(1500-25-50-100-200, kasper.getAccount().getSum());
}
```

Som det ses i koden ovenfor, lander en spiller på et felt af typen fleet hvor efter han køber dette felt. Derefter lander en anden spiller, på samme felt og bliver nødt til at betale ejeren leje. Vi bruger mere specifikt men metode der kaldes assertEquals som sammenligner

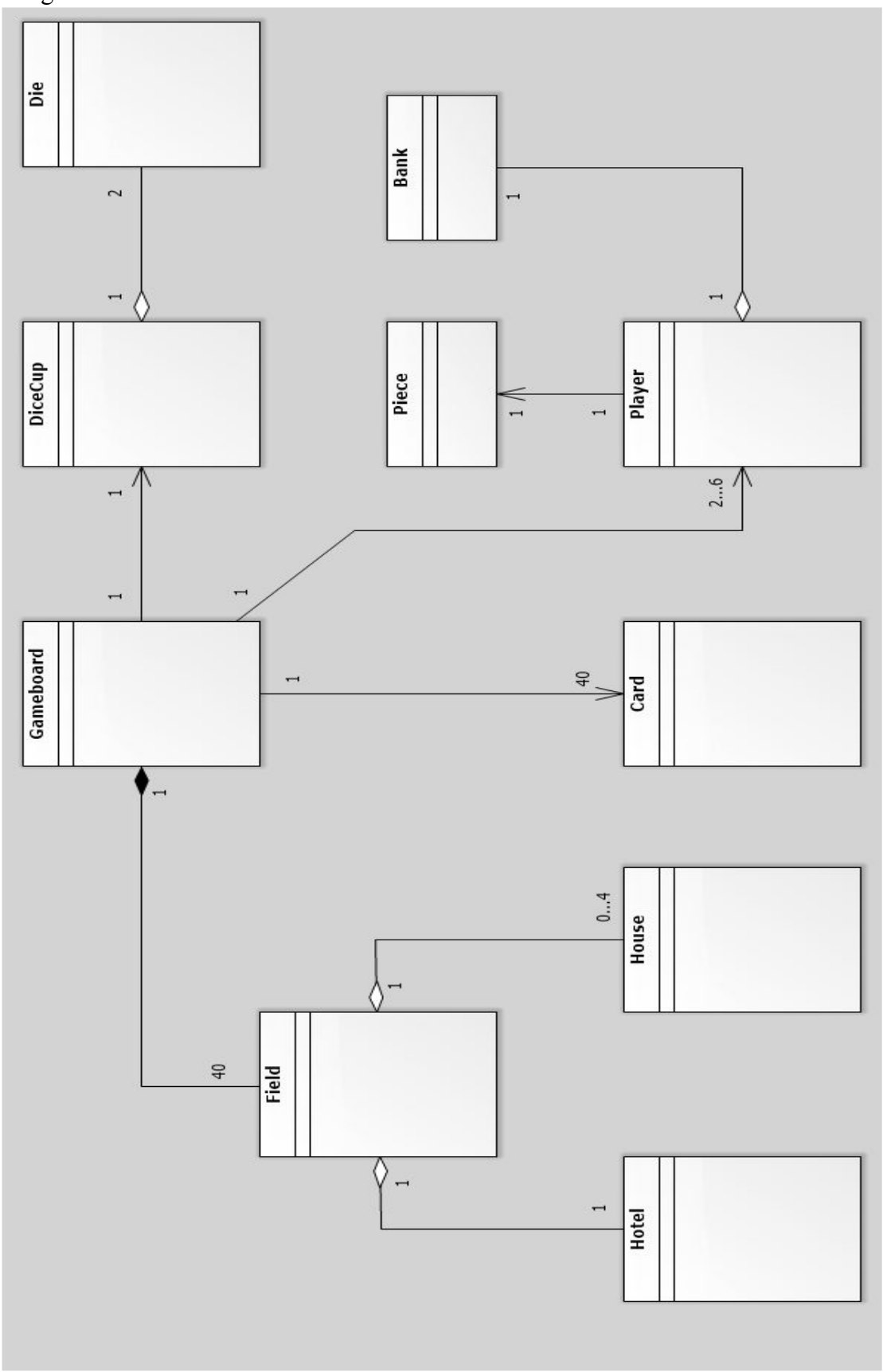
resultaterne og udgør en boolean ud fra hvorvidt det forventet resultat og det enkelt lige resultat er ens. Dette scenarie gentages indtil en ejer kommer til at eje alle fire fleet felter, og en anden spiller lander på feltet og betaler leje.

Konklusion

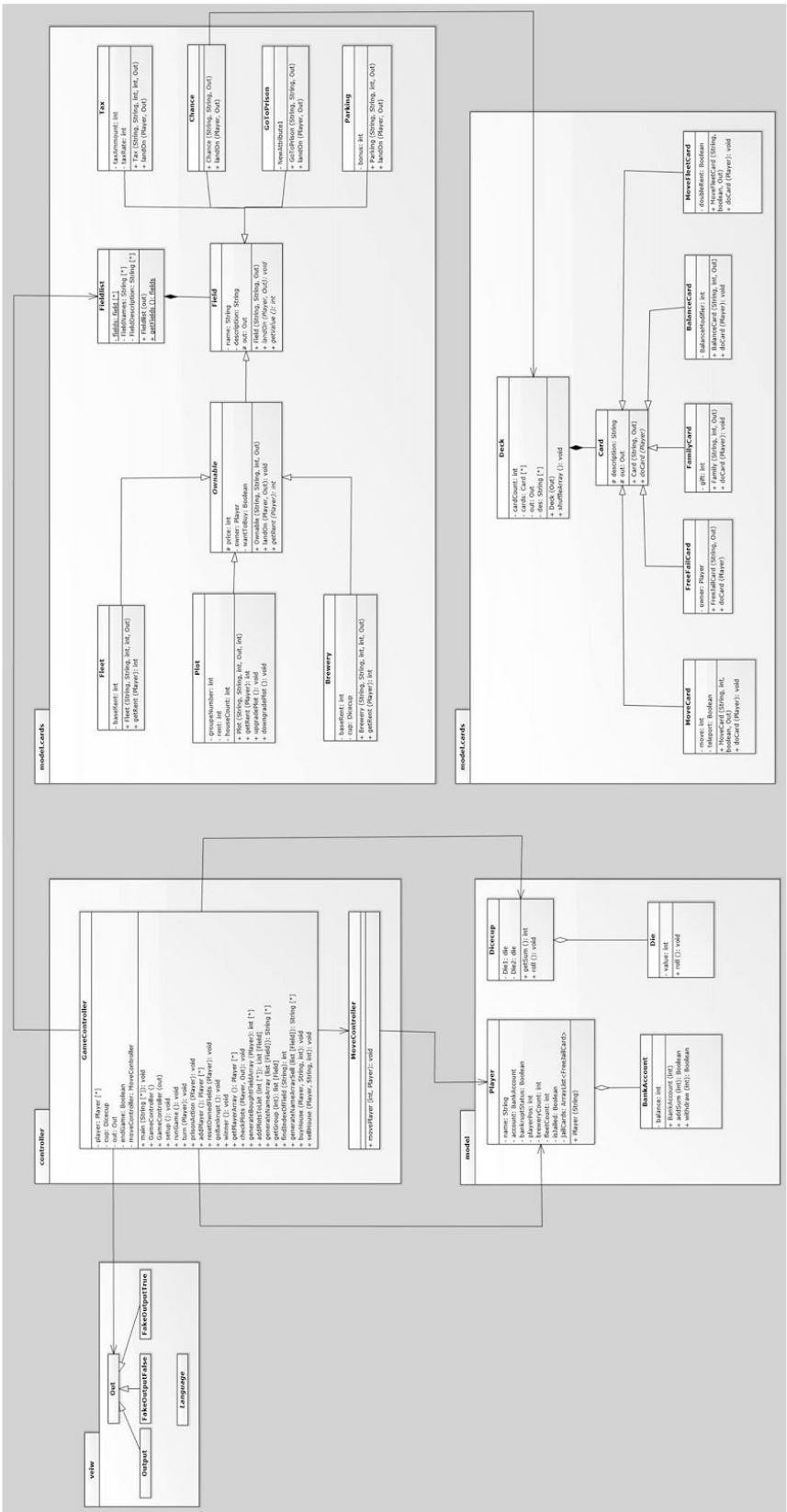
Vi har udviklet at program der simulere matadorspil. Det lykkedes at implementere de fleste spil elementer og regler fra det fysiske spil i vores computerspil. Vores udvikling er gået ud fra en prioriteret rækkefølge af feature-listen for at sikre de vigtigste dele - vha. MoSCoW modellen (se bilag). Vi har implementeret alle features under *Must have*, *Should have* og nogle features fra *Would have*. De resterende features er ikke blevet implementeret. Alt i alt et velfungerende program.

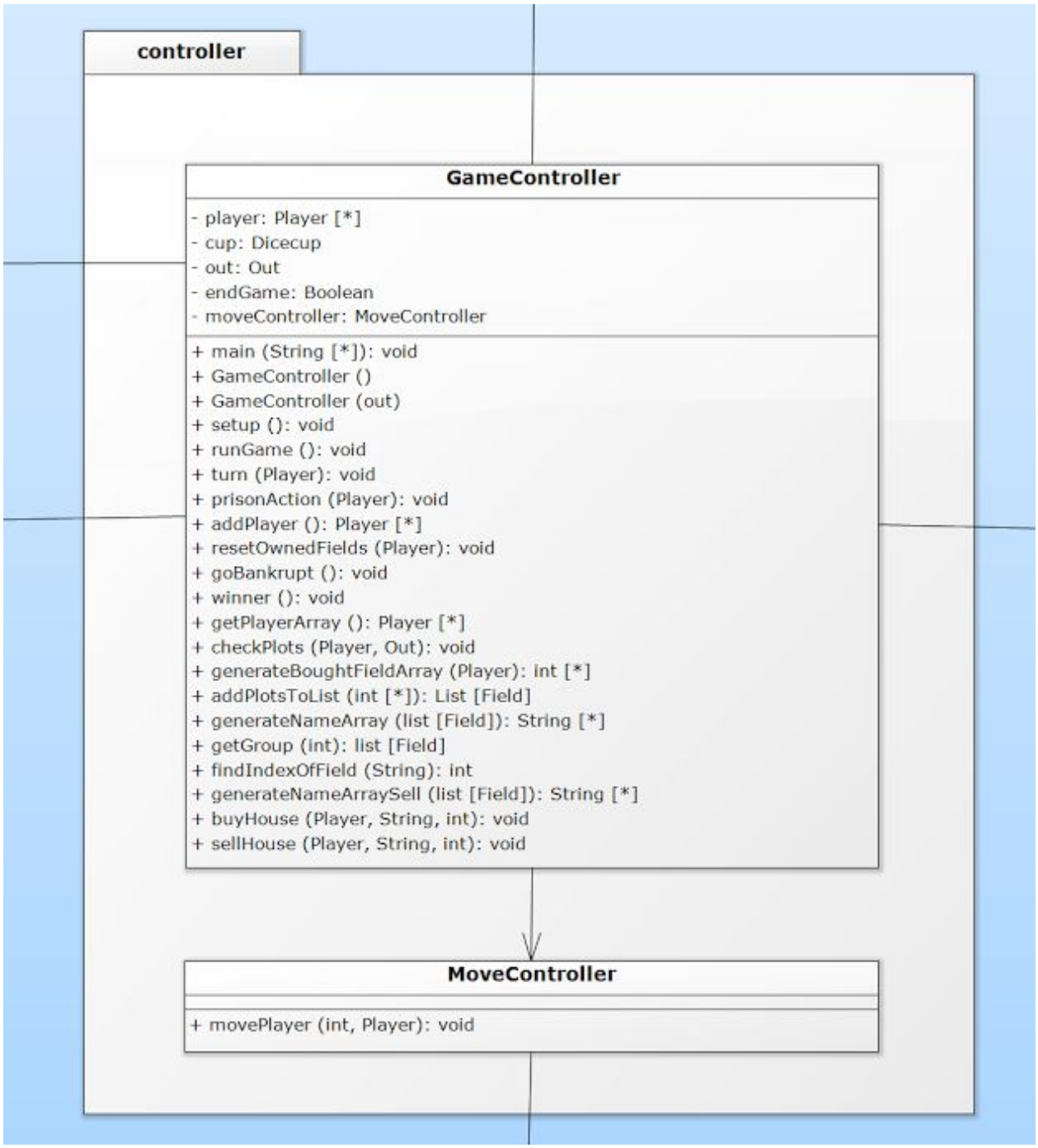
Vi har opfyldt kravspecifikationen til en vis grad. Programmet er test og kan køre på DTU's maskiner i DTU databar. Vi har lavet de nødvendige analyse- og design modeller og ligeledes dokumenteret programmet.

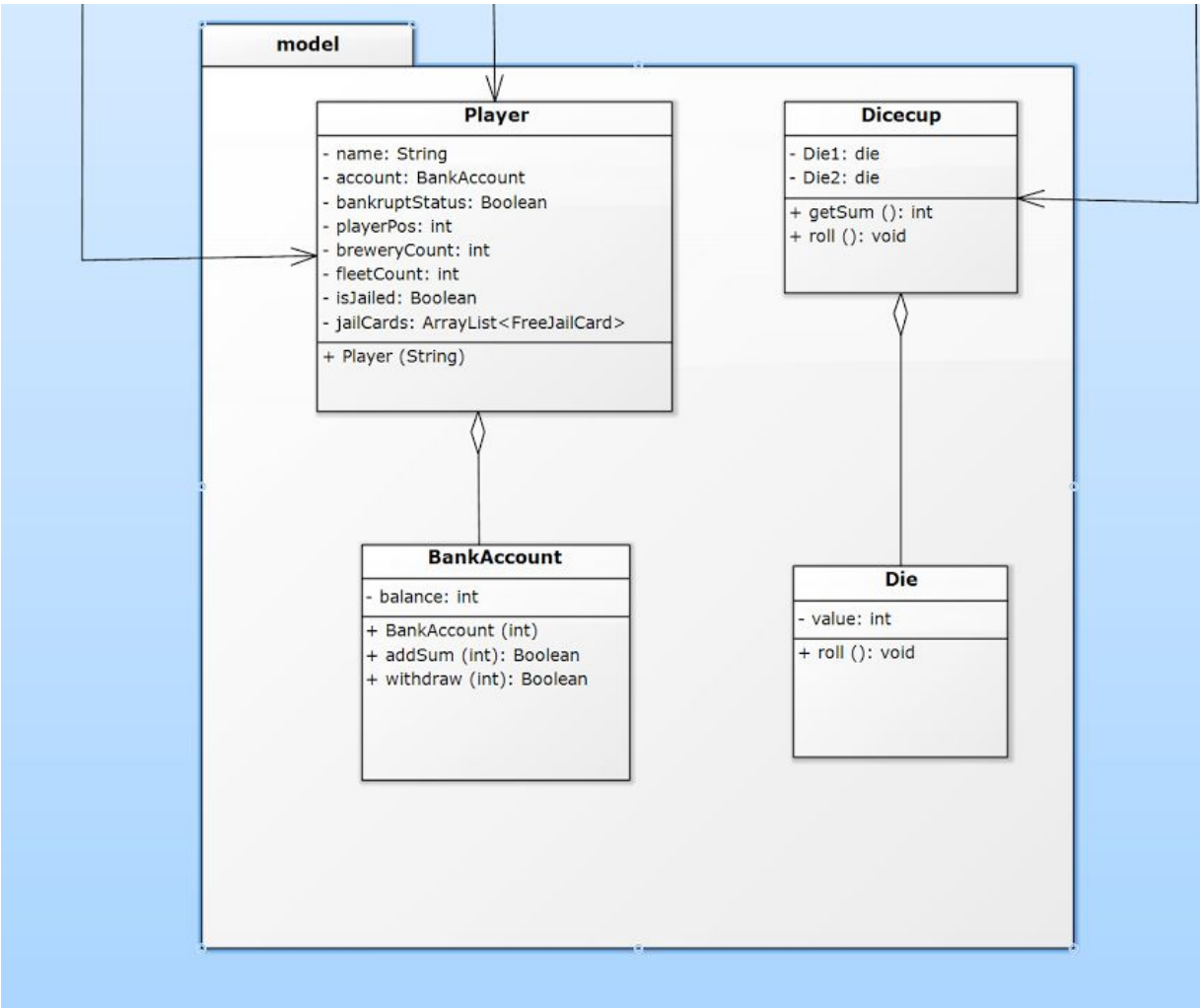
Bilag 1 Domænemodel

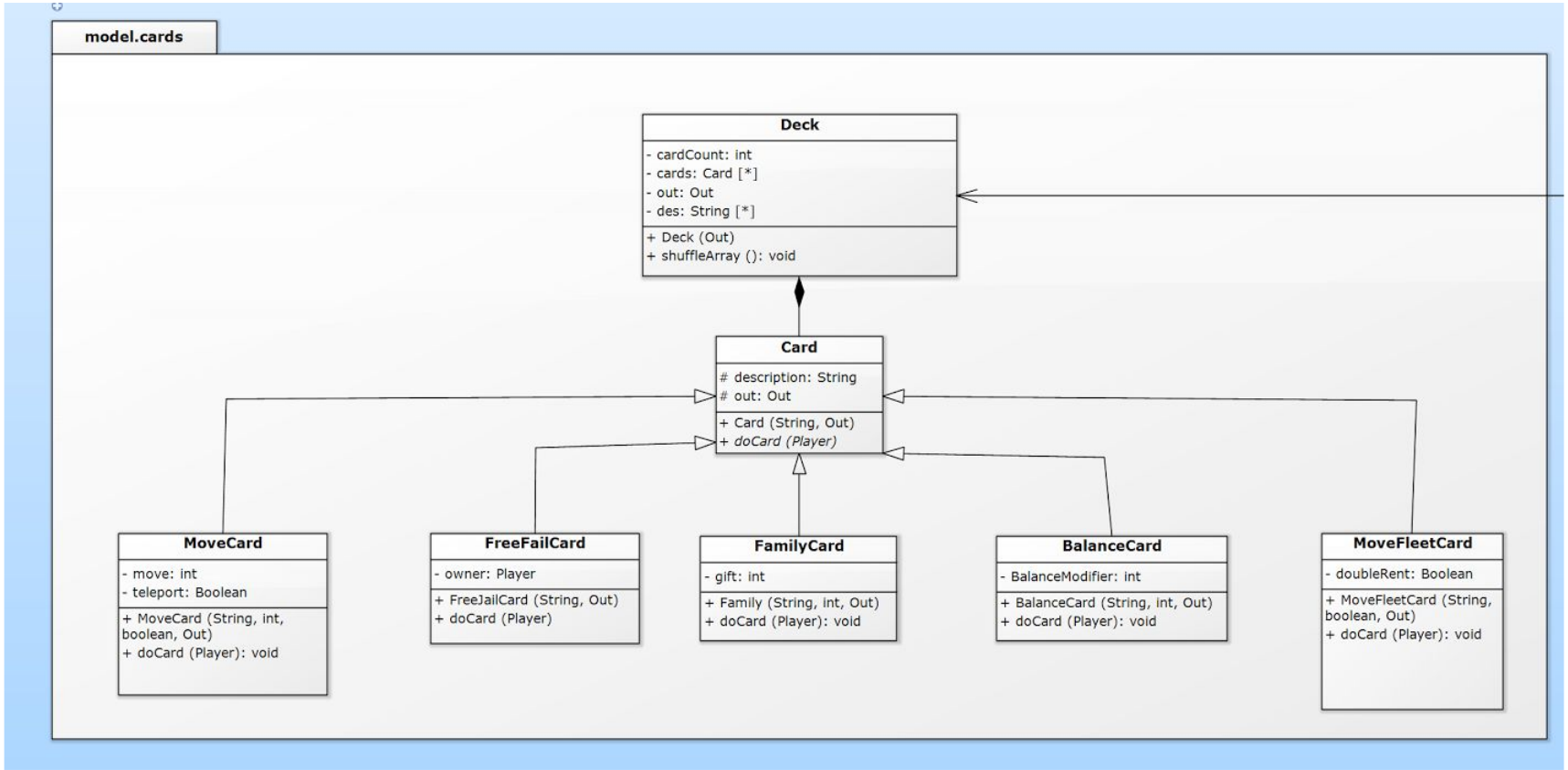


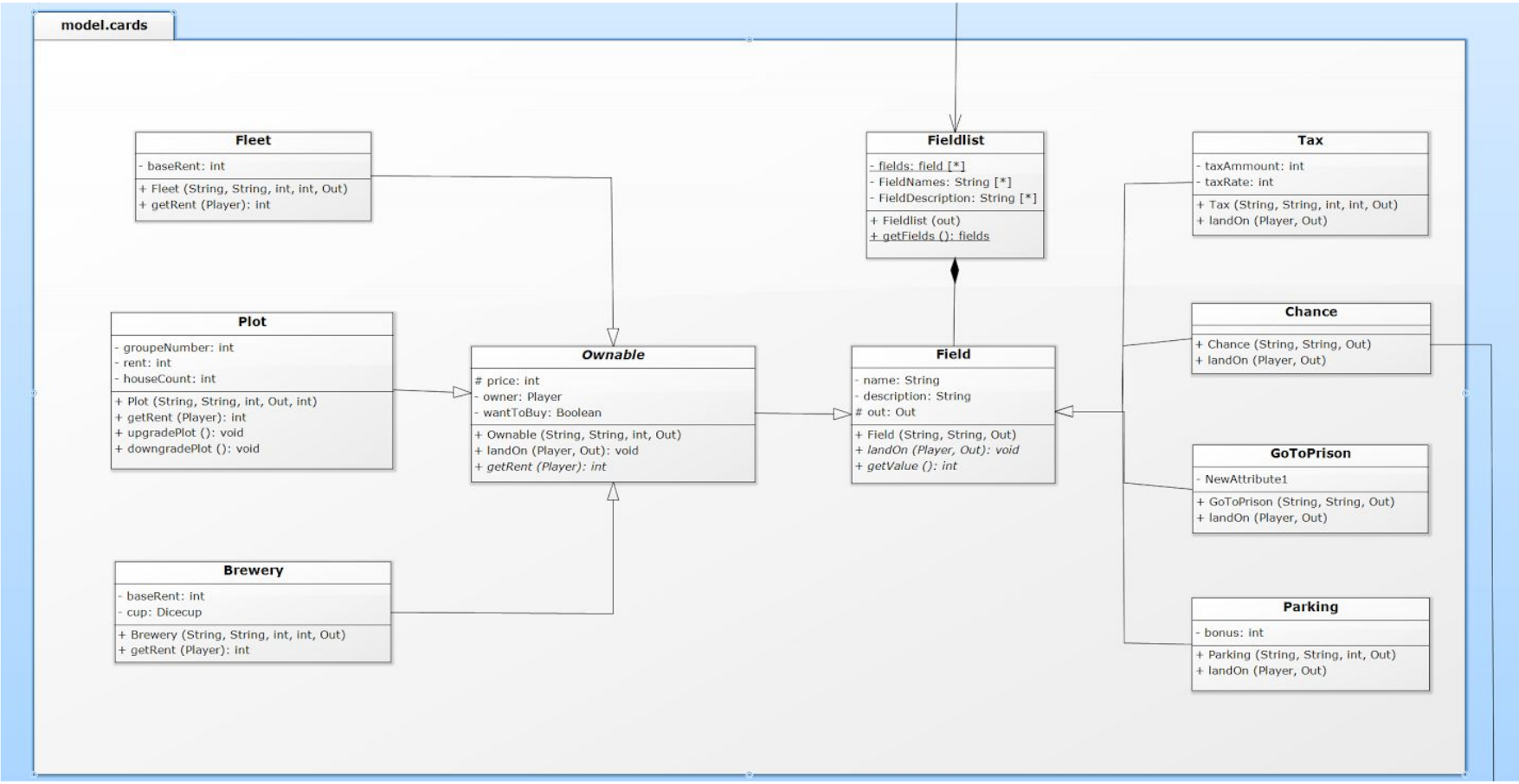
Bilag 2 Klassediagram

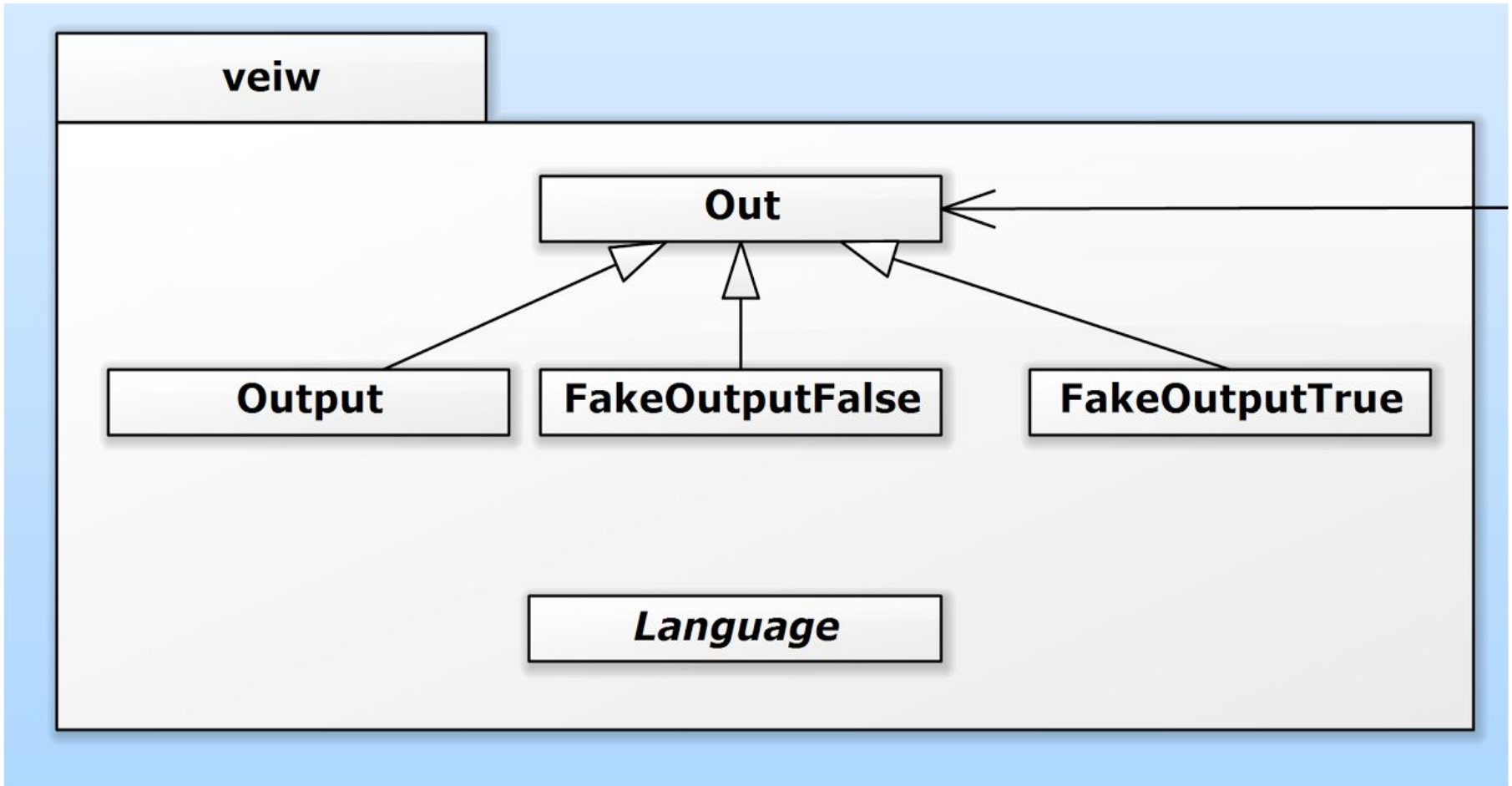



controller
GameController
- player: Player [*]
- cup: Dicecup
- out: Out
- endGame: Boolean
- moveController: MoveController
+ main (String [*]): void
+ GameController ()
+ GameController (out)
+ setup (): void
+ runGame (): void
+ turn (Player): void
+ prisonAction (Player): void
+ addPlayer (): Player [*]
+ resetOwnedFields (Player): void
+ goBankrupt (): void
+ winner (): void
+ getPlayerArray (): Player [*]
+ checkPlots (Player, Out): void
+ generateBoughtFieldArray (Player): int [*]
+ addPlotsToList (int [*]): List [Field]
+ generateNameArray (list [Field]): String [*]
+ getGroup (int): list [Field]
+ findIndexOfField (String): int
+ generateNameArraySell (list [Field]): String [*]
+ buyHouse (Player, String, int): void
+ sellHouse (Player, String, int): void
MoveController
+ movePlayer (int, Player): void

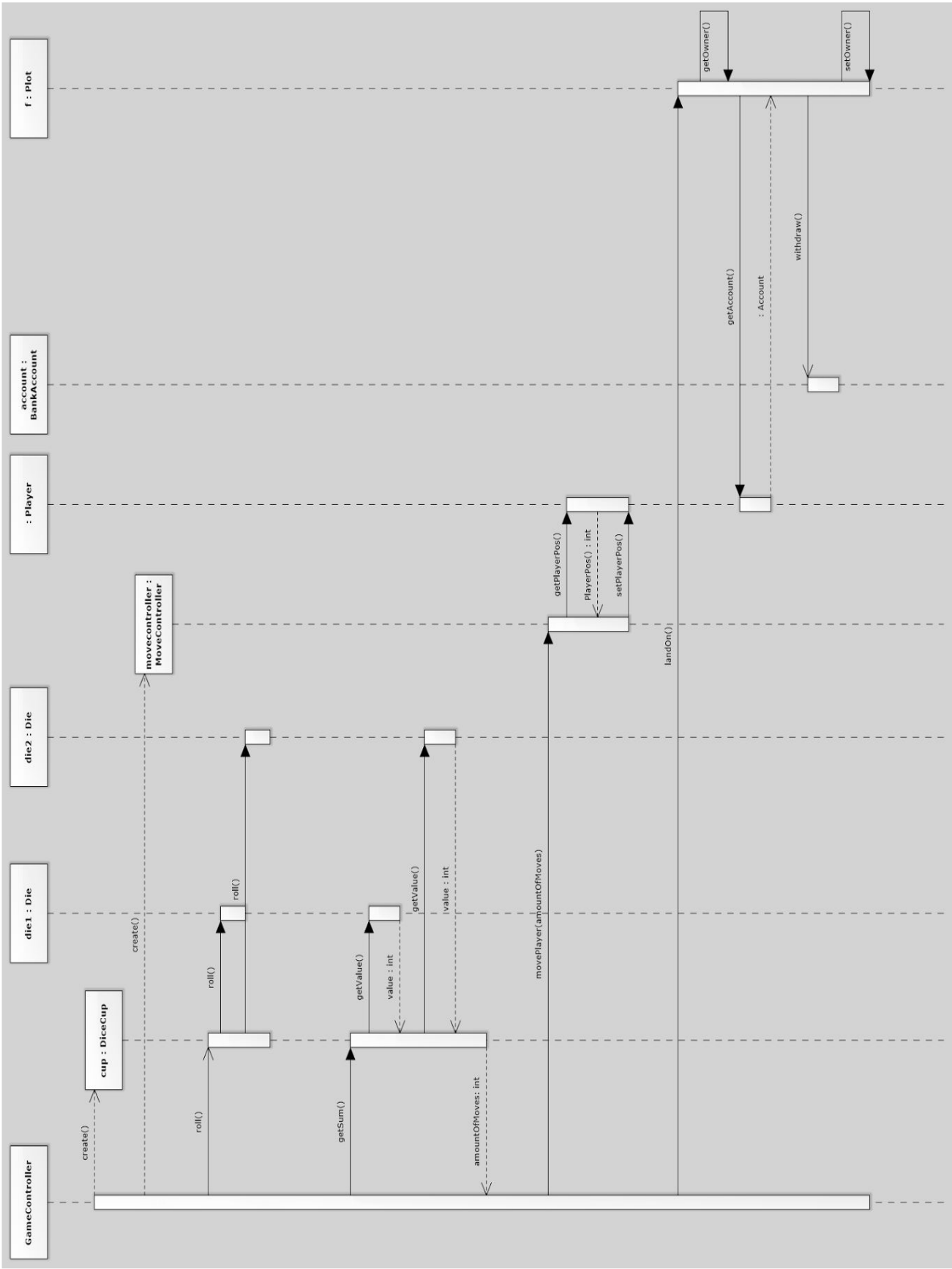

model
Player
- name: String
- account: BankAccount
- bankruptStatus: Boolean
- playerPos: int
- breweryCount: int
- fleetCount: int
- isJailed: Boolean
- jailCards: ArrayList<FreeJailCard>
+ Player (String)
Dicecup
- Die1: die
- Die2: die
+ getSum (): int
+ roll (): void
BankAccount
- balance: int
+ BankAccount (int)
+ addSum (int): Boolean
+ withdraw (int): Boolean
Die
- value: int
+ roll (): void


model.cards
Deck
- cardCount: int
- cards: Card [*]
- out: Out
- des: String [*]
+ Deck (Out)
+ shuffleArray (): void
Card
description: String
out: Out
+ Card (String, Out)
+ doCard (Player)
MoveCard
- move: int
- teleport: Boolean
+ MoveCard (String, int, boolean, Out)
+ doCard (Player): void
FreeFailCard
- owner: Player
+ FreeJailCard (String, Out)
+ doCard (Player)
FamilyCard
- gift: int
+ Family (String, int, Out)
+ doCard (Player): void
BalanceCard
- BalanceModifier: int
+ BalanceCard (String, int, Out)
+ doCard (Player): void
MoveFleetCard
- doubleRent: Boolean
+ MoveFleetCard (String, boolean, Out)
+ doCard (Player): void

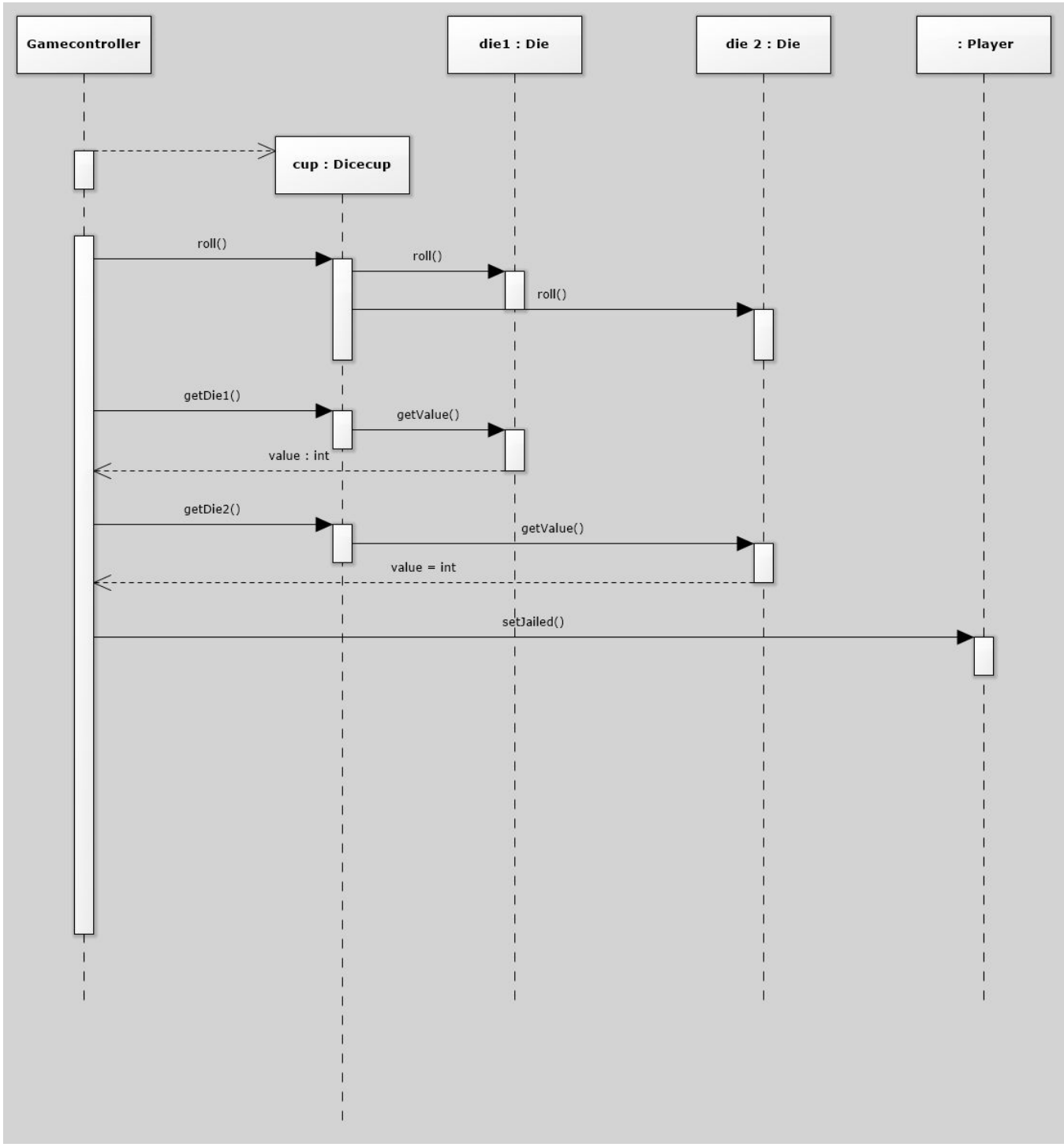

model.cards
Fleet
- baseRent: int
+ Fleet (String, String, int, int, Out)
+ getRent (Player): int
Fieldlist
- fields: field [*]
- FieldNames: String [*]
- FieldDescription: String [*]
+ Fieldlist (out)
+ getFields (): fields
Tax
- taxAmmount: int
- taxRate: int
+ Tax (String, String, int, int, Out)
+ landOn (Player, Out)
Plot
- groupeNumber: int
- rent: int
- houseCount: int
+ Plot (String, String, int, Out, int)
+ getRent (Player): int
+ upgradePlot (): void
+ downgradePlot (): void
Ownable
price: int
- owner: Player
- wantToBuy: Boolean
+ Ownable (String, String, int, Out)
+ landOn (Player, Out): void
+ getRent (Player): int
Field
- name: String
- description: String
out: Out
+ Field (String, String, Out)
+ landOn (Player, Out): void
+ getValue (): int
Chance
+ Chance (String, String, Out)
+ landOn (Player, Out)
GoToPrison
- NewAttribute1
+ GoToPrison (String, String, Out)
+ landOn (Player, Out)
Brewery
- baseRent: int
- cup: Dicecup
+ Brewery (String, String, int, int, Out)
+ getRent (Player): int
Parking
- bonus: int
+ Parking (String, String, int, Out)
+ landOn (Player, Out)


veiw
Out
Output
FakeOutputFalse
FakeOutputTrue
Language

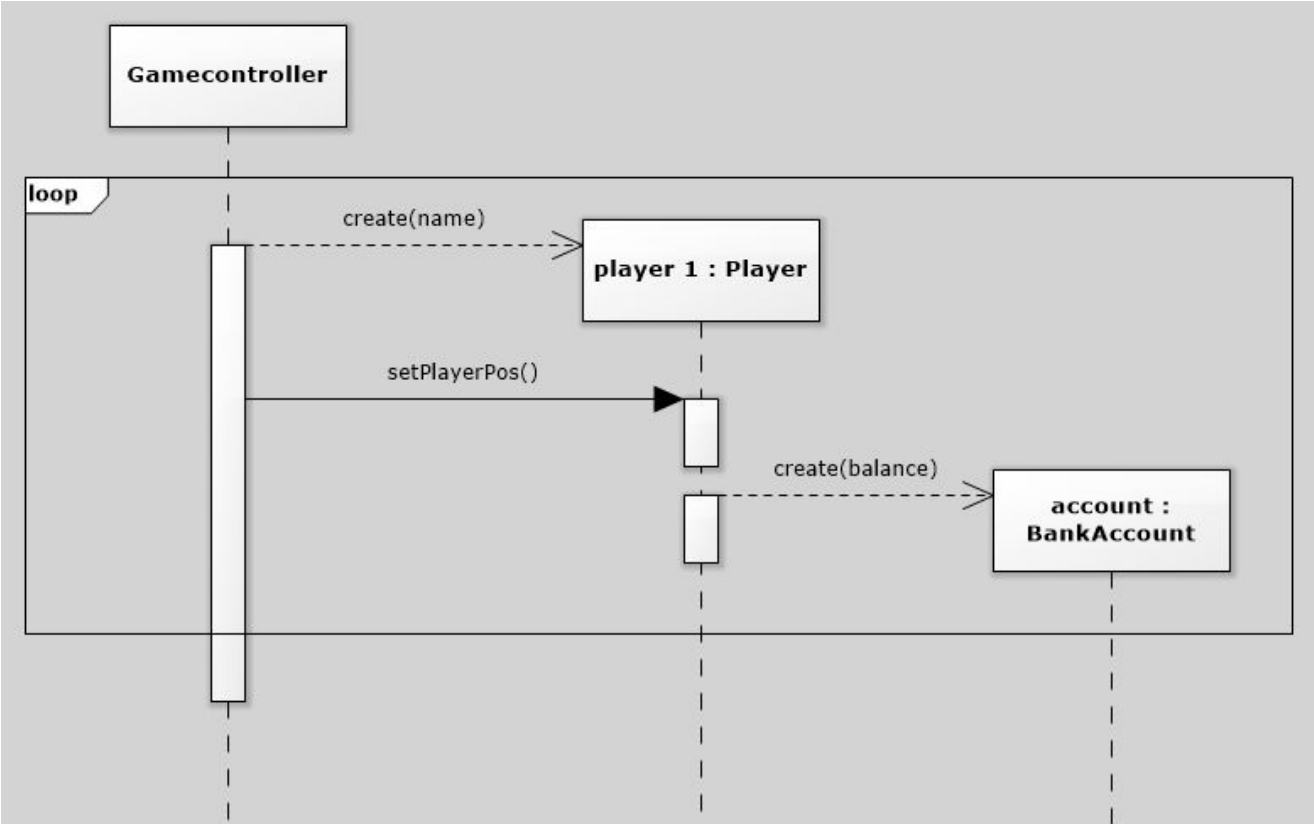
Bilag 3 DSD spil en tur



Bilag 4 DSD spil en fængelstur



Bilag 5 DSD opret spillere



Bilag 6 MoScoW model

Must have

- Der skal være 2-6 spillere
- Man anvender 2 terninger som slår fra 1-6
- Start kapital kr. 30.000
- Der skal være felter
    - 1x Parkering [parking]
        - Kan IKKE købes
    - ?x Grunde [plot]
        - Grunde deles op i farve kategorier
        - Kan købes
        - Leje og pris
        - Leje betales til ejer
        - Ejes (af samme spiller) grunde af samme farve betales der dobbelt leje for ubebygget grunde af den farve
    - 2x Virksomheder [firm]
        - Kan købes
        - Leje og pris
        - Leje betales til ejer
        - Lejen udregnes ved at spilleren slår med terningerne og ganger med antallet af øjne kr. 100.
    - 1x Start [parking]
        - Kan IKKE købes
    - 4x Flåde [fleet]
        - Kan købes
        - Leje og pris
        - Leje betales til ejer
        - Lejen øges af antallet grunde ejet af samme spiller
- Efter bankerot bliver spillerens ejendomme givet tilbage til banken (nulstillet)
- Når man passere start får man en bonus på kr. 4000

Should have

- Huse (maks 4 huse pr grund)
- Hoteller (bygget ud fra 4 huse og beløb)

    - Der skal være 4 huse på grunden
    - Koster 4 huse plus 1 hus
    - Der må kun være et hotel pr. grund
- Leje forøges pr. hus / hotel på grunde
- Priser, leje og pant på "skøde" (GUI)
- Prøv lykken felt (chance kort)
    - Kan IKKE købes
- Ekstra tur når man slår to ens

Would have

- Der skal være følgende felter
    - 1x Fængsel [prison]
    - 01x Fængselscelle [parking]
        - man er på besøg (dvs. der er konsekvenser) når man lander på fængsel feltet (uden at man er blevet sat i fængsel)
        - man kan blive fri på følgende 4 måder
            - betale en bøde på 1000 inden man kaster terningerne
            - benytte løsladelses kort (hvis du har det fra chance kort)
            - slå 2 ens - man bliver fri, rykker antal øjne og får ekstra kast
            - du skal betale bøden efter 3 omgang i fængslet
            - man kan kan deltage i auktioner / handle grund med andre spillere mens man er i fængsel
        - man får ikke leje fra sine grunde mens man er i fængsel

Nice to have

- Pantsætning af ejendomme
    - kun ubebygget grunde kan pantsættes til banken
    -
    - spilleren beholder grunden (der nu er inaktiv) og modtager ikke leje for dette felt
    - du kan købe feltet uanset hvornår (gøre det aktivt) ved at betale "lånet" tilbage plus 10% af lånet f.eks. 500 bliver 550
- Auktioner

- Køb og salg mellem spillere
    - Man kan sælge og købe ubebygget grunde ect. indbyrdes mellem spillerne (hvis man vil sælge en grund der allerede har bygninger skal disse sælges til banken inden grunden kan sælges)
- Man skal fordele sine huse jævnt over sine felter dvs. inden man kan bygge hus nr 2 på samme felt skal der være bygget 1 hus på alle de andre felter i samme gruppe og så fremdeles
- Skylder spilleren mere end han ejer skal spilleren betale det han kan og overdrage alle sine grunde til kreditor efter at spilleren har solgt evt. bygninger til banken. Og derefter udgå af spillet
    - Er banken kreditor sælger bankøren straks modtagne grunde på auktion
- Man kan evt spille spillet på tid eks. 10 min, 15 min, 30 min, 1 time, 2 timer, 1 døgn