```
public class CDIO_final {

    public static void main(String[] args) {
System.out.print("CDIO_final");
    }
}
```

Gruppemedlemmer:

Thomas Mattsson - s175206
Rasmus Gregersen - s175221
Andreas Mørch Secher - s175208
Patrick Marcell Madsen - s175209
Søren Kaare Rasmussen - s175202

Indholdsfortegnelse

Tidsregistrering

CDIO_final							
Time-regnskab	Ver. 2018-01-15						
Dato	Deltager	Design	Impl.	Test	Dok.	Andet	Ialt
2/1-2018	Andreas					2	2
-	Rasmus	1,5				0,5	2
-	Patrick	1			0,5	0,5	2
-	Thomas				1	1	2
-	Søren	0,5			1	0,5	2
3/1-2018	Andreas		2				2
-	Rasmus	2	1		1		4
-	Patrick		3			0,5	3,5
-	Thomas	1	2			1	4
-	Søren		4				4
4/1-2018	Andreas		3,5				3,5
-	Rasmus		3,5			1,5	5
-	Patrick		5				5
-	Thomas	1	4				5
-	Søren		7				7
5/1-2018	Andreas		5				5
-	Rasmus		5				5
-	Patrick		5				5
-	Thomas		3		2		5
-	Søren		5				5
8/1-2018	Andreas		7				7
-	Rasmus		4,5	1,5		1	7
-	Patrick		6	1			7

-	Thomas		7				7
-	Søren		7				7
9/1-2018	Andreas		7				7
-	Rasmus		4	2		1	7
-	Patrick		5,5	0,5		1	7
-	Thomas		4,5	1			5,5
-	Søren		7				7
10/1-2018	Andreas		6				6
-	Rasmus		7				7
-	Patrick		5	1,5			6,5
-	Thomas		4	2	1		7
-	Søren		7				7
11/1-2018	Andreas		6			2	8
-	Rasmus		6	1		1	8
-	Patrick		5,5	1,5			7
-	Thomas		6	2			8
-	Søren		2	2	4		8
12/1-2018	Andreas				4		4
-	Rasmus		2,5	1	0,5		4
-	Patrick		3	1			4
-	Thomas			0,5	4		4,5
-	Søren	3	0,5		1		4,5
13/1-2018	Andreas						0
-	Rasmus		1,5	0,5			2
-	Patrick						0
-	Thomas						0
-	Søren	2			2		4
14/1-2018	Andreas				3		3
-	Rasmus		1,5	2	1,5		5
-	Patrick		4		2	1	7
-	Thomas			5	5		10

-	Søren	4	2		4		10
	Sum	16	187	26	37,5	14,5	281

1 Indledning

1.1 Problemformulering/Case:

Vi skal udvikle et tæt på fuldt implementeret Monopoly spil. Her har vi taget de vigtigste krav til spillereglerne, som er opstillet længere nede i vores kravspecifikation. Størstedelen af spillereglerne er taget fra de udleverede monopoly regler.

Vi har terninger, spillere, spillepladen og felterne på plads fra det forrige projekt, men en del af de essentielle Monopoly funktioner mangler. I dette tredje spil ønskes derfor at forrige del bliver udbygget med forskellige typer af chancekort, samt funktionen hvor man kan købe huse og hoteller på de forskellige grunde. Derudover skal man også kunne pantsætte grunde og handle indbyrdes med de andre spillere på pladen.
Hvor man i sidste iteration af Monopoly spillet blot kunne lande på et felt og rykke videre, skal man altså nu kunne foretage en række valg på felterne, der enten kan købes, byttes pantsættes osv. Spillet skal nu også kunne håndtere op til 2-6 spillere.

1.2 Formål, Værktøjer og Metoder:[1]

Efter udviklingen af det sidste spil erfarede vi, at en mere iterativ proces var os til stor gavn i løbet af projektet, da vi i større omfang kunne tilpasse vores program til de tests vi udførte på de tidlige iterationer. Den proces vil vi holde fast i, til dette projekt. Under hele forløbet vil vi følge en strengt tilrettelagt prioriteringsliste. Listen skal forhåbentlig hjælpe os med at holde styr på hvad der skal være i fokus hele tiden, så ikke vi afviger for meget fra den overordnede plan, og spilder tiden.

Indledelsesvis vil vi stadig foretage en kort analyse, så vi får en bedre forståelse af domænet, der skal skabes, samt for at sikre at vi får alle spillets mest væsentlige funktioner med, sådan at der ikke glemmes hverken krav eller designelementer, og vigtigst af alt, at spillet lever op til kundens visioner og forventninger.

I analysen vil vi anvende UML artifacts som “Use-Case beskrivelser” og “Domænemodel” til at få et overblik over systemets forskellige klasser og kravspecifikationer. Da vores system er af mindre størrelse, tager vi kun de relevante artifacts med, som giver mening i vores projekt, sådan at der ikke er en overflødig mængde af diagrammer og lign.

I vores løbende Design-fase, vil vi nu anvende vores UML artifacts til at udarbejde et design-klassediagram, og sekvensdiagram. Disse artifacts skal hjælpe os i implementeringen af systemet, sådan at udviklerne ved fuldstændig hvordan systemet skal bygges op, med de forskellige klasser, attributter, metoder og relationer.

Ved at have et godt fundamentalt design på papir, sikrer vi at “GRASP”[2] - principperne overholdes, og vi sikrer et funktionelt og bæredygtigt system.

Til implementeringen af dette spil, vil vi arbejde i Eclipse, der er et udviklingsværktøj for programmører, hvor man kan skabe java programmer, f.eks et Monopoly spil. For at vores

[1] (Udvidelse samt forbedring af indledningen fra CDIO_3 rapporten)
[2] https://en.wikipedia.org/wiki/GRASP_(object-oriented_design)

udviklere har nemmere ved at samarbejde omkring udviklingen af selve koden til spillet, bruger vi programmet GitHub. GitHub tillader os at lave god versionsstyring ved at vi kan arbejde i forskellige "branches", sådan at der ikke går noget kode tabt, og så tillader GitHub også, at op til flere udviklere kan arbejde samtidigt på det samme program.

Inden spillet er klar til at blive afleveret til kunden, skal det testes. Kunden har specifikt bedt os om at udføre en række tests på programmet, for at sikre at de modtager et bæredygtigt og tilregneligt produkt. Vi vil udføre de nødvendige tests på programmet og sikre, at spillet lever op til den forventede kvalitet.

2 Analyse

2.1 Krav

2.1.1 Funktionelle krav

1. Chance-kort (laver dem som klasser for de forskellige typer af chance kort samt får dem til at nedarve fra en abstract chance kort klasse. Så vil vi have vores chance kort i et array som vi så kan trække dem fra.)
2. Auktion af felt ingen ejer, men som man ikke vil købe.
3. Man skal kunne pantsætte sin bolig.
4. 'Slip ud af fængsel'-kort. (Vi har tænkt os at det skal bruges automatisk hvis man har det når man kommer i fængsel)
5. Kom ud af fængsel ved at slå 2 ens
6. Hvis man er i fængsel i 3 runder i træk betales 1000 kr og flyt det antal øjne man har slået
7. Spillet skal have 40 felter.
8. Salg af huse og hoteller.
9. Kan kun købe hoteller, hvis man har max huse på alle felter af samme farve.
10. Kan kun købe huse, hvis man ejer alle felter af samme farve. (omskrivning af vores metode som gør man får dobbelt leje hvis man ejer alle felt i en farve)
11. Spillet bedømmer ud fra en randomizer, hvem der starter.
12. 2 ens tre gange i træk, gør at man kommer i fængsel.
13. 2 ens giver ekstra tur.
14. 10% tab på salg af hus og hotel.

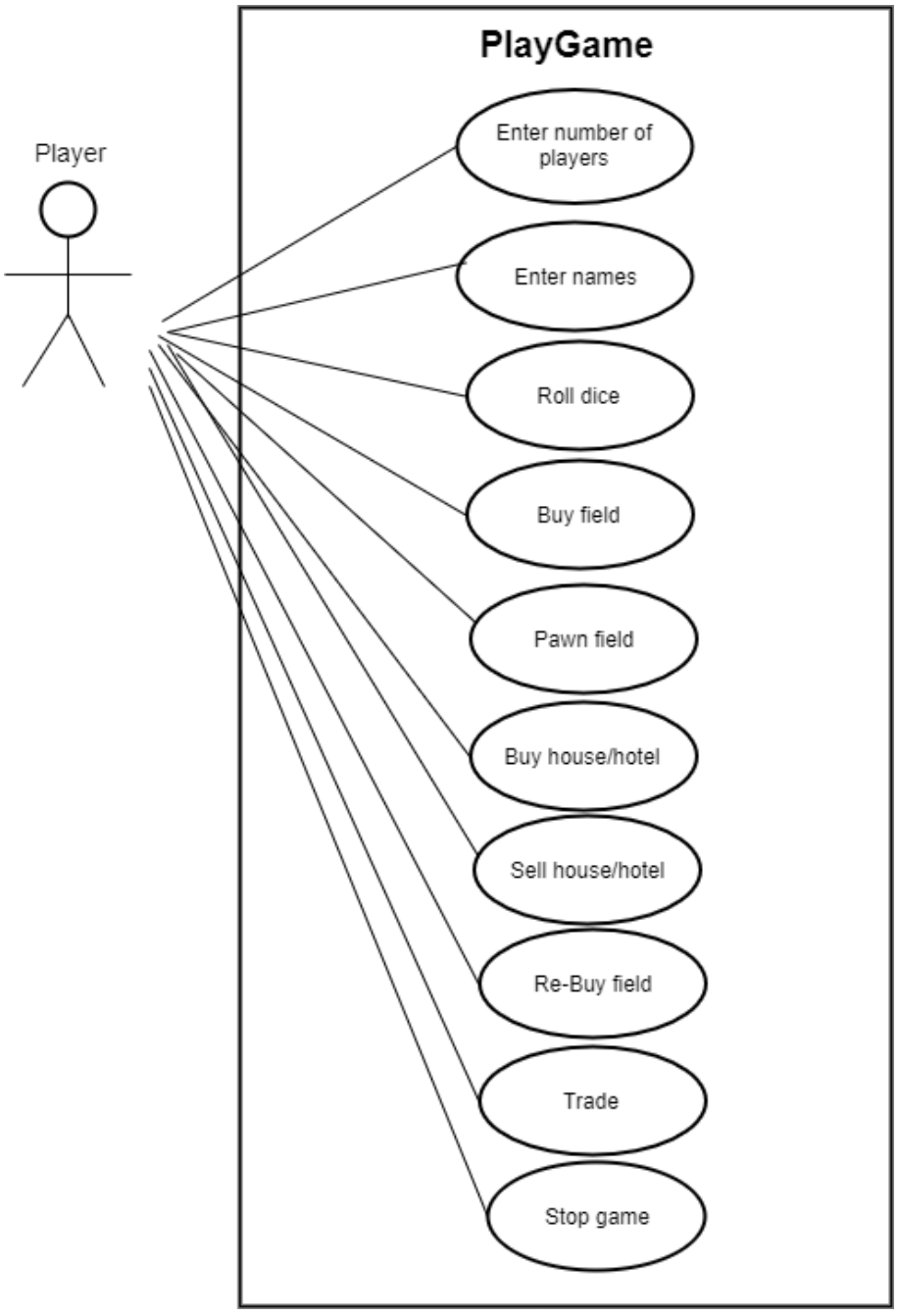
2.1.2 Ikke-funktionelle krav:

15. Spillet skal køre uden bemærkelsesværdige forsinkelser[3].
16. Spillet skal let kunne oversættes til et andet sprog.
17. Spillet skal kunne spilles på DTU's databarer.
18. Spillet skal have en koldstart på under 6 sekunder og en varmstart på under 3 sekunder

[3] En bemærkelsesværdig forsinkelse er alt over 300 millisekunder.

2.2 Use case diagram

De krav som spilleren skal kunne inde i matador spillet, er udviklet som use cases. Nedenstående diagram viser de forskellige interaktioner som spilleren kan tilgå i programmet.



Figur 2.1: Use Case diagram over matador spillet

2.3 Use case - Fully dressed

Use Case Beskrivelse

Spillerne angiver antallet af spillere mellem 2-6. Spillerne skriver deres navn, hvor hvorefter hver spiller får 30.000 kr. som startbeløb. Når det er gjort, vælges en tilfældig spiller, som starter med

at kaste terningerne. Systemet rykker derefter spilleren, afhængigt af terningernes sum. Herefter vises feltets oplysninger til spilleren, hvor han kan vælge at tage beslutninger om enten at afslutte sin tur, købe grunden. Næste spiller kaster dernæst terningen, hvorefter samme procedure sker. Dette gentager sig indtil alle spillerne bortset fra én spiller har mistet alle sine point og har ikke nogle huse at pantsætte. Systemet udråber den sidste spiller som vinder, hvis alle er gået fallit.

Use Case Sections	Comments
Use Case Navn	PlayGame
Scope	Matador.
Niveau	User-goal
Primær Aktør	Spiller
Interessenter	Ingen
Forudsætning	Programmet er installeret og klar til brug
Success garanti	Systemet udnævner en vinder blandt spillerne.
Primære Scenarie	**1.** Spiller starter spillet med 2 til 6 spillere. **2.** Tilfældig spiller vælges af systemet til at kaste de første terninger. **3.** Systemet flytter spilleren til et felt afhængigt af terningernes øjne. **4.** Feltets oplysninger udskrives til spilleren i GUI'en. *Denne proces gentager sig indtil alle spillere er bankerot (har 0 point og ingen grunde) bortset fra én spiller.* **5.** Systemet udkårer den sidste spiller vinder.
Alternativ flow	**4a.** Spilleren lander på et **ledigt** bolig-felt: **1.** Spilleren kan betale boligens pris. **a.** Spilleren ejer bolig-feltet. **2.** Spilleren vælger at afslutte sin tur. **4b.** Spiller lander på et **ejet** bolig-felt: **1.** Spilleren betaler boligens lejepris. **a.** Ejeren modtager pengene. **4c.** Spilleren lander på chance felt: 1. Spilleren trækker et kort som bliver udskrevet i GUI'en.

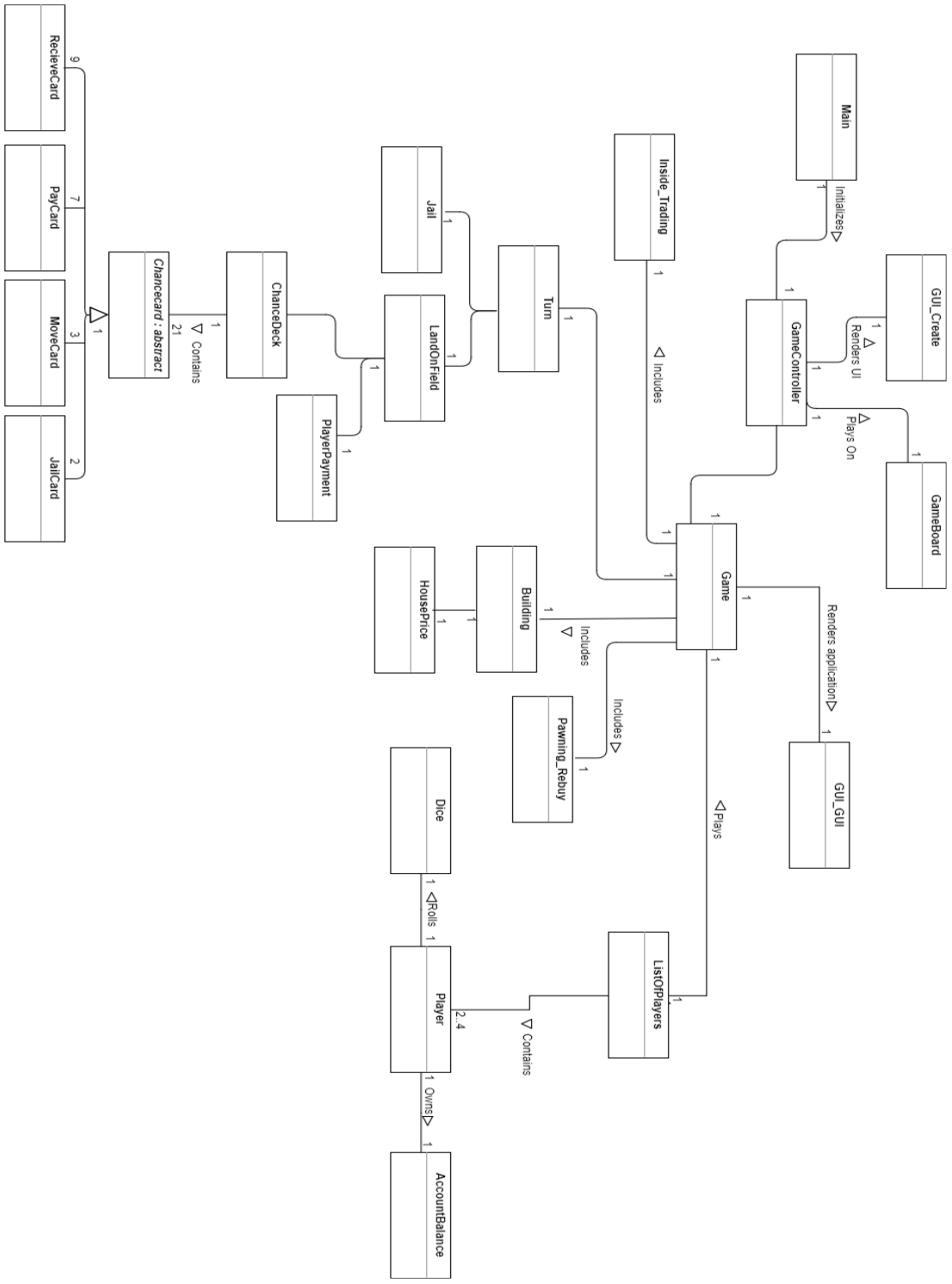
	a. Kortet bliver aktiveret **3a.** Spiller passerer startfeltet: 1. Spilleren modtager 4000 point **3b** Spiller lander på "gå i fængsel": 1. Spilleren flyttes til fængslet (felt 10) 2. Spilleren modtager ikke 4000 point ved passering af startfelt *For at komme ud af fængslet*: - Spilleren skal betale 1000 point for at komme ud af fængslet *Eller* - Spilleren skal kaste terningerne med to ens (hvis dette ikke er opnået inden for 3 ture, betaler han automatisk 1000 point) ***a.** Hvilken som helst tid, hvis systemet crasher: **1.** Spilleren genstarter spillet. **2.** Spilleren følger derefter det primære scenarie 1.-5.
Specielle krav	Spillet skal have tekst, der siger, hvad brugeren skal gøre for at starte spillet. Spillet skal køre uden bemærkelsesværdige forsinkelser[4]. Spillet skal kunne spilles på DTU's databarer. Spillet skal have en koldstart på under 6 sekunder og en varmstart på under 3 sekunder
Frekvens	Et spil
Andet	Intet

Tabel 2.2: En fully dressed use case beskrivelse af programmet

[4] En bemærkelsesværdig forsinkelse er alt over 300 millisekunder.

2.4 Domænemodel

Nedenstående domænemodel viser opstillingen af klasserne og deres forbindelse til hinanden i vores program.

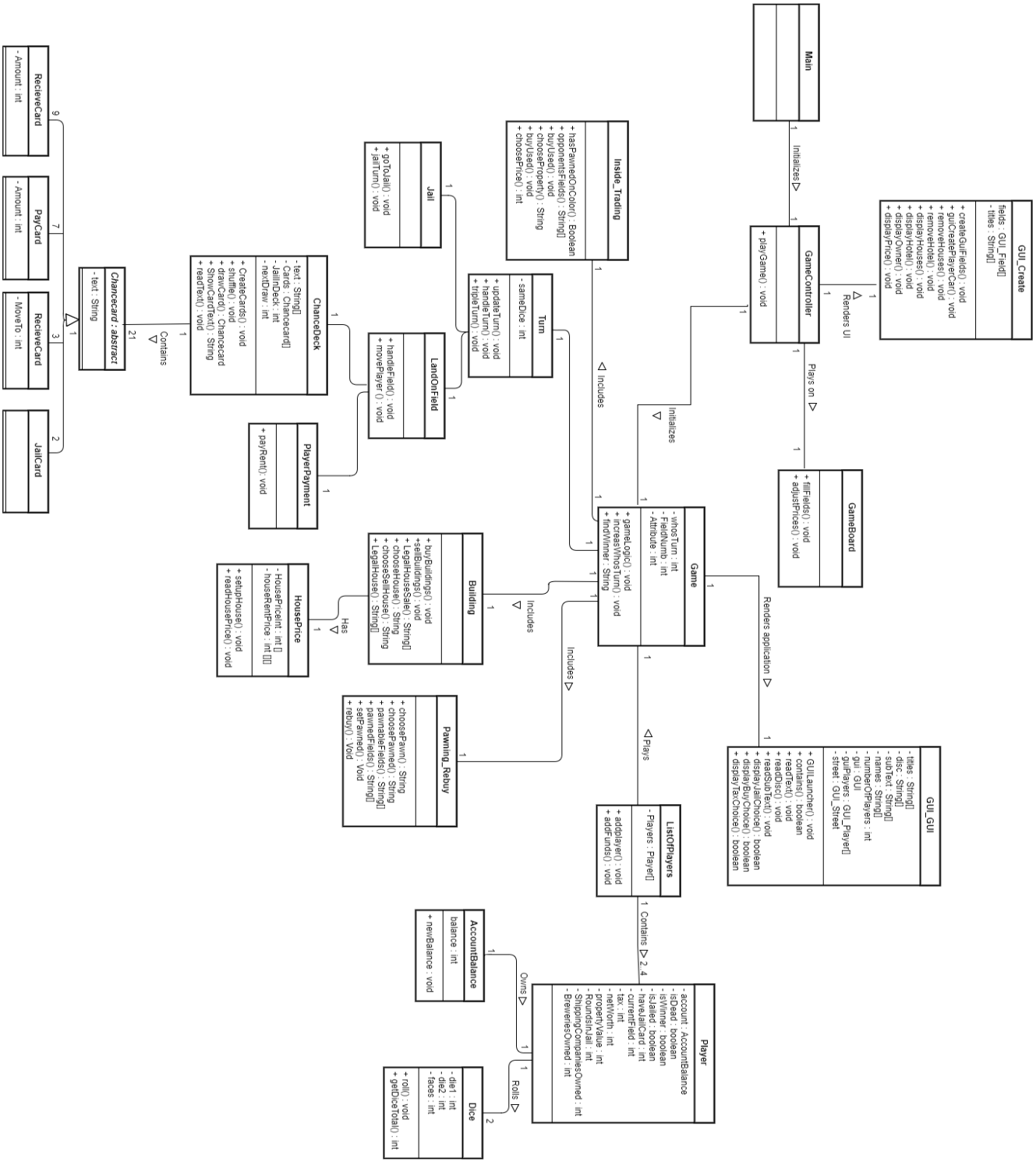


Firgur 2.3: Domænemodel med relation og multiplicity

3 Design

3.1 Designklassediagram

Nedenfor er illustreret et designklassediagram, som indeholder deres tilhørende attributter og metoder.

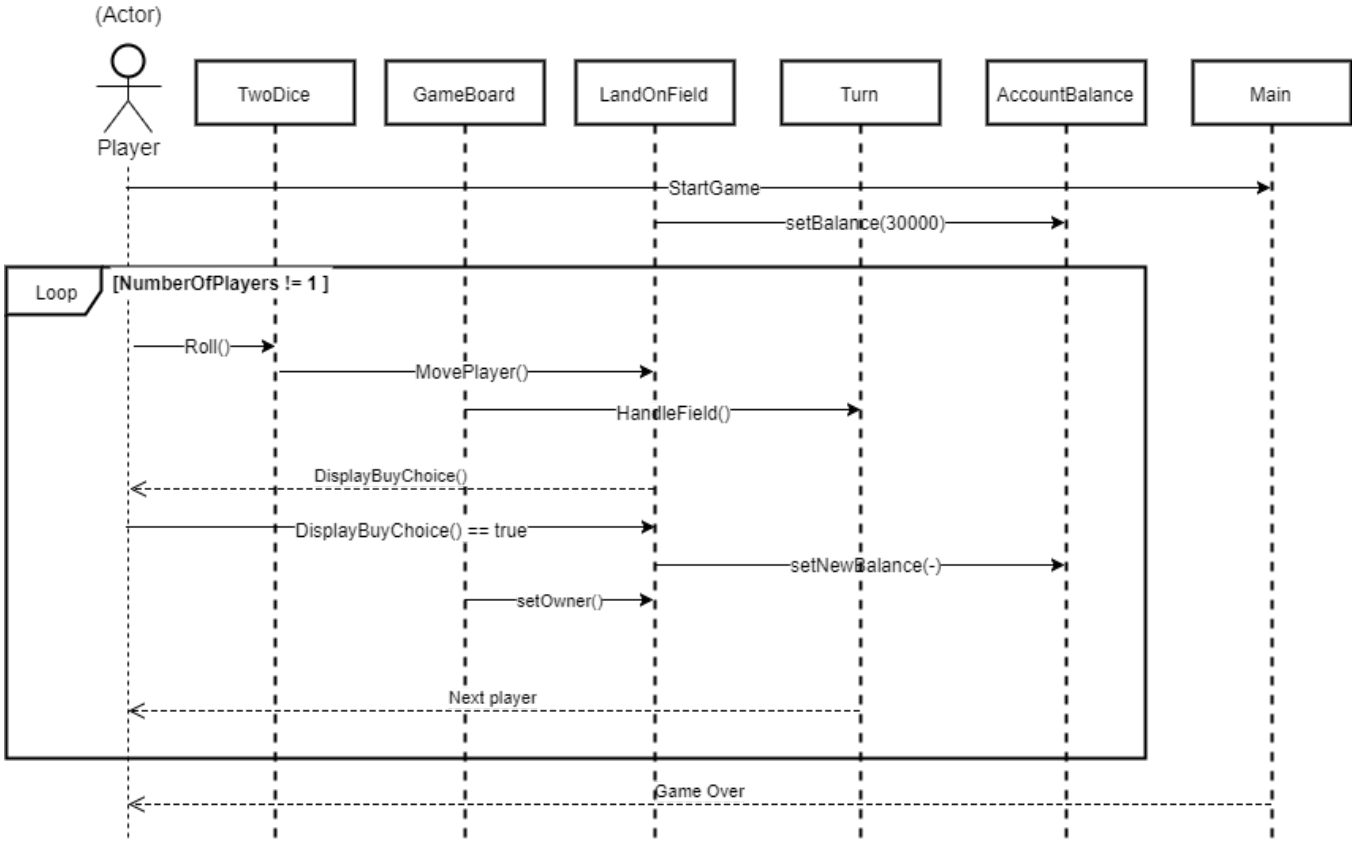


Figur 3.1: Designklassediagram over spillets klasser med deres attributter og metoder

3.2 Sekvensdiagram:

3.2.1 Buy field

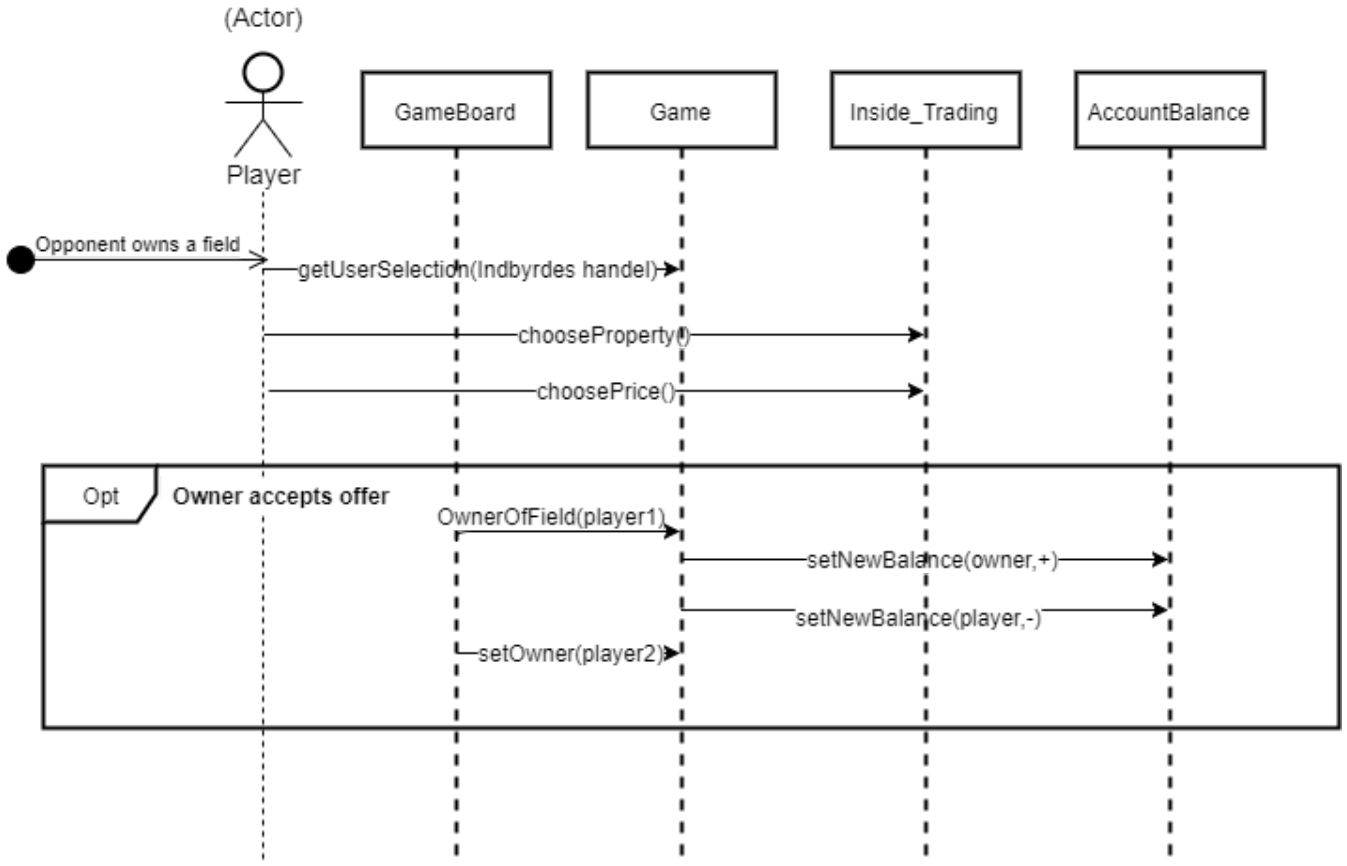
Diagrammet nedenfor illustrerer et systemdiagram, der følger processen når en spiller køber en grund i spillet. Først starter spilleren spillet, hvilket opretter en konto til brugeren af 30.000 kr. Derefter skal spilleren kaste med terningen, hvor så han bliver rykket til et ledigt felt. Spilleren får besked om han vil købe grunden eller afslutte sin tur. Når han køber grunden, tages der point fra spilleren, hvorefter grundens oplysninger bliver ændret så feltet er ejet af spilleren. Dernæst er det en anden spillers tur.



Firgur 3.2: Sekvensdiagram over use case: Buy field

3.2.2 BuyUsed:

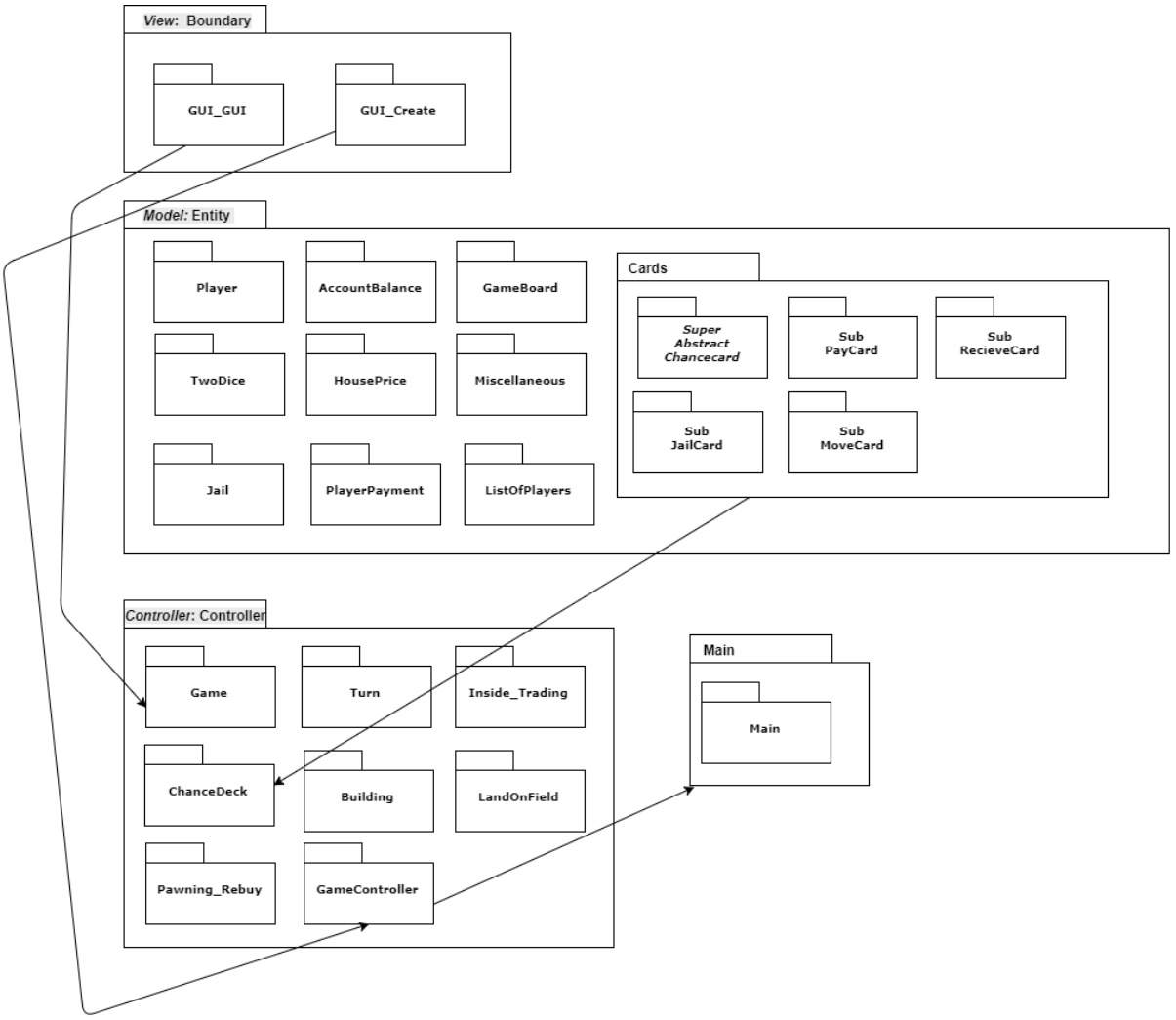
BuyUsed er funktionen til at lave en indbyrdes handel med andre. Diagrammet nedenfor viser hvordan en handel foregår. Hvis en spiller allerede ejer et felt, kan man vælge at lave en indbyrdes handel med spilleren, hvor så man skal vælge hvilken grund, man vil købe og derefter indtaste den værdi, man gerne vil give for grunden. Hvis ejeren af grunden accepterer, finder game klassen ejeren af grunden og laver en transaktion mellem ejeren og køberen, hvorefter køberen nu får ejerskab af grunden.



Figur 3.3: Sekvensdiagram over use case: BuyUsed

3.3 Pakkediagram

Vores kode er delt op i 5 pakker, som vist i vores pakkediagram. Vi har en pakke, der hedder 'Boundary', som indeholder GUI'en med klasserne GUI_GUI og GUI_Create, som udgør user-interfacen i spillet. Inde i Entity-pakken har vi information expert klasserne, hvilket udgør vigtige data for spillets funktioner. Controller pakken sørger for sammenspil mellem boundary- og entity pakkerne og får spillet til at fungere. Main-metoden befinder sig i vores Main-pakken og det er her hvor spillet startes. Nogle af klasserne uddybes i implementerings afsnittet.



Figur 3.4: Pakkediagram med forbindelser som er de væsentlige forbindelser

4 Implementering

4.1 ChanceDeck

ChanceDeck klassen består af dækkets funktioner. Det vil sige at, det er her hvor kortene bliver oprettet i en array af 21 kort. Kortene er oprettet af forskellige typer: RecieveCard, PayCard, JailCard og MoveCard. Kort-typerne er subklasser af Chancecard. Chancecard er en klasse, der ikke skal instantieres, så derfor er det en abstrakt klasse, hvor vi bruger de nedarvede klasser til at specificere kortene og deres funktion. Derudover har ChanceDeck også den opgave at trække kortet og aktivere den. Dette gøres med metoden drawCard(). Inde i drawCard er variablen draw det øverste kort i dækket. Det trukkede kort skal så håndteres, hvilket vi bruger operatoren instanceof til:

```
if(draw instanceof RecieveCard) {
    System.out.println(((RecieveCard)draw).getAmount());
    ListOfPlayers.getPlayers(Game.getWhosTurn()).setNewBalance(((RecieveCard)draw).getAmount());
}
```

Instanceof tjekker om objektet er en instans af en specifik type. Dette giver en boolsk værdi, så derfor bruger vi den til at tjekke typerne så de kører deres kode. Hvis man eksempelvis trækker kortet af typen RecieveCard så skal den tilføje de point til spillerens konto.
Endvidere har ChanceDeck klassen også metoden shuffle().

```
52      public static void shuffle() {
53          for(int c = 0; c < 3; c++) {
54              for(int i = 0; i < ChanceDeck.Cards.length - 1; i++) {
55                  int j  = (int) ((Math.random()) * (ChanceDeck.Cards.length - 1));
56                  Chancecard temp  = ChanceDeck.Cards[j];
57                  ChanceDeck.Cards[j] = ChanceDeck.Cards[i];
58                  ChanceDeck.Cards[i] = temp;
59              }
60          }
61          ChanceDeck.nextDraw = -1;
62          System.out.println("Kortene er blandet");
63      }
```

Her bruges en nested for-loop, hvor den ydre løkke sørger for at blande kortene tre gange og den indre løkke blander kortene vha. Math.random af antallet af kort i dækket hvor så den bliver castet til en int.

4.2 GameLogic(metode)

En af de mest centrale metoder i spillet er vores spillogik, kaldet GameLogic() i Game-klassen. GameLogic() sørger for at give spilleren forskellige valg, når det er deres tur, samt vælge hvem der starter, blande chance-kortene, lave terninger og give alle spillere 30000 kr.

```
public static void gameLogic() throws IOException {
    //Game logic

    //Randomize whosTurn
    whosTurn = (int) Math.ceil(Math.random() * GUI_GUI.getNumberOfPlayers());


    //      //Create dice
    TwoDice dice = new TwoDice();
    ChanceDeck.CreateCards();
    ChanceDeck.shuffle();

    ListOfPlayers.addFunds(GUI_GUI.getNumberOfPlayers());
```

Efter disse ting er på plads, kommer den vigtigste del af metoden, nemlig switch-casen der styrer alle spillerens valg. Switch-cases har mulighederne: “Kast”, “Byg huse/hotel”, “Pantsæt grunde”, “Genkøb”, “Indbyrdes handel”, “Sælg huse” og en default der hedder “Selection not recognized”.

```
switch (GUI_GUI.gui.getUserSelection("                                    Det er: " + ListOfPlayers.getPlayers(whosTurn).getName() + "'s tur, vælg hvad du vil fortage dig", "Kast", "Byg huse/hotel", "Pants
case "Kast":
    System.out.println("1"); //Printout to console for log purposes.
    dice.roll();
    turn.updateTurn(dice.getdie1(), dice.getdie2(), ListOfPlayers.getPlayers(whosTurn));
    break;
case "Byg huse/hotel":
    System.out.println("2"); //Printout to console for log purposes.
    if (build.LegalHouse().length != 0) {
        build.buyBuildings(misc.titleToInt(build.chooseHouse()), ListOfPlayers.getPlayers(whosTurn), InTr.hasPawnedOnColor(ListOfPlayers.getPlayers(whosTurn).getCurrentField()) );
    } else {
        GUI_GUI.gui.showMessage("                                    Du ejer ikke alle grunde i denne farve endnu");
    }
    break;
case "Pantsæt grunde":
    System.out.println("3"); //Printout to console for log purposes.
    if (PawReb.pawnableFields().length != 0) {
        PawReb.setPawned(misc.titleToInt(PawReb.choosePawn()), misc.hasHousesOnColor(ListOfPlayers.getPlayers(whosTurn).getCurrentField()));
    } else {
        GUI_GUI.gui.showMessage("                                    Du ejer ikke nogen grunde");
    }
    break;
case "Genkøb":
    System.out.println("4"); //Printout to console for log purposes.
    if (PawReb.pawnedFields().length != 0) {
        PawReb.rebuy(misc.titleToInt(PawReb.choosePawned()), ListOfPlayers.getPlayers(whosTurn));
    } else {
        GUI_GUI.gui.showMessage("                                    Du ejer ikke nogen pantsatte grunde");
    }
    break;
case "Indbyrdes handel": //Printout to console for log purposes.
    System.out.println("5");
    if (InTr.opponentsFields().length != 0) {
        InTr.buyUsed(misc.titleToInt(InTr.chooseProperty()), misc.hasHousesOnColor(ListOfPlayers.getPlayers(whosTurn).getCurrentField()), ListOfPlayers.getPlayers(whosTurn));
    } else {
        GUI_GUI.gui.showMessage("                                    Dine modstandere ejer ikke nogen grunde");
    }
    break;
case "Sælg huse": //Printout to console for log purposes.
    System.out.println("5");
    if (build.LegalHouseSale().length != 0) {
        build.sellBuildings(misc.titleToInt(build.chooseSellHouse()), ListOfPlayers.getPlayers(whosTurn));
    } else {
        GUI_GUI.gui.showMessage("                                    Du ejer ikke nogen huse endnu");
    }
    break;
default:
    System.out.println("Selection not recognized"); //Printout to console for log purposes.
    break;
}
```

Hvis vi starter fra en ende af: Kast kalder metoden dice.roll(), som kaster de to terninger. Derefter bliver metoden Updateturn() kaldt, som kalder metoderne til at returnere terningernes værdier og finde ud af hvis tur det er. Inde i UpdateTurn() bliver der kørt en række “if-else” statements, der bl.a. har til formål at tælle hvor mange gange spilleren har slået 2 ens. UpdateTurn() kalder så handleTurnI() som håndterer hvad der faktisk blev slået, og alt efter om spilleren sidder i fængsel eller ej, flytter spilleren til det korrekte felt. Når spilleren lander på feltet er det “handleField”, som styrer hvad der sker, når man lander på feltet. Når turen så er slut, breaker switch-casen, og GUI’en viser næste spiller de samme 6 valg.
Det næste valg er “Byg huse/hotel”, som først kalder metoden LegalHouse() og ser om String arrayet deri er forskellige fra 0. Hvis den er 0 går casen ned i en else, som printer: “Du ejer ikke alle grunde i denne farve endnu”. Hvis arrayet er anden end 0, betyder det at for-loopet i LegalHouse() har fundet at spilleren ejer alle felter af en given farve. Spilleren får nu en liste over de grunde, han kan købe huse på. Trykker han køb kører metoden buyBuildings() en serie af if statements, for at tjekke om der er pantsatte grunde og om spilleren har råd. Opfylder spilleren alle kravene, bliver gui’ens metode displayHouse() kaldt.
“Pantsæt grunde” minder om køb huse. Først bliver pawnebleFields() metoden kaldt i en if-statement, hvor den bliver sammenlignet med 0. Ligesom før bliver der kørt et for-loop, hvor der bliver tjekket om spilleren ejer nogle felter på brættet. Hvis spilleren ikke gør dette, bliver der udskrevet: “Du ejer ikke nogen grunde”. Hvis spilleren faktisk ejer nogle grunde, vil en liste over disse bliver printet til spilleren. Klikker han på en grund, bliver metoden setPawned() kaldt, som først tjekker om der er huse på grunden, og derefter sætter grunden som pantsat, så der ikke længere bliver krævet leje. Er der tale om rederier eller elselskaber, retter den også disse til.
“Genkøb” er mere af det samme. En if else-statement tjekker først om arrayet i metoden pawnedFields er forskellig fra 0. Hvis det er det printer spillet, at der ikke er noget at genkøbe. Hvis spilleren har pantsatte ejendomme, vil det blive fundet på samme måde som før, blive fundet af den næsten samme for-loop, hvor én parameter er ændret.

```
public String [] pawnedFields() {
    String [] SFields = new String[40];
    String [] refinedFields;
    int size = 0;
    for (int i=0; i<40; i++) {
        if(Game.getWhosTurn() == Fields[i][4] && Fields[i][8] == 1) {
            SFields[i] = "" + Fields[i][0];
            if(SFields[i] != null) {
                SFields[i] = GUI_GUI.getTitles()[i];
                size++;
            }
        }
    }
    refinedFields = new String[size];
    int temp = 0;
    for (int i=0; i<40; i++) {
        if(SFields[i] != null) {
            refinedFields[temp] = SFields[i];
            temp++;
        }
    }
    return refinedFields;
}
```

En liste bliver præsenteret til spilleren, hvor de et klik på en grund kalder rebuy() metoden. Metoden går ind og finder genkøbs-prisen og ganger den med 1,1. Dette sammenlignes med spillerens konto, for at se om spilleren har nok penge. Hvis man har nok penge bliver grunde sat som ikke pantsat, og spillerens konto bliver opdateret.
Indbyrdes handel tjekker med samme slags for loop alle de ejede felter på pladen, som ignorerer ens egne. Felterne bliver præsenteret til brugeren, som kan trykke på en af dem. Metoden choosePrice() bliver så kaldt, og spilleren kan indtaste et bud, og så klikke OK. Hvis spilleren der bød på grunden har penge nok, så vil buyUsed() metoden stå for at kalde metoderne til at opdatere de to kontoer og opdatere GUI'en.
Den sidste mulighed er "Sælg huse" som fungerer meget på samme måde som sælg grunde. En metode der kalder en anden metode, som bruger et for-loop til at finde alle spilerens huse. Spilleren kan så klikke på en grund hvorpå de vil sælge, og en ny metode sørger for at opdatere konto og gui.
While-løkken uden om switch-casen sammenligner antallet af levende spillere med døde spillere, og bryder lykken, hvis der kun er 1 spiller tilbage.

4.3 GameBoard(klasse)

Gameboard er som navnet antyder vores spillebræt. Dette spillebræt består af et todimensionelt int array som har 40 pladser i den første dimension, mens der er 10 pladser i den anden dimension. Den første dimension indeholder vores felt numre, som har de 40 pladser. De 10 pladser i den anden dimension er for attributterne for felterne. Indeks 0 i anden dimension indeholder ikke en attribut, siden den bruges til at referere til felt nummeret. De andre indeks er derimod attributter for lejen, farven, ejet, ejer, kan ejes, købspris, pantsætnings værdi, er pantsat og til sidst bygninger.
Til nogle af disse attributter så bruger vi dem på en måde, som minder om true og false ved at hvis for eksempel isOwned er 1, så er feltet ejet, mens hvis den er 0, så er den ikke ejet. Vi bruger det samme princip på isOwnable og isPawned. De andre attributter bruger vi dog på en anden måde.

Et eksempel på dette er owner som vi sætter til den whosTurn værdi som den spiller som er owner tager en tur på. Så hvis en spiller tager sin tur når whosTurn er 3 så vil de felter han ejer sige at ejeren er 3. Buildings attributten er meget simpel i og med at det bare er hvor mange huse man har på en grund dog hvis Buildings er 5 så er der ikke 5 huse siden man kun kan bygge 4 men det betyder derimod at der er et hotel på grunden. buyPrice og pawnPrice er rimelig nemme at gennemskue ved er siden at det bare er købsprisen på en grund samt hvor mange penge man får for at pantsætte en grund.
Vi laver dette array ved at bruge metoden fillFields som initialisere vores array. Metoden fillFields fungerer ved at vi først laver et tomt todimensionelt int array, int fields[][], med det antal pladser vi skal bruge derefter har vi et forloop der kører 40 gange og inde i dette forloop sættes felt nummeret samt køres en switchcase som tager imod en int i som er den int som forloopet forøger hver gang man løber igennem det. Dette gør så vi skiftevis kommer igennem alle cases og i hver case der initialiserer vi attributterne for det specifikke felt. Der er også felter hvor der ikke skal initialiseres noget som for eksempel chance felter siden det er et specielt felt i forhold til de normale grund felter. Når forloopet så er kørt færdigt så kaldes Game klassens metode setFields() som tager et todimensionelt int array som argument og som så sætter Fields i Game klassen til det array som blev lavet i GameBoard klassen.

```
/**
 * Field[][] har formen [FieldNumb][Attributes], hvor [Attributes] = [FieldNumb, rent, color, isOwned, owner, isOwnable, buyPrice, pawnPrice, isPawned, buildings]
 */
public class GameBoard {
    public static void fillFields() {
        int field[][];
        field = new int [40][10];
        for (int i = 0; i < 40; i++) {
            field[i][0] = i;
            switch (i) {
            case 0:
                break;
            case 1:
                field[i][1] = 50;
                field[i][2] = 1;
                field[i][5] = 1;
                field[i][6] = 1200;
                field[i][7] = 600;
                break;
            case 2:
                //              Chance felt
                break;
            case 3:
                field[i][1] = 50;
                field[i][2] = 1;
                field[i][5] = 1;
                field[i][6] = 1200;
                field[i][7] = 600;
                break;
```

4.4 PawnableFields(metode)

I metoden pawnableFields kører vi et for loop der går igennem alle felterne på pladen. Inden i loopet har vi nogle if statements der tjekker om feltets ejer matcher med den spiller hvis tur det er og om feltet ikke allerede er pantsat.
Hvis if statementet er true så sætter vi feltet ind i et midlertidigt array med 40 indexes. Så har vi et nested if statement der tjekker om indexet i det midlertidige array er null, og hvis den ikke er null, så sætter vi indexet til at være lig med titlen (gadenavnet) på feltet. Samtidigt increaser vi int size, der holder styr på hvor mange felter der ikke er null.

Derefter laver vi et refined array der har størrelse size. Her laver vi et nyt for loop og tjekker om index ikke er null i det første array, og hvis det er true så sætter vi værdien ind i vores refined array. Samtidigt increaser vi int temp, der sørger for at vores refined array kun går videre til næste index når der bliver indsat en værdi.
Det er nødvendigt at lave et refined array, da en metode i den udleverede GUI, der giver spilleren mulighed for at vælge i en dropdown menu, ikke har et tjek for en NullPointerException. Derfor er vi nødt til at lave et dynamisk array der tilpasser sig så det er præcist den rigtige størrelse.
Af den årsag har vi et par variationer af koden i pawnableFields til hver gang vi skal lave en dynamisk dropdown. Det er blandt andet i metoden pawnedFields() og opponentsFields(), her tjekker vi bare nogle andre værdier fx om feltet ikke er ejet af den spiller hvis tur det er.

5 Test

Det er altid vigtigt at teste ens software, for at se om det opfylder de krav, der er stillet til programmet. Vi har i vores projekt valgt, at teste både efter black-box- og white-box- principperne. Vores black-box tests er lavet af en fra holdet, som har ladet som om han ikke har kendskab til koden. White-box testene er udført ved hjælp af JUnit, som er meget god til at køre scenarier mange tusinde gange. Dette er en fordel, når man f.eks. vil se om ens kode holder sig inden for den teoretiske sandsynlighed. At teste programmet grundigt og finde så mange fejl så muligt er en af måderne, hvorpå man kan sikre sig et program af høj kvalitet.
Vi har i ved hjælp at vores test fundet mange fejl og rettet dem. Dog har vi også fejl som vi ikke har kunne nå eller kunne finde ud af at rette. Men på trods af dette føler vi at vores program er tilfredsstillende testet.

5.1 Black-box test

Black box testing er en metode, hvor man tester programmet uden at kende koden. Man kører programmet og prøver forskellige input og scenarier, derefter sammenligner man outputtet med det forventede output. Det hele skrives så ned i et skema. Dette giver udviklerne mulighed for at rette eventuelle fejl, som testerne har fundet. I vores projekt har en af vores gruppemedlemmer testet koden, ved at prøve forskellige scenarier af, og dokumentere hvad der skete. Dertil har vi udtænkt et forventet output, som vi sammenligner med. Det forventede output er det, vi kunne forestille os et perfekt implementeret Matador ville gøre. Når fejlen bliver opdaget, siger man det til udviklerne, som så kan rette den. Det samme testscenarie bliver så kørt igen i et nyt skema, for at se om fejlen er rettet. Selvom vi måske ikke får rettet alle fejl, så er det stadig vigtigt at have kendskab til dem.
Skemaet nedenunder er vores originale skema, hvor alle fejlene først blev opdaget. Når de så blev løst efterhånden, så skrev man det ind under “fix status”. I bilag 1 har vi et nyere skema, hvor alle testscenarier er kørt igen, for at teste om de er blevet løst.

Test Case ID (Kørsel)	Resumé	Aktuelt output	Forventet output	Status (Fix status)
#BB01 (12/1/18 kl. 12:30)	Input 1 person. Tryk OK.	Som forventet.	Spillet lader ikke spilleren trykke OK.	Bestået.

#BB02 (12/1/18 kl. 12:32)	Input 1 person. Tryk “enter”.	Spillet starter og spørger efter navn. Derefter crasher det.	Spiller lader ikke spilleren trykke enter.	Ikke bestået. (Anmeldt) (Løst)
#BB03 (12/1/18 kl. 12:37)	Input 6 personer. Giv dem samme navn.	Spillet viser kun 1 spiller-	Spillet tillader ikke at have det samme navn	Ikke bestået. (Anmeldt) (Løst)
#BB04 (12/1/18 kl. 12:39)	Lav “indbyrdes handel” Byd flere penge end du har.	Som forventet	Spillet siger at du ikke har nok penge.	Bestået.
#BB05 (12/1/18 kl. 12:44)	Lav indbyrdes handel. Byd noget der ikke er et positivt tal.	Som forventet.	Spillet sletter tegnet med det samme.	Bestået.
#BB06 (12/1/18 kl. 12:47)	Byg huse på en grund, uden at have alle af samme farve.	Spillet siger ikke noget, men går bare tilbage til den oprindelige menu.	Spillet siger at du ikke kan, fordi du mangler de andre grunde.	Ikke bestået. (Anmeldt) (Løst)
#BB07 (12/1/18 kl. 12:50) (12/1/18 kl. 13:12)	Genkøb grunde, selvom man ikke ejer nogle.	Spillet spørger “hvilken grund vil du pantsætte, og viser en liste over alle de ejede grunde. Klikker man på en, spørger spillet så hvad man vi byde på det.	Spillet siger at du ikke har nogle grunde.	Ikke bestået. (Delvist løst)
#BB08 (12/1/18 kl. 12:55)	Lav indbyrdes handel, uden af nogen ejer noget.	Spillet giver ingen besked, men går tilbage til tidligere menu.	Spillet siger at det kan du ikke.	Ikke bestået. (Anmeldt) (Løst)

#BB09 (12/1/18 kl. 13:03)	Sælg huse uden at have nogle.	Spillet går tilbage til tidligere menu, uden at sige noget.	Spillet siger at du ikke ejer nogle.	Ikke bestået. (Anmeldt) (Løst)
#BB10 (12/1/18 kl. 13:07)	Lav indbyrdes handel.	Spillet spørger hvilken grund du vil pantsætte, og viser alle andres grunde. Klikker man går den videre med indbyrdes handel.	Spillet spørge hvilket grund du vil handle.	Ikke bestået. (Anmeldt) (Løst)
#BB11 (12/1/18 kl. 13:23)	Giv en spiller “mellemrum som navn”	Spillet opretter en spiller med et usynligt navn.	Spillet siger at navnet ikke er gyldigt.	Ikke bestået. (Anmeldt) (Løst)
#BB12 (12/1/18 kl. 13:25)	Lav indbyrdes handel. Byd 0 kr.	Spillet spørger om modtageren vil acceptere buddet. Gør man det, siger spillet “handel afvist”.	Spillet afbryder handlen.	Ikke bestået. (Anmeldt) (Løst)
#BB13 (12/1/18 kl. 13:25)	Lav indbyrdes handel af en pantsat ejendom.	Spillet overfører ejerskab, men viser grunden til ikke længere at være pantsat.	Spillet overfører ejerskabet fra den ene til den anden spiller, dog beholder den grunden som pantsat.	Ikke bestået. (Anmeldt) (Løst)
#BB14 (12/1/18 kl. 13:25)	Start spil med 0 spillere.	Spillet crasher.	Spillet siger at tallet er ugyldig.	Ikke bestået. (Anmeldt) (Løst)
#BB15	Kom i fængsel.	Spillet giver	Spillet spørger dig	Ikke bestået.

(12/1/18 kl. 14:01)	Lad det blive din tur igen.	den normale menu. Klikker man på “kast”, spørger spillet om man vil betale eller slå 2 ens. Klikker man betal, slår spillet automatisk for en, om afslutter ens tur.	om du vil betale for at komme ud af fængsel, eller prøve at slå to ens. Derefter kan man fortsætte med sin tur.	(Anmeldt)
#BB16 (12/1/18 kl. 14:21)	Slå så man rammer “De fængsles”	Spillet smider en i fængsel uden at sige noget.	Spillet smider dig i fængsel, imens det siger at du er kommet i fængsel.	Ikke bestået. (Anmeldt) (Løst)
#BB17 (12/1/18 kl. 16:13)	Start spillet og tryk enter i stedet for at indtaste antal spillere.	Spillet crasher.	Spillet siger “ugyldig input”	Ikke bestået. (Anmeldt)
#BB18 (14/1/18 kl. 22:18)	Spil spillet indtil én spiller ejer huse på alle grundene af én farve. Vent til en anden spiller lander på grundene, og ikke har råd.	Spillet spørger spilleren uden penge, hvilken grund de vil pantsætte. Man kan så vælge alle ens grunde. Når man vælger en, siger spillet at man først skal sælge sine huse (selvom man ikke har nogle.)	Spillet erklærer spilleren uden penge for død.	Ikke bestået. (Anmeldt.)

Figur 5.1: Black-box testing tabel

5.2 JUnit

JUnit er en måde, hvor man i Java kan få 1 stykke kode til at teste et andet stykke kode. I vores tilfælde har vi lavet 2 forskellige te.

5.2.1 MoveCardTest

I Matador er chancekort en stor del af spillet, derfor er det vigtigt ift. dette projekt at de virker. Vi har valgt at teste den type kort, som beder spilleren om at flytte til et bestemt felt. I denne test starter spilleren på felt 2, og trækker så et kort der, skal flytte spilleren til felt 30. Junit vurderer så om currentField er blevet ændret til det rigtige.

Test Case ID	TC01
Test Case name	MoveCardTest
Summary	Test om moveCard rykker på spillerens currentField
Requirements	Skal rykke spilleren til felt 30
Preconditions	Spilleren trækker et chance kort med typen MoveCard, hvor han skal rykke til felt 30
Postconditions	
Test procedure	1. Åben projektet i Eclipse 2. Vælg “MoveCardTest” klassen ind i pakken Test 3. Kør programmet
Test data	getCurrentField
Expected result	currentField = 30
Actual result	True
Status	Bestået
Tested by	Søren Kaare Rasmussen
Date	11/01/18
Test environment	Eclipse 4.7.1a på Windows 10

Figur 5.2: Tabel over JUnit test af moveCard

5.2.2 TwoDiceTest

Terninger er den vigtigste del af Matador, da de bestemmer stort set alt interaktion i spillet. Derfor er fair terninger et must have, for denne type spil. Vi har fået Junit til at slå terningerne 60000 gange, og så sammenligne med den teoretiske sandsynlighed. Hvis resultatet er inden for 0,66% er testen bestået.

Test Case ID	TC02
Test Case name	TwoDiceTest
Summary	Test om terningens sandsynlighed, stemmer overens med den teoretiske statistik for to 6-siders terninger.
Requirements	Overholde den statistiske sandsynlighed med en max fejlmargen på 400 slag eller 0,66%.
Preconditions	Opret terning og kald getDiceTotal
Postconditions	
Test procedure	1. Åben projektet i Eclipse 2. Vælg "TwoDiceTest" klassen 3. Kør programmet
Test data	2 6-siders terningerrullet 60000 gange.
Expected result	True
Actual result	Vores resultater = teoretisk sandsynlighed +- 0,66%
Status	Bestået
Tested by	Thomas Mattsson
Date	14/01/18
Test environment	Eclipse 4.7.2 på macOS 10.13.2

Figur 5.3: Tabel over JUnit test af terningerne

6 Projektplanlægning

Som planlægning havde vi allerede på første dag udviklet en prioriteringsliste, der indebærer kategorierne: must have, should have og could have. Dette har givet os et godt overblik over selve spillet og alle dets regler og funktioner. På den måde kan gruppen sørge for, hvilke feature de vil implementere og derefter krydse af. Dette minder meget om SCRUM, hvor man også samler en liste krav, og så løser dem én efter én.

	Krav	Status	Årsag
Must have	Spillere kan gå fallit	Færdig	De helt essentielle dele af matador. Efter disse er implementerede vil man kunne spille spillet og vinde eller tabe.
	40 felter	Færdig	
	2 terninger	Færdig	
	Korrekt valuta ift. matador	Færdig	
Should have	Fjern kast for døde spillere	Færdig	De strategiske komponenter af matador, som helt sikkert ville gøre produktet bedre og sjovere at spille.
	Fjern biler for døde spillere	Færdig	
	Kom ud af fængsel ved at slå 2 ens	Færdig	
	Hvis man er i fængsel i 3 runder i træk betales 1000 kr og flyt det antal øjne man har slået	Færdig	
	Sæt grunde for døde spillere til salg igen	Færdig	
	Fængsel	Færdig	
	Komme ud af fængsel ved at betale 1000 kr. med det samme	Færdig	
	Betal dobbelt rent hvis alle i samme farve er eget	Færdig	
	Pantsættelse	Færdig	
	Simple chancekort	Færdig	
	Genkøb af pantsatte felter	Færdig	
	Kan selv vælge hvilke felter der skal pantsætte når fallit	Færdig	
	Komme ud af fængsel ved løsladelseskort	Færdig	
	Undgå crash ved tomt dropdown	Færdig	
	Betal indkomstskat og statskat feltet	Færdig	
	Kan kun købe huse, hvis man ejer alle felter med samme farve	Færdig	
	Huse og hoteller.	Færdig	
	Man kan vælge om man vil købe huse eller ej.	Færdig	
	Indbyrdes handel med grunde til aftalt pris.	Færdig	
	Kan sælge huse til halv pris	Færdig	
	Rigtig leje pris ved ejerskab af flere rederier/brygerier	Færdig	
	Vis hvem der vinder med en tekst i GUI	Færdig	
	Jævn bygning af huse på felter af samme farve	Ikke færdig	
Could have	Spillet bedømmer ud fra en randomizer, hvem der starter.	Færdig	De dele af spillet, som spillet sådan set godt kan undvære, og som mange spillere af Matador ikke kender så godt.
	2 ens giver ekstra tur.	Færdig	
	Når alle lykkekort er blevet brugt, skal man blande bunken	Færdig	
	2 ens tre gange i træk, gør at man kommer i fængsel.	Færdig	
	Prisen på grunden skal komme frem på GUI'en efter spilleren, som ejede det dør.	Færdig	
	Indkomstskat: Betal 4000 eller 10% af ens værdier. Spilleren skal skal vælge inden deres samlede værdier regnes sammen.	Færdig	
	Fængslede kan opkræve leje.	Færdig	
	Service meddelelser til bruger fx "Du røg i fængsel"	Færdig	
	Auktion af felt, der er blevet landet på, men som ikke købes	Ikke færdig	
	Overfør ejerskab af grunde til feltes ejer når man dør	Ikke færdig	
	Resterende kompliceret chancekort	Ikke færdig	
	Fængslede kan være med i auktioner.	Ikke færdig	

Figur 5.4: Prioriteringsliste over vores mål af hele projektets forløb

7 Konklusion

7.1 Proces:

Vi har i vores arbejdsgang valgt at arbejde iterativt. Denne proces gør at vi kan udvikle og ændre idéer og modeller løbende igennem hele forløbet.

Vi startede derfor projektet med at analysere opgaven og danne mulige løsninger og derfra fokusere på design, så vi hurtigt kunne opstille vores klasser med deres tilhørende associationer, attributter og metoder, så vi kunne gå i gang med at kode tidligst muligt.

Hele processen har været vellykket, eftersom vi kunne arbejde med analyse, design og implementeringen sideløbende, som hele tiden ændrer vores syn på systemets arkitektur, hvor vi derefter kan modificere nogle ting som følge af vores iterative tilgang.

Vi har eksempelvis ændret meget af vores designklassediagram og domænemodel, som følge af vores implementering, så modellerne passer med koden. Som helhed har vores arbejdsproces, hjulpet med at visualisere og udvikle vores monopoly-spil på en effektiv måde.

Det at vi valgte at følge en prioriteringsliste igennem forløbet, har virkelig vist sig at være en god ide. I vores forrige forløb er vi ofte røget lidt ud på et sidespor, og fulgt nogle mindre vigtige, og ofte tidskrævende funktioner til ende, hvor vi denne gang havde noget håndgribeligt at kigge på, så vi aldrig var i tvivl om, hvor vores fokus skulle lægges. Det har både sparet os tid, og sikret at vi fik de vigtigste funktioner med i spillet.

7.2 Produkt:

Vores afleverede produkt lever op til næsten alle de funktionelle og ikke-funktionelle krav, på nær de funktionelle-krav: 2 og 9, nemlig kravene omkring Auktion samt Jævn bygning af huse på felter med samme farve. Grunden til at disse krav ikke er blevet implementeret er at vi har prioriteret anderledes, med højere fokus på implementeringen af chancekort, huse og hoteller, og indbyrdes handel. Disse funktioner mener vi er mere grundlæggende og absolut nødvendige, for at spillet rent faktisk føles som monopoly. Vi valgte derfor at udelade disse krav. Ser man bort fra de 2 krav, har vi udviklet et rimelig velfungerende monopoly spil, som overholder alle vores andre krav specifikationer.

7.3 Perspektivering:

Igennem forløbet har vi udviklet monopoly-spillet, hvor vi har prøvet at opfylde alle vores krav. Ser vi overordnet på resultatet, er det et fornuftigt produkt vi har leveret, dog ikke uden mangler. Hvis man skulle se på nogle af de erfaringer, vi har skabt under forløbet, så er den der står mest ud, helt klart test. Fordi vi tidligere primært har arbejdet med meget små projekter, har vi ikke rigtig mærket konsekvenserne ved ikke at teste undervejs i forløbet. Vi har bygget små systemer op og testet dem, og rettet fejl, og det har været nok når det drejer sig om at lave 2 terninger der skal kunne slås med, og vises på en spilleplade, men når det drejer sig om et fuldt implementeret monopoly spil, så er det simpelthen essentielt at teste konstant undervejs. Den vigtigste testform vi tager med os, er en simpel Black-box test. Vi har opdaget mange fejl, ved blot at lade som om vi var en almen forbruger af systemet, og derved fundet frem til fejl, vi ellers ikke havde opdaget. Dette giver nu meget god mening eftersom black-box testing eller, forbrugertest, er den nok mest udbredte testform man ser hos spiludviklere. Man kan få indsamlet en masse data og opdage mange fejl ved blot at lade andre teste ens spil, og så kan man endda gøre det uden omfattende udgifter.

8 Bilag

Bilag 1: Anden kørsel af black box tests, hvor de fleste problemer er fixet.

Test Case ID (Kørsel)	Resumé	Aktuelt output	Forventet output	Status (Fix status)
#BB01 (14/1/18 kl. 12:35)	Input 1 person. Tryk OK.	Som forventet.	Spillet lader ikke spilleren trykke OK.	Bestået.
#BB02 (14/1/18 kl. 12:37)	Input 1 person. Tryk “enter”.	Som forventet	Spiller lader ikke spilleren trykke enter.	Bestået.
#BB03 (14/1/18 kl. 12:38)	Input 6 personer. Giv dem samme navn.	Som forventet.	Spillet tillader ikke at have det samme navn	Bestået.
#BB04 (14/1/18 kl. 12:39)	Lav “indbyrdes handel” Byd flere penge end du har.	Som forventet	Spillet siger at du ikke har nok penge.	Bestået.
#BB05 (14/1/18 kl. 12:40)	Lav indbyrdes handel. Byd noget der ikke er et positivt tal.	Som forventet.	Spillet sletter tegnet med det samme.	Bestået.
#BB06 (14/1/18 kl. 12:41)	Byg huse på en grund, uden at have alle af samme farve.	Som forventet.	Spillet siger at du ikke kan, fordi du mangler de andre grunde.	Bestået.
#BB07 (14/1/18 kl. 12:43)	Genkøb grunde, selvom man ikke ejer nogle.	Som forventet.	Spillet siger at du ikke har nogle grunde.	Bestået.
#BB08 (14/1/18 kl. 12:45)	Lav indbyrdes handel, uden af nogen ejer noget.	Som forventet.	Spillet siger at det kan du ikke.	Bestået.
#BB09 (14/1/18 kl. 12:50)	Sælg huse uden at have nogle.	Som forventet.	Spillet siger at du ikke ejer nogle.	Bestået.
#BB10 (14/1/18 kl. 12:53)	Lav indbyrdes handel (Kræver at nogen ejer noget).	Som forventet.	Spillet spørge hvilket grund du vil handle.	Bestået.
#BB11	Giv en spiller	Som	Spillet siger at	Bestået.

(14/1/18 kl. 12:55)	“mellemrum som navn”	forventet.	navnet ikke er gyldigt.	
#BB12 (14/1/18 kl. 12:57)	Lav indbyrdes handel. Byd 0 kr.	Som forventet.	Spillet afbryder handlen.	Bestået.
#BB13 (14/1/18 kl. 12:59)	Lav indbyrdes handel af en pantsat ejendom.	Som forventet.	Spillet overfører ejerskabet fra den ene til den anden spiller, dog beholder den grunden som pantsat.	Bestået.
#BB14 (14/1/18 kl. 13:00)	Start spil med 0 spillere.	Som forventet.	Spillet siger at tallet er ugyldig.	Bestået.
#BB15 (14/1/18 kl. 14:01)	Kom i fængsel. Lad det blive din tur igen.	Spillet giver den normale menu. Klikker man på “kast”, spørger spillet om man vil betale eller slå 2 ens. Klikker man betal, slår spillet automatisk for en, om afslutter ens tur.	Spillet spørger dig om du vil betale for at komme ud af fængsel, eller prøve at slå to ens. Derefter kan man fortsætte med sin tur.	Ikke bestået. (Anmeldt)
#BB16 (14/1/18 kl. 13:02)	Slå så man rammer “De fængsles”	Som forventet.	Spillet smider dig i fængsel, imens det siger at du er kommet i fængsel.	Bestået.
#BB17 (14/1/18 kl. 13:04)	Start spillet og tryk enter i stedet for at indtaste antal spillere.	Eclipse melder fejl, men spillet crasher ikke.	Spillet siger “ugyldig input”	Ikke bestået. (Delvist løst)
#BB18 (14/1/18 kl. 23:18)	Spil spillet indtil én spiller ejer huse på alle grundene af én farve.	Spillet spørger spilleren uden penge,	Spillet erklærer spilleren uden penge for død.	Ikke bestået. (Anmeldt.)

	Vent til en anden spiller lander på grundene, og ikke har råd.	hvilken grund de vil pantsætte. Man kan så vælge alle ens grunde. Når man vælger en, siger spillet at man først skal sælge sine huse (selvom man ikke har nogle.)		