Danmarks Tekniske Universitet

02312 Indledende Programmering

Gruppe 20

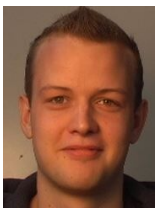
20. januar 2020

M1 - CDIO Final

Peter Kyhl Revsbech
s183760
GitHub: PeterRevsbech

Ida Schrader
s195483
GitHub: Idkirsch

Martin Mårtensson
s195469
GitHub: DAT4



Christian Kyed
s184210
Github: CKyed

Mohamad Abdulfatah Ashmar
s176492
GitHub: MO-AR

Marie Seindal
s185363
GitHub: MarieSeindal

GitHub link: CDIO_final
https://github.com/CKyed/CDIO_final

0.1 Timeregnskab

		Peter	Christian	Ida	Marie	Martin	Mohamad	I alt
Man	**d.6/1**	4	4	4	4	4	4	**24**
Tirs	**d.7/1**	8	8	8	8	8	8	**48**
Ons	**d.8/1**	8	8	8	8	8	8	**48**
Tor	**d.9/1**	8	8	8	8	8	8	**48**
Fre	**d.10/1**	8	8	8	8	8	8	**48**
Man	**d.13/1**	8	8	8	8	8	8	**48**
Tir	**d.14/1**	8	8	8	8	8	8	**48**
Ons	**d.15/1**	8	8	8	8	8	8	**48**
Tor	**d.10/1**	8	8	8	8	8	8	**48**
Fre	**d.17/1**	8	8	8	8	8	8	**48**
Samlet	**tid i timer**	**76**	**76**	**76**	**76**	**76**	**76**	**456**

Tabel 1: Timeregnskab for de individuelle dage og personer

Indhold

Resumé

Dette er en rapport over et 3-ugers projekt udført af studerende på DTU. Projektet går ud på at designe og implementere et matadorspil i Java. Den første del af rapporten omhandler kravanalysen. Her diskuteres og præseneteres de krav, som gruppen har stillet til projektet, dvs. især hvilke funktionaliter det færdige produkt skal indeholde.

Analysedelen beskriver gruppens overvejelser omkring hvad et fysisk matadorspil er, kan og indeholder. Dette leder videre til afsnittet om Design, hvor gruppens softwaredesign til projektet præsenteres og diskuteres. Gruppen benytter sig af et Model-View-Controller mønster samt et antal GRASP mønstre. Dette skal tilsammen sikre, at projektets arkitektur overholder almindelig god praksis for softwareudvikling, og bl.a. tillader videreudvikling og vedligeholdelse af projektet samt gøre det forståeligt for andre udviklere. I afsnittet præsenteres både en detaljeret og en forsimplet version af systemets klassediagram samt flere sekvensdiagrammer til at beskrive udvalgte dele af koden.

I implementeringsafsnittet beskriver gruppen i højere detaljeringsgrad hvordan udvalgte dele af koden rent faktisk er implementeret og virker. Det beskrives hvordan spillet indlæser data og tekstbeskeder fra tekstfiler på forskellige sprog. Primært indeholder afsnittet en beskrivelse af systemets forskellige contollerklasser, som tager sig af store dele af systemets logik. Det diskuteres hvordan gruppen har testet deres program, og der konkluderes slutteligt ud fra arbejdsforløbet som en helhed, at projektet har været vellykket og at de vigtigste krav til programmet er opfyldt.

1 Indledning

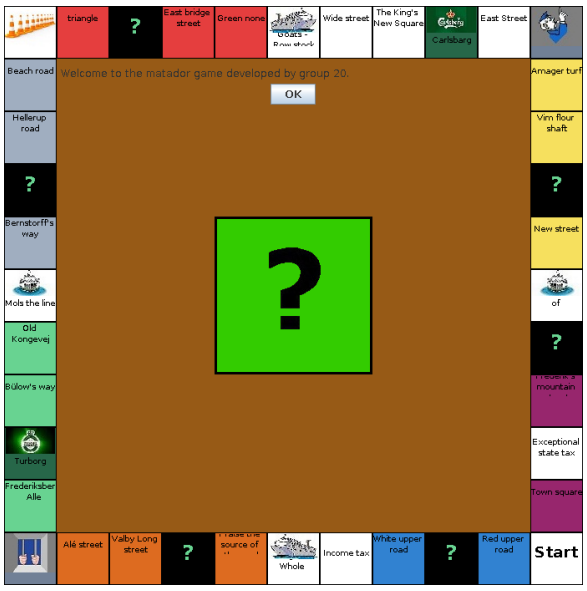
Dette projekt er lavet i tre-ugersperioden som en del af faget "Indledende programmering". Projektet består i at planlægge og udføre implementeringen af et klassisk matadorspil.

Målet med projektet er at vi lærer hvordan man udfører et større projekt i en gruppe, hele vejen fra 'idé' til udførelse. Vi har fået mulighed for at afprøve forskellige designmønstre og arbejdsflows. Spillet tager afsæt i tidligere projekter, men har flere features som komplicerer spillet. Valg og fravalg af features kommer vi ind på i analysedelen under krav.

Vi har valgt at strukturere os på en måde så hele gruppen har været med til at tage beslutninger vedrørende analyse. I implementeringsdelen af projektet havde vi øje for at dele opgaverne op på en måde så alle havde noget at lave, enten alene eller i mindre grupper. Undervejs har vi jævnligt holdt scrum-møder, både i starten, slutningen -og midt på dagen. For nærmere indblik i vores tidsforbrug se Gantt-diagrammet i slutningen af rapporten.

Vores endelige produkt er en spilbar computerversion af spillet Matador skrevet i Java, hvor vi har implementeret næsten alle regler og chancekort. Vi har primært fokuseret på at programmere logikken af spillet. Vi har fået udleveret en GUI som vi har brugt således at brugeren nemmere kan interagere med spillet. Ydermere har vi lavet vores egen lille gui som giver brugeren en mulighed for at vælge sprog i starten af spillet.

Figur 1: Brugeren bliver præsenteret med denne række af flag når han åbner spillet. Den udleverede GUI præsenteres herefter i det sprog som tilhøre det flag som brugeren har klikket på.



Figur 2: Her ses den engelske udgave af spillets udleverede GUI umidelbart efter at det engelske flag er klikket på.

Figur 3: Her ses den arabiske udgave af spillets udleverede GUI efter at spillere er oprettet med navn og biler og spiller 1 er rykket 4 felter fra start.

2 Krav

Vores arbejde med krav indebærer en usecase-model, idet vi forsøger at benytte de redskaber, vi har lært om Unified Process. I dette afsnit om krav, vil vi derfor starte med en beskrivelse af vores krav, og dernæst præsentere vores usecasediagram samt uddybe usecasemodellen med tekstbeskrivelser af de individuelle usecases.
Efter usecasemodellen beskriver vi vores krav i en prioriteret liste, der gerne skal danne overblik over, hvad projektet skal indeholde, samt hvad de vigtigste arbejdsopgaver bliver.
Vores krav er opdelt i 2 hovedkategorier:

- **Funktionelle krav**
- **Ikke-funktionelle krav**

Derudover er kravene opdelt i 3 kategorier, der viser kravenes prioritet.

- **Prioritet A**
  Krav under denne prioritet skal under alle omstændigheder implementeres i projektet.
- **Prioritet B**
  Krav som helst skal implementeres. Vi vil gøre alt hvad vi kan for at få dem med i det færdige produkt. Programmet kan dog stadig fungere som et acceptabelt Matadorspil uden disse krav.
- **Prioritet C**
  Krav som gruppen gerne vil implementere, men de bliver ikke taget i betragtning før at alle krav i de 2 andre grupper er implementeret.

Der er sat en * ud fra de krav, som vi ikke nåede at implentere. Dette er kun aktuelt i prioritet C.

2.1 Funktionelle krav

A

Opstart og drift

K1 Spillet spilles af 3 til 6 spillere

K2 Spillernes brikker starter på start-feltet.

K3 Hver spiller starter med 30.000 kr.

K4 Spillerernes tur går efter den rækkefølge de tilmeldte sig

K5 En spiller slår altid med to terninger på én gang

K6 En spiller rykker sin brik svarende til antal øjne på terningerne

K7 En grund kan ejes af en spiller eller af banken.

Felter

K8 Når start passeres, modtager spilleren 4000 kr.

K9 På "De fængsles"rykker spilleren til fængslet.

K10 På "Parkering"sker der ingenting.

K11 Skylder en spiller mere end spilleren ejer, udgår spilleren af spillet

Fængsel

K12 Betal 1000 kr inden terningkast for at komme ud.

Grunde

K13 Spilleren skal kunne købe den grund, spilleren lander på, hvis den ikke ejes allerede.

K14 Hvis en spiller lander på et ejet felt, skal spilleren betale leje til ejeren.

K15 Lejens pris fastsættes ud fra feltets skøde og bygninger samt pantsætningsstatus.

K16 Grundene tilhører en serie, som angives ved grundens farve.

Huse og hoteller

K17 En spiller kan opføre huse på en grund, når alle grunde i serien ejes.

K18 Når 4 huse ejes på en grund, kan et hotel tilkøbes. Hotellet erstatter husene.

K19 Det maksimale udviklingsniveau af en grund er et hotel.

B

Fængsel

K20 Er man i fængsel kan man ikke opkræve husleje

K21 Man kan komme ud af fængsel ved at benytte løsladelseskortet.

Huse og hoteller

K22 Huse og hoteller kan sælges til banken for halvdelen af deres værdi.

K23 Værdien af et hotel svarer til værdien af 5 huse.

Pantsætning

K24 En spiller kan pantsætte en grund hos banken. Spilleren modtager pansætningsbeløbet, men kan ikke modtage leje for grunden, når den er pansat.

K25 En pantsætning kan ophæves mod en ekstra betaling på 10% i rente.

Chancekort

K26 Spilleren skal kunne trække chancekort som indeholder forskellige ordrer.

K27 Et brugt chancekort lægges nederst i bunken efter brug.

Ejendomme

K28 Ejer man alle grundene i en serie, modtager man dobbelt leje af grundene, hvis de er ubebyggede.

Gæld

K29 Skylder en spillere mere spilleren ejer, skal spilleren forsøge at redde sig selv ved at sælge sine bygninger til banken. Lykkes det ikke, afleveres alt det spillerne ejer til banken og spilleren udgår af spillet.

C

Opstart

K30 * Spillerne kan selv vælge hvem der skal starte spillet.

K31 * Spillerne kan vælge at bruge en algoritme til at bestemme, hvem der starter spillet.

Slå to ens

K32 Slår man 2 ens, skal man slå en ekstra gang, efter at kravene for det felt man er landet på først er afviklet.

K33 * Slår man 2 ens, 3 gange i samme tur, ryger man direkte i fængsel.

Fængsel

K34 * Er man i fængsel kan man stadig købe grunde gennem intern handel.

K35 Har man 3 gange forsøgt at slå 2 ens uden held, betales bøden på 1000 kr. og spilleren forlader fængslet.

K36 Man kan komme ud af fængsel ved at kaste 2 ens. Man rykker derefter det antal frem man har slået, og får derudover et ekstra kast.

Indkomstskatte

K37 Spilleren vælger betalingsmåden for indkomstskatten inden værdierne tælles sammen.

K38 På "Indkomstskatten"betaler spilleren 4.000 kr. eller med 10% af sine værdier.

Chancekort

K39 * Alle chancekortene i Matador skal implementeres i spillet.

Intern handel

K40 * Spillere skal kunne handle internt med skøder og løsladelseskort.

K41 * For at kunne handle internt med sine skøder skal en grund være fri for huse og hoteller.

K42 * Spillere bestemmer selv prisen på grunde og løsladelseskort de sælger i interne handler.

K43 * Interne handler samt handler med banken må kun igangsættes af en spiller i starten af spillerens tur.

K44 * Er en spillers balance mindre end det beløb, spilleren skal betale, må spilleren undtagelsesvist igangsætte interne handler og sælge til banken.

Huse og hoteller

K45 Huse skal opføres jævnt på en serie, dvs. forskellen på udviklingsniveauet på to grunde i en serie på maks svare til et hus.

Gæld

K46 * Skylder en spillere mere end spilleren ejer, skal spilleren sælge sine bygninger til banken og derefter aflevere det resterende til sin kreditor. Derefter udgår spilleren af spillet.

Auktion

K47 * Er banken kreditor for en spiller, som går fallit, sættes spillerens grunde straks på auktion.

K48 * Ejendomme sættes på auktion hvis de ikke købes

K49 * Prisen af en grund på auktion starter på grundens pris hos banken.

K50 * Er man i fængsel kan man stadig købe på auktion.

Tekst og indlæsning

K51 Der skal laves flere tekstfiler, så spillerne kan vælge hvilket sprog de vil spille spillet.

2.2 Ikke-funktionelle krav

A

K52 Programmet skal reagere på inputs i løbet af 0.0 til 0.5 sekunder.

K53 Programmet skal kunne køre på DTU's databars computere

K54 Programmet skal køre på java 1.8.

K55 Alle tekstbeskeder skal indlæses fra en tekstfil på en måde, så det er nemt at videreudvikle programmet så det kan køre på et andet sprog.

K56 Data omkring de enkelte felter, f.eks. deres navne og pris skal indlæses fra en tekstfil.

2.3 Implementerede krav

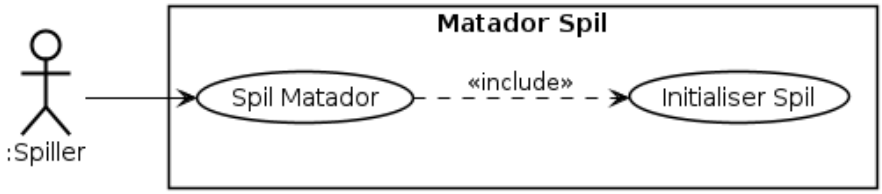
Kravlisten har vi lavet de første dage af projektet, da vi indledende formulerede vores krav. Vi har imidlertid arbejdet med løbende at redigere i kravene. Dette har vi gjort, fordi vores forståelse for projektet og de enkelte arbejdsopgaver i forbindelse med at implementere visse krav er blevet bedre undervejs.
Vi er endt med, som vi stærkt håbede på, at have implemeteret alle A-krav og alle B-krav. Det betyder at spillet fungerer på et høj niveau med både køb, salg og pantsætning af grunde og med køb og salg af huse samt et antal chancekort. Vi har derudover haft tid til at implemtentere et stort antal af C-kravene, men ikke dem alle.
Vi har ikke nået at implementere intern handel eller auktioner, og alle krav ifm. disse er altså ikke implementeret. Det er dog lykkedes at implementere muligheden for at skifte sprog i spillet ved at indlæse alle spillets beskeder fra tekstfiler.

2.4 Use case modellen

I vores use case model har vi valgt at betragte systemet med en use case, samt en include-use case. Disse to use cases ses i figur 4.

UC1: Spil Matador, er den mest centrale og beskriver som grundlæggende flowet af det fungerende spil. Denne use case er beskrevet på et fully dressed niveau.

UC2: Initialiser Spil beskriver hvordan brugerne kan opsætte spillet, så det er klar til at blive spillet. Denne er beskrevet på et casual niveau.



Figur 4: Use case diagram

2.4.1 Fully dressed beskrivelse af UC1 (Spil Matador)

Primær aktør:	Spiller
Forudsætninger:	Spillere skal være oprettet og have skrevet navn og valgt bil.
Success scenarie:	En vinder er fundet.

Main flow

1. <<Initialiser Spil>>
2. Det er første spillers tur.
3. Spilleren vælgere at slå med terningerne.
4. Spilleren lander på en ejendom uden ejer.
5. Spilleren køber grunden.
6. Spillerens tur afsluttes.
7. Det er nu næste spillers tur.

punkt 3 til 7 gentages indtil vinder er fundet.

Alternate flows

*0a På ethvert tidspunkt, hvis en spiller skal betale et større beløb end spillerens balance.

(a) Spilleren får mulighed for at handle med banken eller andre spillere for at betale beløbet.

(b) Lykkes det ikke at betale beløbet, går spilleren fallit.

(c) Den fallerede spiller overdrager alle sine værdier til kreditoren og udgår af spillet.

3a Spilleren vil bebygge en grund først.

(a) Hvis spilleren ejer alle grunde i serien, kan spilleren få lov at bygge. Ellers ikke.

(b) Spilleren betaler prisen for det antal huse eller det hotel, spilleren vil købe.

3b Spilleren vil handle med banken først.

(a) Hvis spilleren ønsker at pantsætte en grund, gøres det.

(b) Hvis spilleren ønsker at ophæve en pantsætning, gøres det.

(c) Hvis spilleren ønsker at sælge et antal huse eller et hotel, gøres det.

3c Spilleren vil udføre en handel først.

(a) Hvis en anden spiller ønsker at lave en handel med den aktive spiller, udføres handlen.

3d Spilleren er i fængsel.

(a) Hvis spilleren er i fængsel kan han enten vælge at betale 1000 kr, prøve at slå sig ud eller benytte sig at sit "løsladelseskort".

(b) Vælger spilleren at prøve at slå sig ud, skal spilleren slå 2 ens, for at dette lykkes.

(c) Lykkes det at slå sig ud, rykker spilleren det antal øjne som terningerne viser og spiller sin tur.

(d) Vælger spilleren at bruge "løsladelseskort", gøres dette.

(e) Vælger spilleren at betale, gøres dette.

(f) Spilleren ruller og flytter.

(g) Spillerens tur afsluttes.

4a Spilleren lander på "Parkering", "Start", "På besøg", "I fængsel"eller på en grund, spilleren selv ejer.

(a) Gå til 6.

4b Spilleren lander på en ejendom, som en anden ejer.

(a) Spilleren betaler leje til ejeren.

(b) Gå til 6.

4c Spilleren lander på "Prøv Lykken".

(a) Spilleren trækker et chancekort fra bunken og udfører handlingen på kortet

(b) Gå til 6.

4d Spilleren lander på "Gå i fængsel".

(a) Spilleren rykker direkte i fængsel og modtager ikke 4000 kroner for at passere start.

(b) Gå til 6.

4d Spilleren lander på "Indkomstskatten".

(a) Spilleren vælger betalingsformen for indkomstskatten.

(b) Har spilleren valgt at betale 4000 kr., sker dette.

(c) Har spilleren valgt at betale 10% af sine værdier, beregnes dette og betales.

(d) Gå til 6.

5a Spilleren ønsker ikke at købe grunden.

(a) Grunden sælges på auktion.

(b) Hvis ikke nogen ønsker at købe grunden, beholder banken den.

6a Spilleren har slået 2 ens.

(a) Hvis spilleren allerede har slået 2 ens tidligere i samme runde, ryger spilleren i fængsel.

(b) Spilleren får en ekstra tur og går til 3.

2.4.2 Casual beskrivelse af UC2 (Initialiser Spil)

Primær aktør:	Spiller
Forudsætninger:	Programmet er startet
Success scenarie:	Systemet ved hvilke spillere, der er med og rækkefølgen, turene skal forløbe i.

Main flow

1. Spilleren vælger antallet af spillere.
2. På skift angiver hver spiller nu sit navn og hvilken type bil, de vil spille med.
3. Når alle spillere har angivet bil og navn, kan spillet begynde.

Alternate flow

2a En spiller har angivet det samme navn som en allerede eksisterende spiller:

   (a) Spilleren bedes indtaste et nyt navn.

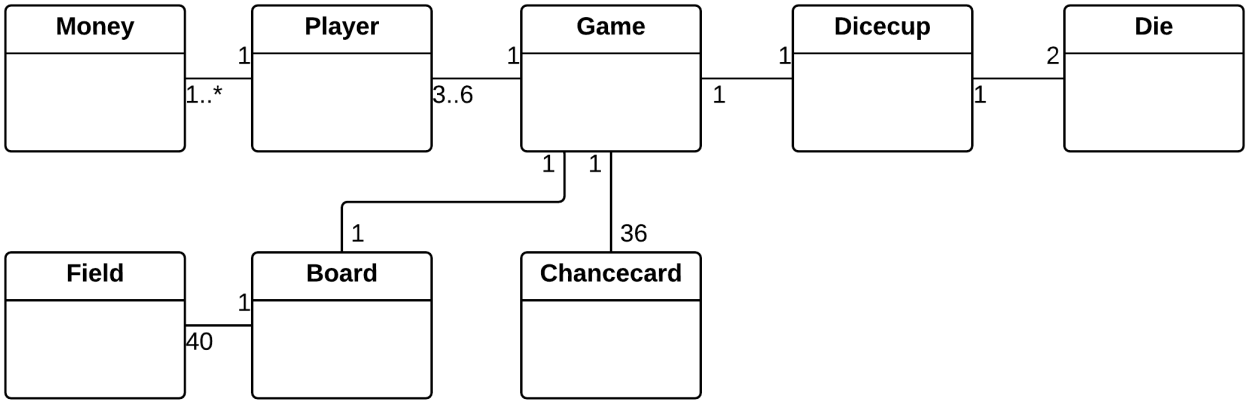
2b En spiller har angivet den samme bil som en eksisterende spiller.

   (a) Spilleren bedes vælge en ny bil.

3 Analyse

3.1 Domænemodel

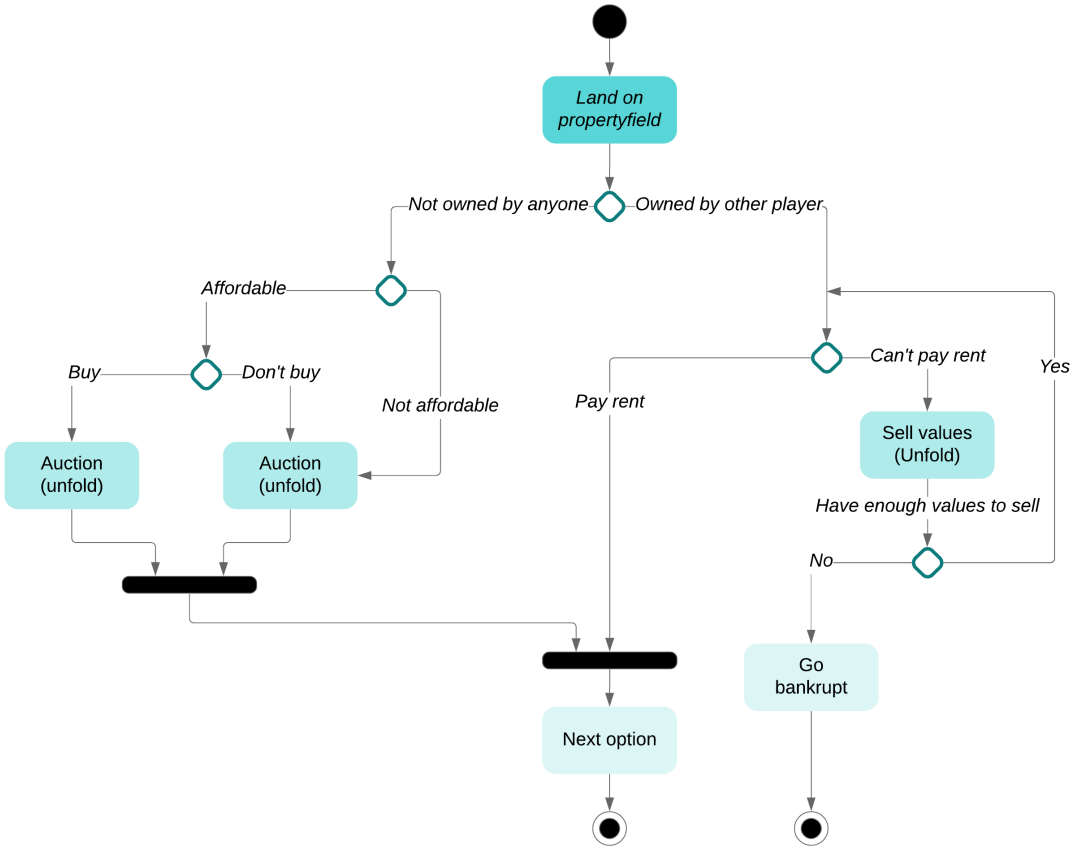
Vores domænemodel (Figur 5) afspejler det fysiske spil, Matador. Vi har derfor i domænemodellens klassediagram medtaget de "klasser", som et virkeligt matadorspil indeholder. Disse har vi kunne opstille på baggrund af vores kendskab til matadorspillet. Nogle af klasserne svarer direkte til genstande, som spillets æske indeholder: Et bræt med felter, to terninger, en bunke chancekort og pengesedler. Mange af klassernes navne svarer til de navneord, som optræder i vores use case tekstbeskrivelser. Ordene er blevet oversat til engelsk for at gøre overgangen til klassediagrammet, og derefter til koden, nemmere.



Figur 5: Domænemodel

3.2 Aktivitetsdiagram

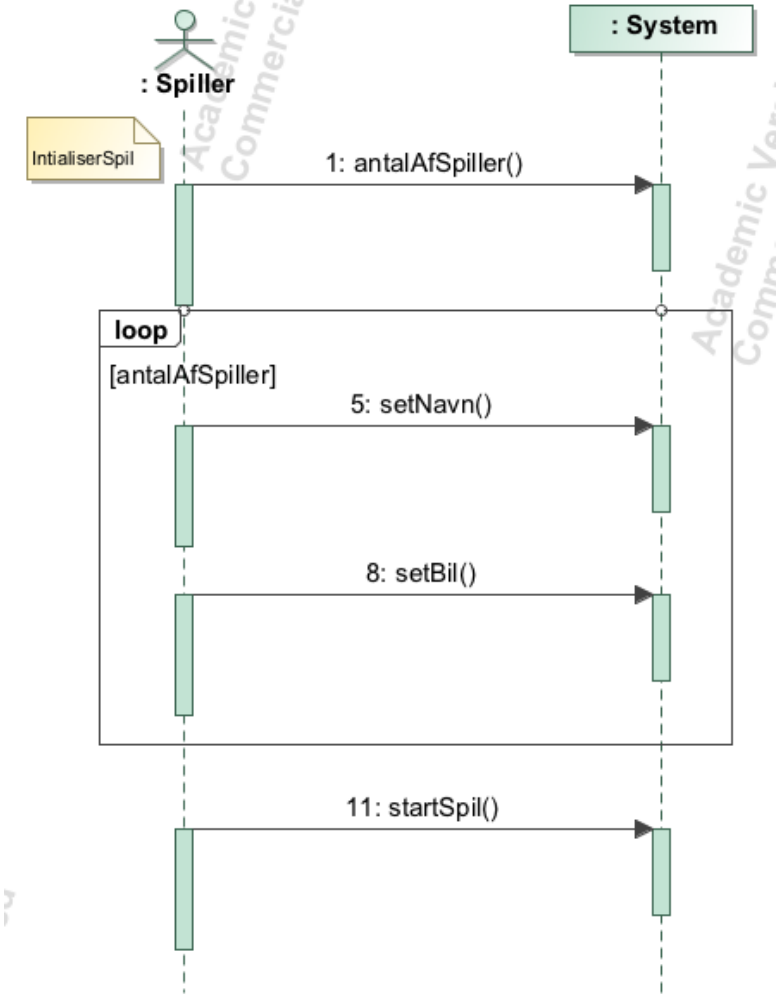
Figur 6 er et aktivitetsdiagram, der beskriver hvad der sker i Matador, når en spiller lander på et ejendomsfelt. Diagrammet har vi lavet for at forstå forløbet. Det kan bruges som inspiration, når funktionaliteten for det forløb skal implementeres i programmet. Bemærk at diagrammet er lavet med det udgangspunkt, at alle krav når at blive klaret.



Figur 6: Aktivitetsdiagram over forløbet, når en spiller lander på en ejendom.

3.3 Systemsekvens-digram

Vi laver et systemsekvensdigram, hvor vi beskriver use case (UC2 - IntialiserSpil), som beskrives interaktionen mellem spilleren og systemet. Her forklares hvad der sker skridt for skridt når UC2 bliver kørt.



Figur 7: System sekvens diagram

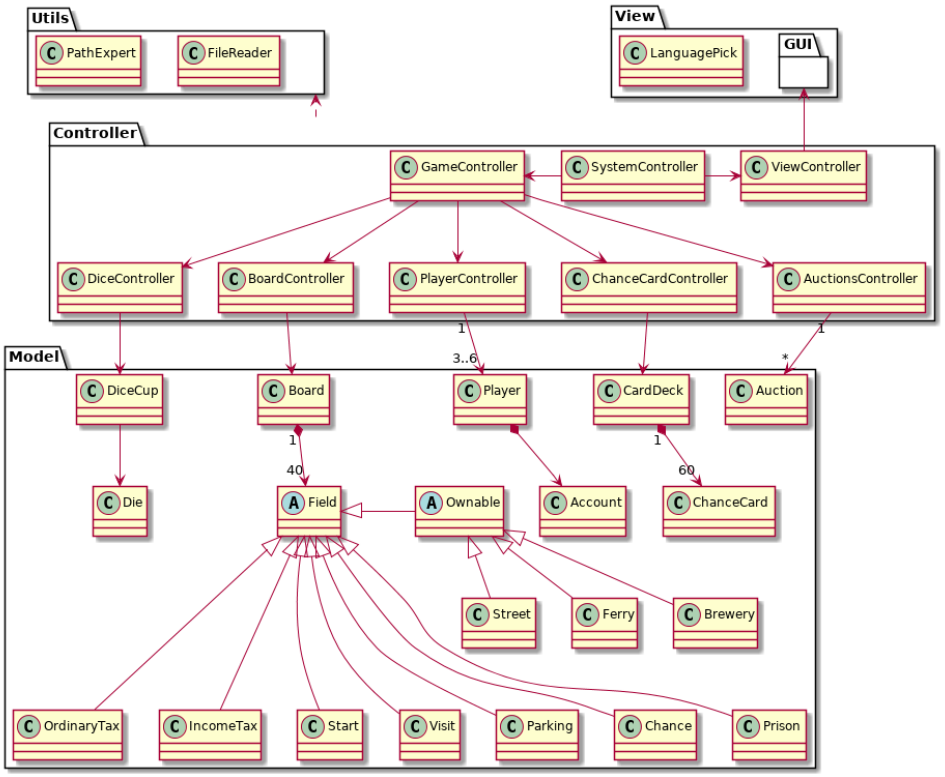
4 Design

4.1 Klasse diagram

I figur 8 ses vores færdige klassediagram i en forsimplet version. Vi har bygget vores Matadorspil op ved brug af MVC[1]. Derfor har vi gjort brug af Controller klasser som bestemmer spillets logik og model klasser som indeholder spillets objekter og en masse atributter. View klasserne er allerede defineret i GUI'en og derfor har vi ikke selv defineret nogen klasser her, på nær LanguagePick som bruges ved spillets opstart til at vælge sproget. Dette kan ses i figur 9. Klassediagrammet i figur 8 indeholder også pakken Utilities, der ses tydligere med metodenavne i figur 10. Denne pakke indeholder to klasser, hhv. FileReader og PathExpert.

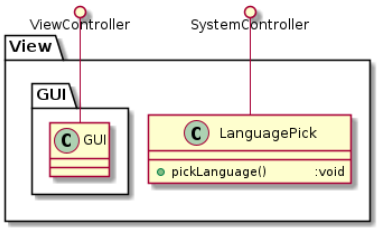
Arkitekturen er lavet på en måde, så det eneste bindeled mellem view-laget og model-laget sker gennem Controller-laget, specifikt gennem SystemControlleren. Dette gør vi for at sikre at alle klassernes ansvarsområder er veldefinerede. Vi vil altså ikke f.eks. have en Player-klasse, der har ansvar for at vise spillerens brik på brættet, da dette hører til view-laget. Desuden sikrer det lav kobling, og gør det langt nemmere, hvis vi f.eks. ønsker at videreudvikle spillet en dag eller bruge en anden GUI.

Model-laget i klassediagrammet har mange ligheder med vores domæne-klassediagram, hvilket er naturligt. Der er dog nogle afvigelser derfra, f.eks. har vi i software-klassediagrammet ingen Bank, idet det er vores program, der skal agere bank. Der er til gengæld et stort antal klasser, som nedarves fra Field, idet et de individuelle felter skal implementeres på vidt forskellige måder. Vi benytter her nedarving til at simplificere koden og lettere få et overblik over de forskellige felt-typers forskelle og ligheder.

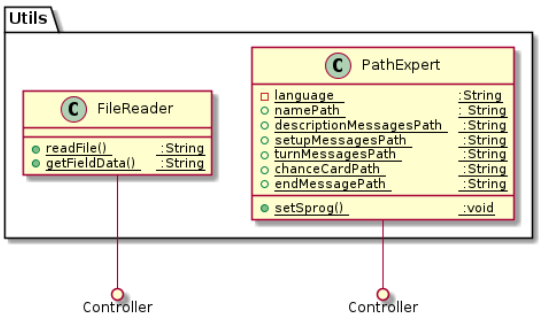


Figur 8: Klasse diagrammet er delt på i 3 lag M, V og C.

[1]Model View Controller Pattern



Figur 9: Viewdiagrammet viser GUIen som vi har fået udleveret. Derudover har vi også vores egen Java Swing gui klasse deri



Figur 10: Utilities indeholder vores FileReader[2], som vi bruger til at indlæse tekstfiler med samt PathExpert, der bruge til at referere til tekstfilernes placering i mappestrukturen.

4.2 Vores brug af Designmønstre

I vores design har vi forsøgt at implementere et antal GRASP-mønstre. GRASP er en samling af principper og guidelines, der skal hjælpe til at lave ansvarsfordelingen af ens klasser. Vi vil nu gennemgå vores design med henblik på hvilke af mønstrene, vi har forsøgt at implementere.

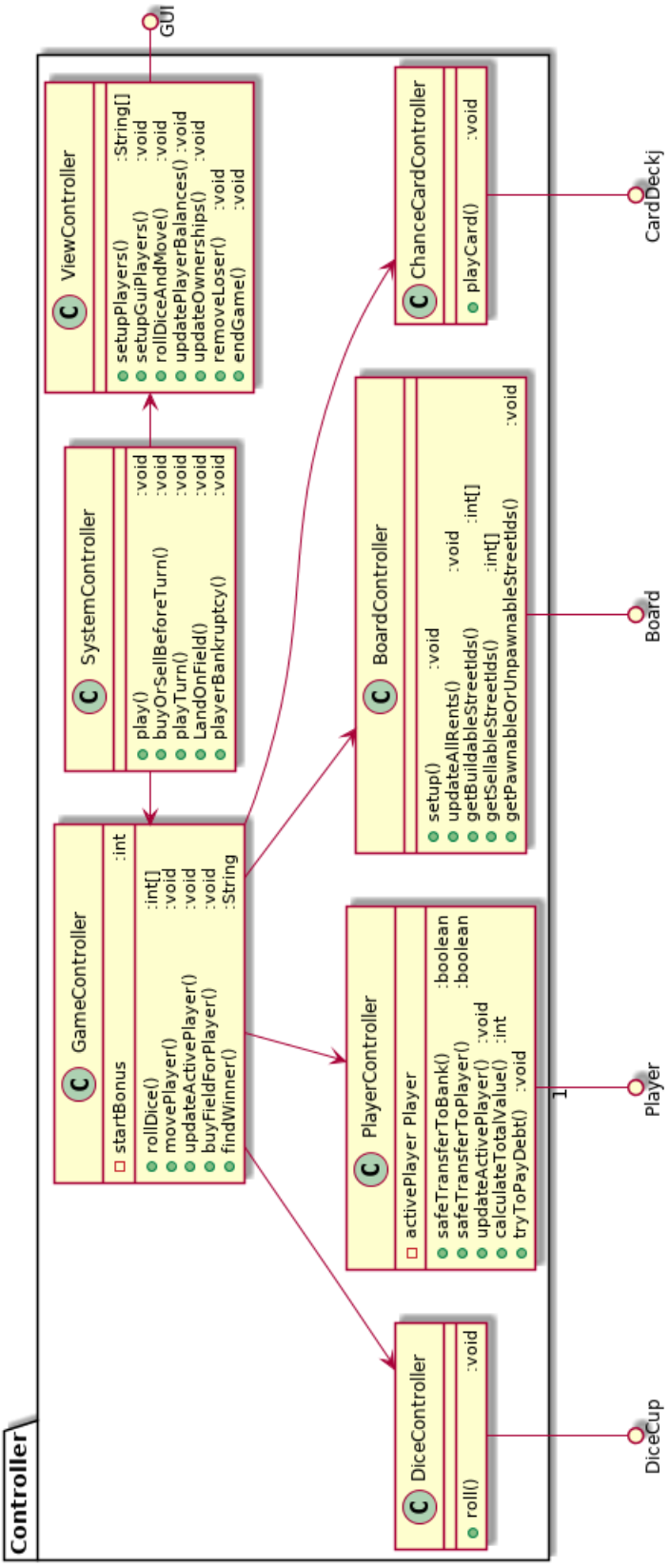
4.2.1 High cohesion

Høj samhørighed betyder, at klasserne skal have veldefinerede ansvarsområder, og disse ansvarsområder skal overholdes.
I vores kode har vi forsøgt at opnå høj samhørighed til alle klasserne. Et eksempel kunne være ViewController-klassen. I figur 11 ses et overblik over controller-klassernes vigtige metoder og attributter, og ViewControlleren er den eneste, der indeholder metoder, der benytter har GUI-en. Ideen er også, at de enkelte controllere har ansvarsområdet for de model-klasser, de tilhører. Eksempelvis skal ChanceCardControlleren gerne håndtere, hvordan chancekortene skal bruges i spillet.

Flere steder giver dette imidlertid store udfordringer. Eksempelvis har det vist sig at være svært at håndtere logikken for alle chancekort i ChanceCardControlleren, idet mange chancekort involverer at spillere bevæger sig rundt på brættet. Brættet er derimod BoardControllerens ansvar, og vi har derfor håndteret disse chancekort i GameControlleren, som har kendskab til både ChanceCardControlleren og til BoardController.

Netop for at klare denne slags problemer har vi valgt at have en GameController, som har det overordnede ansvar for modellaget, og kan klare de opgaver, som kræver flere forskellige dele af model-lagets controllere, men ikke brugerinput fra ViewControlleren. Dette gjorde vi med henblik på at aflaste SystemControlleren, som ellers ville få et meget stort ansvarsområde. I praksis har det vist sig, at der ikke kan ligge ret mange metoder i GameControlleren, idet de fleste funktionaliter i spillet kræver en form for brugerinput fra ViewControlleren. Det kan derfor diskuteres, om det havde været lettere at udelade en GameController i designet.

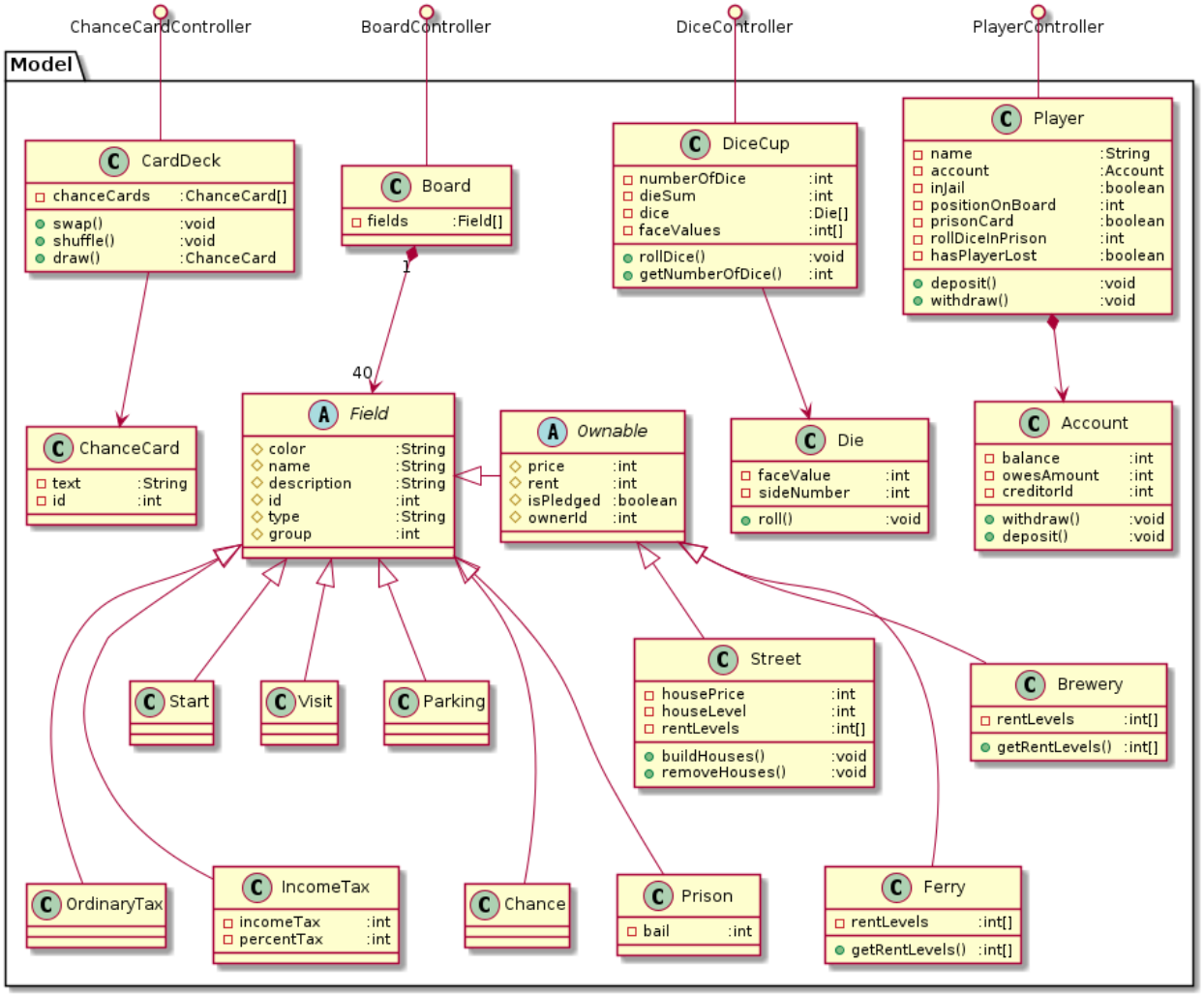
[2]FileReader metoder er beskrevet på side 25



Figur 11: Controllerdiagrammet viser pakken med controller klasserne som styrer model klasserne. mange af vore controllere fungerer også som creatorer for modellerne. En version af dette diagram kan ses i bilaget figur

4.2.2 Low coupling

Lav kobling er et mønster, der dækker over, at klasserne er mindst muligt afhængige af hinanden, og af færrest mulige andre klasser. Dette gør man bl.a. for at reducere virkningen af ændringer, og gøre det lettere at udskifte eller udvikle dele af projektet.



Figur 12: Modeldiagrammet viser alle klasserne i modellaget i systemet. Mode-klassernes attributter indeholder de vigtigste data for systemets tilstand.

Vi har for eksempel forsøgt at opnå lav kopling ved kun at lade klassen BoardController have et objekt af typen Board. Det betyder, at hvis boardet skulle udbygges eller ændres, skulle vi i princippet bare have lavet ændringen i Board og i BoardController. Det hænger også sammen med at opnå høj samhørighed, idet det kun bør være BoardControllerens ansvar at hente information fra Boardet, hvori alle Fields er gemt.

En stor udfordring vi har haft en denne forbindelse var at ViewControlleren skal kunne opdatere alle sine felter med metoden updateOwnerships(), og dette skal gøres ud fra den information som ligger i Boardets array af Fields. Vi har derfor valgt, at ViewControllerens konstruktør og updateOwnerships() skal tage et Board-objekt som argument, så den kan tilgå alle felterne. Strengt taget bør der altså være en dependency fra ViewController til Board på klassediagrammet, som vi dog har udeladt for ikke at gøre diagrammet uoverskueligt.
Det strider også mod Model-View-Controller mønstret, som vi ellers har forsøgt at opnå ved kun at lade SystemControlleren håndtere kommunikation mellem viewlaget og modellaget.

Skulle vi have gjort det anderledes, skulle SystemControlleren altså have hentet alle informationene fra brættet og givet disse til ViewControlleren som simple variable, når brættet oprettes og opdateres. Dette ville imidlertid have været en stor omvej for vores projekt.

4.2.3 Controller

Controller-pattern er et mønster vi har gjort meget for at implementere. En central måde, vi har implemeteret det i vores design, er SystemControllerens ansvar. Den er nemlig ansvarlig for at modtage ordrer fra brugeren gennem UI, og videregive informationen om brugerens valg til modellaget, så de korrekte sekvenser sættes i gang. F.eks. den centrale metode play() i SystemControlleren, se figur 19, beder spilleren træffe flere valg gennem UI-laget, f.eks. om spilleren vil købe noget inden sin tur. Systemcontrolleren håndterer så valgene om, hvilke muligheder spilleren skal have for køb, og gennemfører købet i modellaget, hvis spilleren ønsker det.

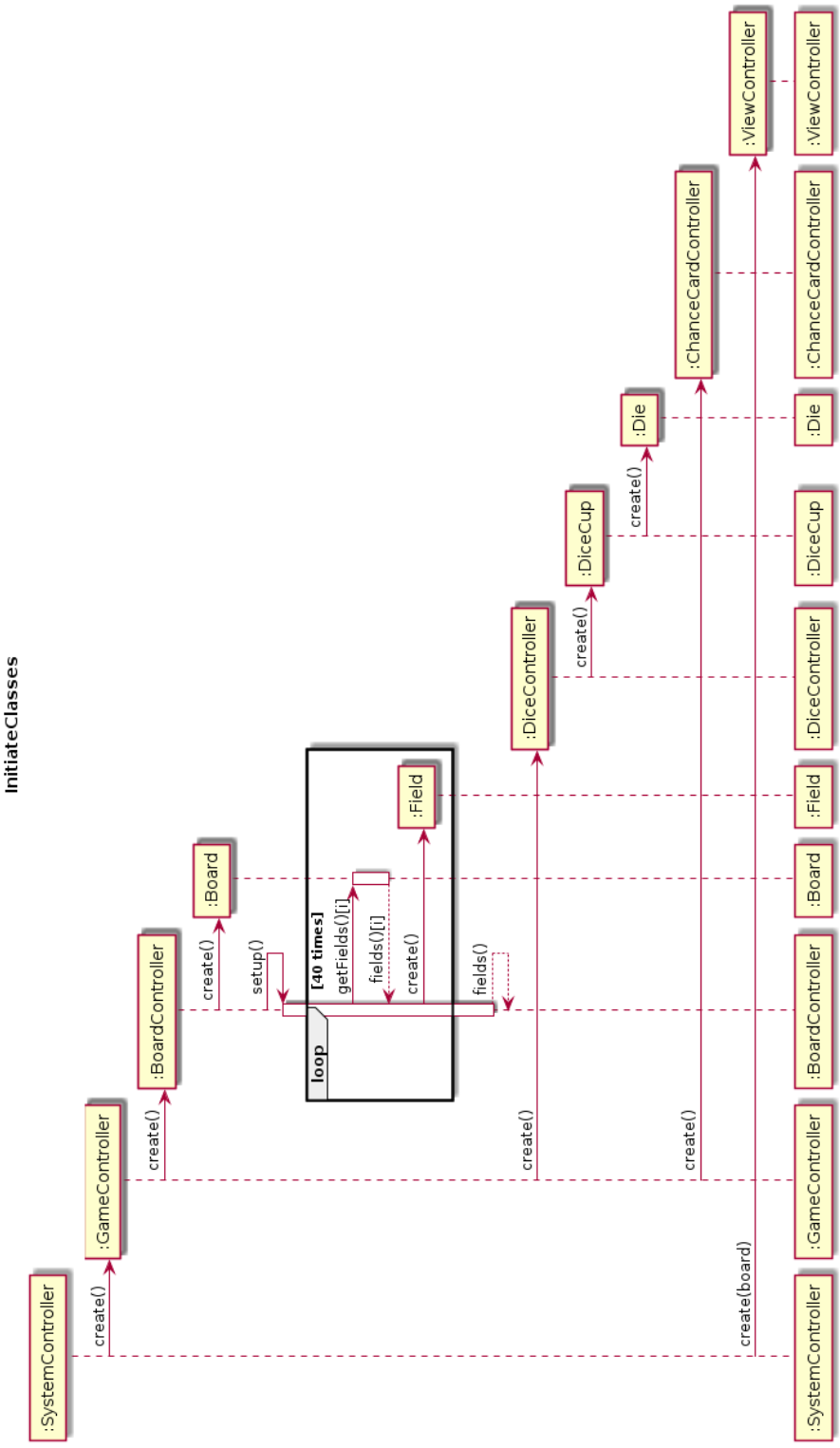
En anden central måde, vi har brugt controller-pattern på, er ved at tildele ansvaret for håndteringen af de centrale model-klasser til deres egen særlige controller-klasse. I figur 12 ses det at de 4 modelklasser, CardDeck, Board, DiceCup og Player alle bliver håndteret af deres respektive controllere. Som eksempel har vi lagt ansvaret for at holde styr på, hvem den aktive spiller er hos PlayerControlleren, da vi ikke synes den enkelte spiller skal have ansvar for at vide hvem den næste spiller bliver, og videregive turen.
På samme måde har vi i BoardControlleren en række metoder, eksempelvis getBuildableStreetIds(), der laver nogle logiske beregninger på boardet, som vi ikke synes Boardet selv skal have ansvar for. Grundlæggende synes vi, Board skal indeholde felterne, og de enkelte felter skal kende til deres egen data. Hvilke gader en bestemt spiller kan bygge på kræver kendskab til alle felterne i serien og deres huse samt til reglerne for udbygning af huse. Det ansvar har vi altså lagt hos BoardControlleren.
En udfordring i den forbindelse er, at Board så stort set er frataget alt sit ansvar, og vi kunne i retroperspektiv måske have udeladt den klasse helt. At vi alligevel har beholdt Board skyldes primært, at vi gerne vil se en klar sammenhæng mellem vores domain-klassediagram og vores domain-lag i software-klassediagrammet.

4.2.4 Creator

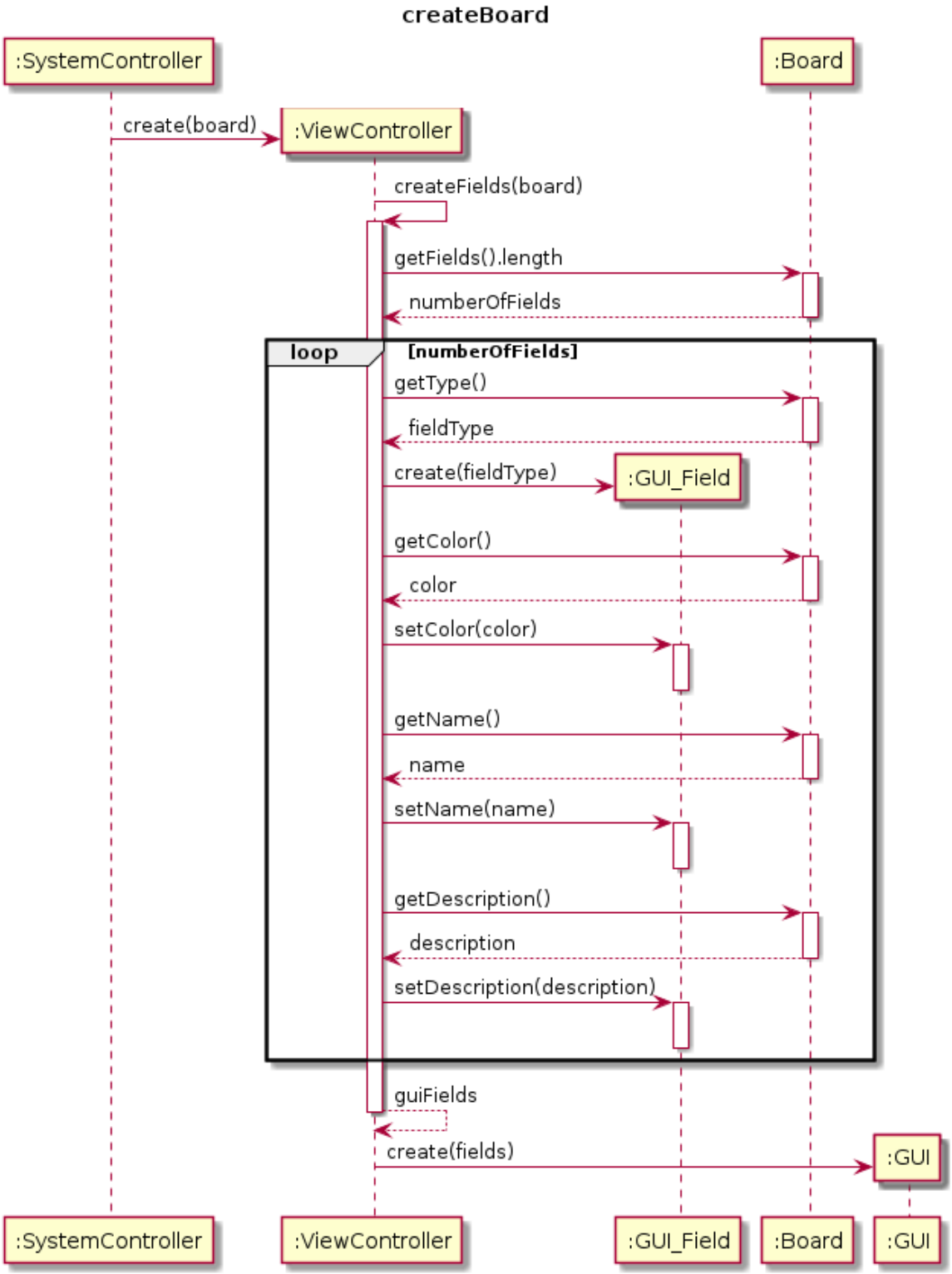
Vi har også forsøgt at implemetere Creator-pattern ved at være opmærksomme på hvilke klasser, der laver instanser af andre klasser. Eksempelvis laver DiceControlleren en DiceCup, og en DiceCup laver instanser af Die. Et andet oplagt sted, vi kunne have brugt mønstret er, at alle Fields kunne være oprettet af Board. Her har vi imidlertid valgt at afvige fra mønstret, da Fields bliver instantieret i BoardControlleren. Vi har valgt at gøre det sådan, fordi vi synes det er mere logisk at controllerne skal være ansvarlige for at importere data fra vores ressourcer, og disse skulle bruges i feltenes konstruktør.

4.3 Sekvens diagrammer

Her har vi Sekvens diagrammer over 3 forskellige sekvenser, hhv. hvordan vores vigtigste controller-objekter og model-objekter oprettes, hvordan Boardet oprettes og hvordan en almindelig tur spilles. Sekvensdiagrammerne giver et overblik over, hvordan grupper af objekter arbejder sammen.

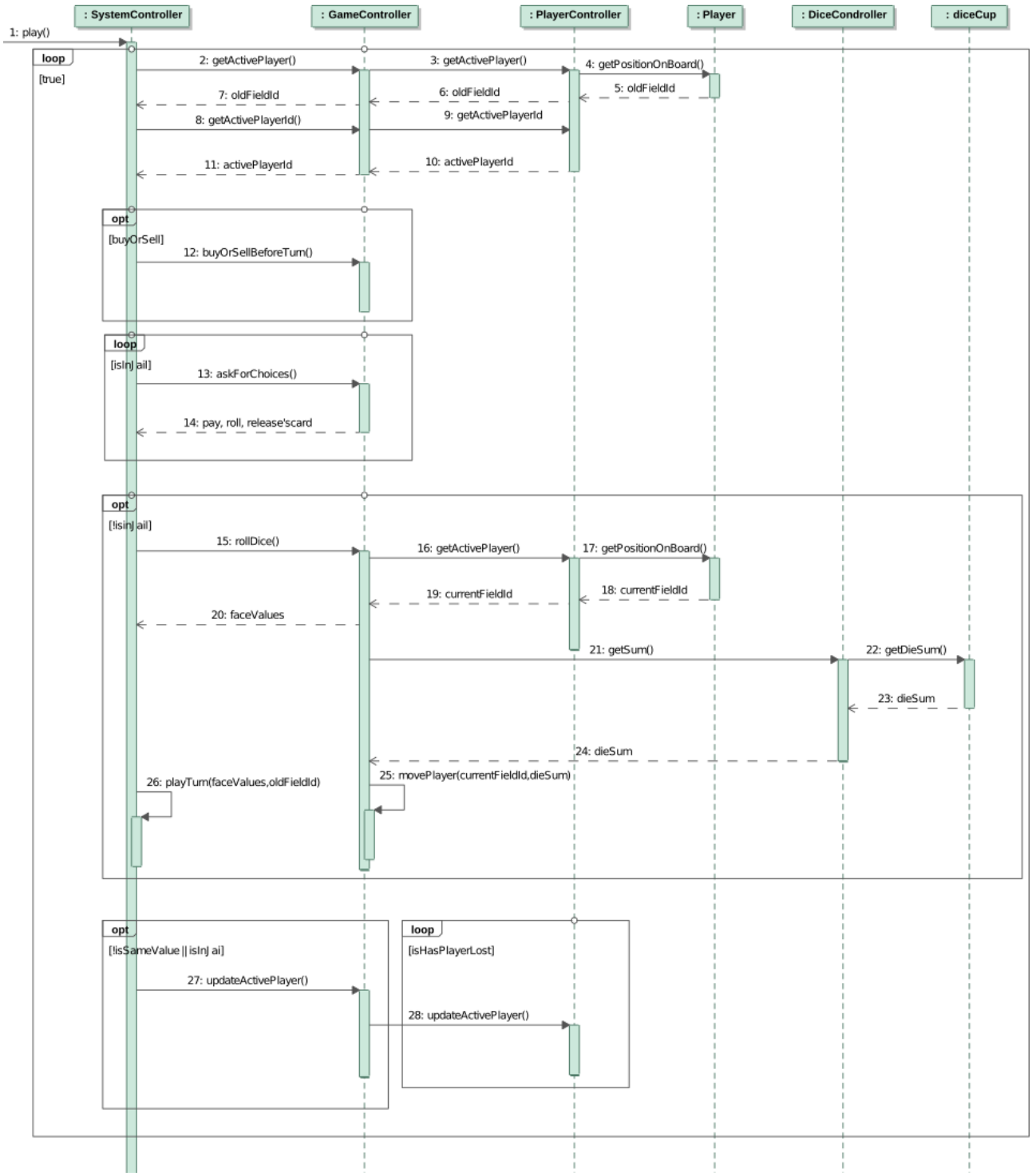


Figur 13: Sekvens diagram over de ting som sker lige efter at en spiller har valgt sprog i begyndelsen af programmet. En masse instanser af klasser bliver oprettet i denne del.



Figur 14: Sekvens diagram over gui boardet som bliver initialiseret.

Vi har lavet en metode i vores viewcontroller som i et for loop med en switch case som tjekker typen af felterne på boardet og felternes navn o.s.v. fra vores model board lag og derudfra generere et GUI board. Dette gør at man kan scale sit board op og ned som man har lyst til i model laget og at view laget bare tilpasser sig derefter.



Figur 15: I diagrammet ovenfor ses et sekvens diagram over spiller tur forløb.

1. Giv spilleren muligheden for at købe og sægle
2. Hvis Spilleren er i fængsel, så vis muligheder for at komme ud.
3. Rul terninger.
4. Flyt spiller.
5. Opdater spiller

5 Implementering

Følgende afsnit kan bruges som hjælp til bedre forståelse af hvad der sker i hvilke klasser. De vigtigste og mere komplicerede metoder fremhæves og beskrives i dybden her. Vi har valgt at beskrive de vigtigste klasser i vores program, da en fuldstændig gennemgang vil være uoverskuelig for læseren. Af samme grund lægger vi primært vægt på pakken `controllers`, hvor den fundamentale spillogik ligger.

5.1 Filereader

Klassen filereader indeholder 2 metoder som tager sig af læsning af filer. Vi bruger hovedsageligt læsning af filer til indstilling af sprog, men også til at indstille forskellige felter i med ting som, id, pris, leje og andre attrubutter at typen int.

5.1.1 readFile metoden

```
1  public static String readFile(String file, String keyword){
2      try {
3          String line;
4          BufferedReader reader = new BufferedReader(
5              new InputStreamReader(
6                  ClassLoader.getSystemResourceAsStream(file),
7                  StandardCharsets.UTF_8
8              )
9          );
10         while ((line = reader.readLine()) != null)
11         {
12             String[*/* parts = line.split(":", 2);
13             if (parts.length >= 2)
14             {
15                 String key = parts[0*/*;
16                 String value = parts[1*/*;
17                 if(keyword.equals(key)){
18                     return value;
19                 }
20             }
21         }
22     } catch (IOException e) {
23         e.printStackTrace();
24     }
25     return null;
26 }
```

ReadFile metoden bliver brugt til at indlæse Strings fra tekstfiler. Metoden er statisk da den bliver kaldt fra mange andre klasser i systemet. I metoden gør vi brug af BufferedReader som vi får til at tage vores resource filer som parameter ved hjælp af InputStreamReader og getSystemResourceAsStream.

Vi har skrevet readFile metoden således at den læser hver linje i den fil den bliver bedt om. Derudover leder metoden efter et kolon i hver linje som den vil bruge til at dele linjen op i et array som er 2 langt den første den er key og den anden er value. Hvis metoden finder en linje som er under 2 langt så sker der ikke noget.

Ud over at tage en fil fra ressource, tager metoden også et keyword, dette keyword bruger metoden til at finde en value ved hjælp af key/value arrayet som metoder generere fra filen. hvis metoden finder den søgte key, så returnere den den value som hører til den samme linje.

5.1.2 getFieldData metoden

```
public static int getFieldData(String state, String attribute){
    String line = readFile(fieldAttributesPath,state);
    String[*/* stringData = line.split(",");
    int[*/* intData = new int[stringData.length*/*;
    for (int i=0;i<stringData.length;i++){
        intData[i*/* = Integer.parseInt(stringData[i*/*);
    }
    String[*/* choices = {
        "id","price","rent","houseprice","house1",
        "house2","house3","house4","house5","series"
    };
    int number = 0;
    for (int i = 0; i < intData.length; i++) {
        if (choices[i*/*.equals(attribute)){
            number = intData[i*/*;
        }
    }
    return number;
}
```

Ud over at læse filer som kan returnere Strings, så har vi også brug for at indlæse heltals værdier fra filer, dette har vi fundet ud af at vi kan gøre med de selv samme keywords som vi bruger til at læse Strings fra filer.

Denne metode gør brug af readFile metoden og modtager en komma seppereret String. I hver kolonne, (mellem kommaer) ligger der en talværdi som er relevant for det tilhørende keyword. Dette er enten id, pris leje o.s.v.

Metoden tager imod det samme keyword som readFile metoden, men denne metode til altid kigge i den samme fil, nemlig fieldAttributes filen, som er deklæret i PathExpert klassen. Metoden gør lige som readFile metoden. den deler en String op i et array ved hjælp af komma seperation. Metoden tager den også en attribut som parameter, som den bruger til at vide om den skal returnere et id eller en pris o.s.v.

Grunden til at dette er smart er fordi at vi i vores tekst filer kan oprette en masse felter som hver har et navn en beskrivelse og en masse andre heltals værdier. Alle disse værdier kan vi bare skrive ind i nogen separate tekstfiler med et keyword, og når vi så skal oprette vores felter i programmet så behøver vi ikke at indlæse dem med andet end et state som ses på næste side i eksemplet over et ownable field.

5.2 Model - Felter generelt

I modellaget har vi i alt 12 klasser, som har at gøre med felter. Det skyldes at der i matador er mange slags felter, men mange af disse deler nogle egenskaber. Vi har derfor opbygget strukturen således, at alle felter nedarver fra klassen Field.

5.2.1 Field

Field-klassen er abstrakt og indeholder de vigtige attributter, som alle typer af felter har. Dvs. bla. en String name, String description, og en String type. Type er en variabel, der grundlæggende fortæller hvilken af Fields childclasses et objekt tilhører [3]. De fleste typer felter nedarver direkte fra Field, f.eks. Start, IncomeTax og Chance.

5.2.2 Ownable

Ownable er endnu en abstrakt klasse. Den nedarver fra Field og har 3 childclasses, der er de tre typer felter, en spiller kan eje, hhv. Street, Ferry og Brewery. Dette skyldes at de alle deler et antal attributter, hhv. int price, int rent, boolean isPledged og int ownerId.

```
1   public abstract class Ownable extends Field {
2   protected int price;
3   protected int rent;
4   protected boolean isPledged;
5   protected int ownerId = -1;
6
7   public Ownable(String state) {
8       super(state);
9       this.price = getFieldData(state,"price");
10      this.rent = getFieldData(state,"rent");
11  }
12 }
```

Listing 1: Ownable

Ovenstående kodeeksempel er taget direkte fra klassen Ownable. Det ses at Ownable er defineret som en abstrakt klasse, idet der ikke skal eksistere nogle instanser af Ownable - dens formål er blot at andre klasser kan nedarve fra den. Derfor er dens attributter også deklereret protected og ikke private som normalt. Det sikrer, at dens childclasses kan tilgå variablene. Ownable nedarver selv fra Field, og dens konstruktør benytter derfor metodekaldet "super()", der kalder på konstruktøren fra moderklassen. Det ses, at argumentet til Fields konstruktør (som her hedder super()) er en String state. Dette er det keyword, vi benytter til vores FileReader, som indlæser data fra vores tekstfiler. Resten af konstruktøren definerer de variable, som Ownables har, der ikke indeholdes i Field. Bemærk at isPledged pr. default bliver false, og derfor ikke medtages i konstruktøren. Ownables metoder, som udelukkende er getters og setters er ikke medtaget i eksemplet.

5.3 Controllers

5.3.1 Systemcontroller

SystemControlleren er den overordnede controller for hele systemet. Dens opgave er at styre alle processer gennem den andre controllere. Den er desuden det eneste bindeled mellem viewlaget og modellaget. Kort sagt kan man derfor sige, at den konstant laver ændringer i modellaget og opdaterer viewlaget.

[3] Vi har senere lært at operatorern "instanceof"kunne have sparet os have benytte denne attribut.

- **SystemController()**
  (Klassens konstruktør)
  Opretter model-laget (gamecontrolleren) og view-laget (viewcontrolleren). Kalder metoden setupPlayers() på viewcontrolleren og bruger navnene som denne returnerer til at oprette PlayerControlleren. Kalder til sidst play().

- **play()**
  Kalder i et uendeligt while-loop metoden rollDice() på gamecontrolleren og derefter playTurn() efter, der er taget hensyn til, om spilleren ønnsker at købe eller sælge noget, og om spilleren er i fængsel. Hvis spilleren er i fængsel, tages der højde for hvordan han ønsker at komme ud, og om det lykkedes.

- **playTurn()**
  Parametre: int[] faceValues, int oldFieldId
  Beder ViewControlleren opdaterer spillerens position og alle spilleres balancer og kalder metoden landOnField().

- **landOnField()**
  Henter fra PlayerController den aktive spillers position på brættet og den tilhørende felttype. Indeholder en switchCase, der skifter på felttypen, og udspiller handingen på feltet. Hvis det er et Ownable-felt kaldes hjælpemetoden playPropertyField(), og på et chancefelt kaldes hjælpemetoden playChanceCard().

- **playPropertyField()**
  I figur 6 vises flowet af metoden på domain-niveau, som vi forstod det inden vi implementerede. Kort sagt håndterer metoden om spilleren selv ejer feltet, eller om en anden ejer det, og håndterer betaling og hvorvidt spilleren vil købe feltet. Der tages højde for, om feltet er pantsat, og om ejeren er i fængsel.

- **playChanceCard()**
  Kalder metoden draw() på ChanceCardControlleren og henter id'et af dette kort. Beder ViewControlleren vise kortets tekst. Herefter kan der ske 2 forskellige ting: Hvis det er et simpelt kort, der ikke kræver bevægelse på brættet kaldes metoden playCard() på ChanceCardControlleren. Ved kort, der kræver bevægelse på brættet, håndterer SystemControlleren selv eksekveringen af kortet. I alle tilfælde opdateres ViewControlleren til sidst.

- **playerBankruptcy()**
  Parameter: int playerId
  Kalder hjælpemetoderne sellHouses() og pawnStreets(), indtil spilleren enten har betalt sin gæld eller tabt. Hvis spilleren taber, kaldes hjælpemetoden looserSituation().

- **looserSituation()**
  Parameter: int playerId
  Kalder hjælpemetoden accountReset() på PlayerControlleren for at markere spillerne som taber. Dernæst kaldes makeFieldsFree() på gamecontrolleren for den tabende spiller. Sidst kaldes removeLoser() på ViewControlleren.

- **buyHouses()**
  Kalder metoden getBuildAbleStreetIds() på BoardControlleren. Hvis listen ikke indeholder noget, viser ViewControlleren en besked om, at der ikke kan bygges. Ellers får spilleren en drop-ned menu, og kan vælge hvilken af de mulige grunde, spilleren vil bygge et hus på. Menuen har også muligheden "Afslut", som ikke køber noget hus. Har spilleren

valgt at købe et hus, trækkes det korrekte beløb, og huset bygges gennem hjælpemetoden tryToBuyHouses() på GameControlleren.

- **sellHouses()**
  Fungerer som buyHouses(), men blot med salg af huse. Metoden getSellableStreetIds, som returnerer en liste over grunde med huse som kan sælges, kaldes på BoardControlleren.

- **pawnStreets()**
  Kalder metoden getPawnableOrUnpawnableStreetIds() på BoardControlleren med parametren "pawnable == true". Tilbyder spillere at pantsætte de mulige grunde, hvis der er nogen. Kalder hjælpemetoden pawnStreet() på GameControlleren.

- **unPawnStreets()**
  Kalder metoden getPawnableOrUnpawnableStreetIds() med parametren "pawnable == false". Tilbyder spillere at ophæve pantsætninger på de mulige grunde, hvis der er nogen. Kalder hjælpemetoden unpawnStreet() på GameControlleren.

5.3.2 Gamecontroller

GameControlleren har det overordnede ansvar for alle modellagets controllere. Dens opgave er at aflaste SystemControlleren fra de opgaver, som ikke kræver nogen form for brugerinput, dvs. kun omhandler modellaget. I praksis involverer de fleste opgaver en form for brugerinput, og GameControlleren er derfor ikke så stor.

- **rollDice()**
  Henter den aktiver spiller fra PlayerControlleren, beder DiceControlleren om at rulle terningerne og flytter spillerens placering i modellaget. Der tages højde for, om spilleren skal modtage startbonus. Returnerer et array af integers med terningernes værdier, så SystemControlleren, der kalder denne metode, kan opdatere view-laget.

- **payOrdinaryTax()**
  Betaler indkomstskatten for den aktive spiller, som hentes fra PlayerControlleren.

- **payIncomeTax()**
  Betaler den ekstraordinære statsskat for den aktive spiller, som hentes fra PlayerControlleren. Spilleren har valget mellem at betale enten 4.000 kr eller 10% af dens samlede værdier.

- **buyFieldForPlayer()**
  Henter den aktive spillers id og placering fra playercontrolleren. Trækker grundens pris fra spilleren og opdaterer grundens ejer.

- **newPos()**
  En switchcase til de chancekort, hvor spilleren rykker på brættet, som også behøver at snakke med viewlayer. Den benytter hjælpemetoden chanceSum() som udregner hvor mange felter en spiller skal rykke.

- **findWinner()**
  Beregner hvem der har vundet som den sidste spiller, der ikke har tabt. Returnerer navnet af den vindende spiller.

- **makeFieldsFree()**
  Parameter: int playerId Sætter ejeren af alle spillerens felter til at være banken, og ophæver alle spillerens pantsætninger.

- **pawnStreet()**
  Parameter: int playerId, int FieldId
  Giver spilleren pantsætningsværdien, og sætter gaden til at være pantsat.

- **unpawnStreet()**
  Parameter: int playerId, int FieldId
  Hæver pantsætningsværdien +10% fra spilleren og ophæver pantsætningen af gaden

5.3.3 Viewcontroller

ViewControllerens opgave er at opsætte og opdatere alt det som spillerne kan se, dvs. brættet, brikker, chancekort og balancer. Næsten alle ViewControllerens metoder kræver ad den ene eller den anden vej en form for input fra modellaget, da det er i modellaget, af al data ligger.

- **Viewcontroller()**
  (Klassens konstruktør)
  Parametre: Board board
  Konstruktøren tager board fra modellaget som parameter, og bygger gui-brættet ud fra det. Alle felterne bliver lavet med de korrekte typer og farver. Ud fra felterne oprettes gui-objektet, og det kommer frem på skærmen for spillerne.

- **setupPlayers()**
  Spillerne bliver på skift bedt om at indtaste deres navn. Det tjekkes om navnene er OK, og et String array af navnene returneres. Til sidst kaldes på hjælpemetoden setupGuiPlayers().

- **setupGuiPlayers()**
  Denne metode sætter spillernes brikker på brættet. De bliver bedt om selv at vælge designet på deres bil.

- **rollDiceAndMove()**
  Parametre: int[] faceValues, int sum, int activePlayerId, int oldFieldId
  Terningerne er blevet rullet i modellaget. Denne metode opdaterer blot hvad terningerne på brættet viser, og flytter den aktive spillers bil det rigtige antal felter.

- **updatePlayerBalances()**
  Parametre: int[] playerBalances
  Opdaterer gui-spillerenes balance, så den korrekte balance kan ses på brættet.

- **updateOwnerships()**
  Parametre: Board board
  Opdaterer gui-brættet, så det tilsvarer model-brættet, der gives som parameter. Dvs. afgiften, ejeren, pantsætningsværdien, antal huse og hoteller opdateres.

- **removeLoser()**
  Parametre: int playerId, int oldFieldId
  Fjerne spilleren fra det felt, spilleren stod på, men lader spillerens navn stå midt på brættet med beskeden "har tabt".

- **endGame()**
  Parametre: String winnerName
  Fortæller hvem der har vundet og afslutter spillet.

5.3.4 Playercontroller

Playercontrollerens vigtigste attributter er players, som er et array af alle Players, og activePlayer, som er den spiller, hvis tur det er. Klassen har primært ansvar for at holde styr på, hvem den aktive spiller er og for alle overførsler af penge til og fra spillere.

- **updateActivePlayer()**
  Metoden giver turen videre til den næste spiller og opdaterer hvem den aktive spiller er. Der tages højde for, at spillere som er udgået af spillet skal springes over.

- **safeTransferToBank()**
  Parametre: int playerId, int amount Håndterer penge til banken. Metoden returnerer en boolean på baggrund af om spilleren har råd til en bankoverførsel. Hvis han ikke har råd gives resterende penge til banken og der udregnes hvor meget han skylder. Banken er kreditor.

- **safeTransferToPlayer()**
  Parametre: int fromPlayerId, int amount, int toPlayerId Håndterer penge mellem spillere. Metoden returnerer en boolean på baggrund af om spilleren har råd til en bankoverførsel. Hvis han ikke har råd gives resterende penge til den anden spiller og der udregnes hvor meget han skylder. Den anden spiller er kreditor.

- **calculateTotalValue()**
  Parametre: int playerId Beregner summen af spillerens samlede værdier. Her tages højde for spillerens grunde og huse samt om grundene er pantsatte. Løsladelseskort tælles ikke som en værdi.

- **tryToPayDebt()**
  Parametre: int playerId Forsøger at betale spillerens gæld til kreditoren. Hvis det lykkedes at betale gælden af, retureneres true og hvis spilleren stadig har gæld, returneres false.

5.3.5 Boardcontroller

Boardcontrolleren er ansvarlig for opsætning og styring af brættet. Dens vigtigste attribut er selvfølgelig Board board. Boardcontrolleren indeholder de nedenfor beskrevne metoder og desuden et antal hjælpemetoder, som bruges til at lette beregnigerne.
Metoder:

- **setup()**
  Her initialiseres alle felter på brættet, de er på forhånd blevet opdelt i kategorier som har forskellige påvirkning på spilleren: street, chance, tax, visit, brewery, parking og start. De initialiseres i korrekt rækkefølge ifht hvor feltet er på matadorbrættet. De tager et keyword som returnerer data fra textfiler.

- **updateAllRents()**
  Opdaterer afgiften (rent) af alle Ownable felter på brættet i modellaget. Her tages højde for hvordan afgiften udregnes, dvs. antal huse på streets, pantsætning og antal ejede felter i serien.

- **getBuildableStreetIds()**
  parametre: int playerId, int playerBalance
  Returnerer et array af integers med fieldId'er på på de streets, som spilleren kan bygge et hus på. Her tages bl.a. højde for om spilleren har råd til udbygningen, om spilleren ejer

alle grunde i en serie og om udbygningen vil være tilladt ift. at der skal bygges jævnt på grundene i en serie.

```
1  public int[*/* getBuildableStreetIds(int playerId, int playerBalance){
2      boolean[*/* buildableArray = new
       boolean[board.getFields().length*/*;
3      int counter =0;
4      //Loops through all fields, checking if they are buildable
5      for (int i=0;i<board.getFields().length;i++){
6          buildableArray[i*/* = isBuildable(i,playerBalance, playerId);
7          if (buildableArray[i*/*)
8              counter++;
9      }
10     //Makes array of size counter for the ids
11     int[*/* buildAbleStreetIds=new int[counter*/*;
12             counter =0;
13     //Loops trough all the fields, if it's buildable, adds it
14     for (int i = 0; i < board.getFields().length; i++) {
15         if (buildableArray[i*/*)
16             buildAbleStreetIds[counter++*/* = i;
17     }
18     return buildAbleStreetIds;
19 }
```

Listing 2: getBuildableStreetIds()

Ovenstående kode er fra getBuildableStreetIds(). Til at starte med laves der et boolean array, buildableArray, med samme størrelse som fields arrayet, der indeholder alle brættets felter. Herefter loopes igennem fields og for hvert felt kaldes hjælpemetoden isBuildable() som tjekker om det er muligt at bygge på den pågældene grund. Hvis muligt gemmes det i buildableArray. Det er nødvendigt at lave et loop først, for at kunne lave et array, med præcis samme størrelse, som antallet af felter der kan bygges på. Når alle felter på brættet er loopet igennem, oprettes der et nyt int array, buildAbleStreetIds med den korrekte størrelse. I et nyt loop igennem brættet, indsættes de korrekte Id'er nu i buildAbleStreetIds ved hjælp af buildableArray.

- **getSellableStreetIds()**
  parametre: int playerId
  Returnerer et array af integers med fieldId'er på på de streets, som spilleren kan sælge et hus på. Her tages bl.a. højde for om udbygningen vil være tilladt ift. at der skal bygges jævnt på grundene i en serie.

- **getPawnableOrUnpawnableStreetIds()**
  Parametre: int playerId, boolean pawnable, int playerBalance
  Metoden kan returnere to ting: 1) Et array af integers med fieldId'er på de Ownables som spilleren kan pantsætte. 2) Et array af integers med fieldId'er på de Ownables som spilleren kan ophæve pantsætningen på.

  Hvis metoden kaldes med "pawnable == true", beregnes det første. Med "pawnable == false"beregnes det andet. playerBalance bruges til at tjekke, om spilleren har råd til at ophæve en pantsætning. Der tages bl.a. højde for, at en grund ikke kan pantsættes, hvis der er bygget huse på.

5.3.6 ChanceCardController

ChanceCardControlleren opretter et CardDeck-objekt og kalder metoden shuffle() på det. Den har ikke fået ansvaret for alle chancekort, da nogle af dem krævede forbindelse til Board, og dette blev implementeret inden Board blev tilføjet som parameter til metoden playCard(). Med de parametre, som playCard() har nu, kunne metoden godt have håndteret alle de implementerede chancekort.

- **playCard()**
  Parametre: int cardId, PlayerController playerController, Board board
  Metoden håndterer udførelsen af alle de simple chancekort, som tildeler en spiller penge. Metoden tager som argument brættet, board, som skal bruges til at beregne spillerens samlede værdier, ifm. matadorlegatet.

5.3.7 DiceController

DiceControllerens primære ansvar er at holde styr på DiceCup-objektet. Den holder styr på, om begge terninger viser det samme.

- **roll()**
  Ruller terningerne i DiceCup og gemmer værdierne samt om de er ens.

6 Versionsstyring

6.1 Git

I dette projekt har vi brugt git som versionsstyrringsværktøj til at udvikle vores system. Vi har benytet git, via en hjemmeside der hedder GitHub.
Link: CDIO_final
https://github.com/CKyed/CDIO_final

6.2 Branchingstrategi

I dette projekt har vi benytte branchingstrukturen "Branching by feature". Det betyder at vi har lavet en ny branch hver gang vi er gået i gang med at udvikle en ny feature.

Eksempel: Vi har lavet branch fra Master some hedder mainDevelopment, og fra denne branch har vi lavet forskellige feature-branches. Efter vi har lavet en feature der fungerer, bliver main-Development merget ind i den feature branch, hvorefter den fuldt opdaterede feature branch bliver merget ind i mainDevelopment. MainDevelopment er altså den branch, vi hele tiden skal opdatere og holde opdateret, for at undgå omfattende merge-konflikter.
Først når dele af projektet er blevet færdigudviklet i MainDevelopment, og vi har testet at det virker, merger vi dette ind i Master. Master er nemlig vores "officielle"branch, altså den man automatisk finder, når projektet hentes ned.

7 Konfigurationsstyring

Software

- 64-bit Windows (10)
- MacOs 10.11 eller nyere
- Linux GNOME eller KDE desktop
- Java Runtime Enviroment (JRE)
- Java 8 Devellopment kit (JDK)
- IntelliJ IDEA v.2019.2.2

Hardware

- 2 GB ram minimum, 4 GB anbefalet.
- 2.5 GB Harddisk plads, SSD anbefalet men ikke påkrævet.
- Minimum 1024x768 skærmopløsning, dog er det anbefalet ikke at bruge for høj skærmopløsning, da programmets pixels er låst.
- Installation af nødvendig software
  - Java 8 Windows/Mac
  - IntelliJ (Installations vejledning til Windows/Mac) Eller download programmet direkte.

7.1 Maven

Vi har brugt Maven til at administrere vores Java projekt. Maven tager sig af vores dependencies og henter dem for os i den rigtige version:

- JUnit version 4.12
- Matador GUI version 3.1.7
- Java source 1.8

Ud over at administrere vores dependencies så bruger vi også maven til at kompilere vores projekt til en jar fil som indeholder hele vores projekt inklusiv alle vores resource filer, som billeder og tekst filerne til oversættelse af programmet.

8 Test

Til at teste vores program har vi benyttet os af automatiserede unit tests ved hjælp af JUnit og derudover også lavet et par brugertest som supplering. Havde vi haft længere tid, og havde det været fokus af vores projekt, måte vi have testet en lagt større del af programmet. I vores unittests, er code-coverage altså meget lav.

8.1 Automatiseret test

Vi har benyttet os af JUnit version 4.12 til at teste vores system på udvalgte steder. Vi har testet betalings metoderne i playerController-klassen. Disse tests sikrer, at metoderne der bruges til overførsel af penge fra spillere til andre spillere og til banken, virker som de skal. Blandt andet tages højde for, om gælden beregnes rigtigt, når spillere går fallit, og om det er de korrekte beløb, som overføres.
I testen af CardDeck klassen, har vi testet shuffle() og draw() -metoderne, da vi vil sikre os, at at disse lever op til de statistiske forventer vi har til et spil kort. Dvs. der tages højde for, at sandsynligheden for at trække alle kort er ens, når dækket er blandet, og at et kort kommer nederst i bunken, når det er brugt.

8.2 Brugertest

Vi har valgt at brugerteste to scenarier i vores Prison-feature. Dette er fordi de er afhængige af brugerinput.

Testcase	Brugertest af fængsel 1
Scenarie	Spilleren er havnet i fængsel og vil betale 1000 kr for at komme ud
Krav	K12: "Betal 1000 kr inden terningkast for at komme ud"
Preconditions	Spiller er i fængsel
Postconditions	Spiller er ude af fængsel og har betalt 1000 kr
Testprocedure	Spillet startes, og initlialiseres som et normalt matadorspil. Spillet spilles som normalt indtil en spiller ryger i fængsel. Der holdes der øje med spillerens balance. Spilleren skal vælge at komme ud af fængslet ved at betale bøden. Det tjekkes om spillerens nye balance er 1000 kr mindre end før.
Testresultat	Forventet scenarie stemmer overens med spilscenarie
Testet af	Ida
Dato	16/01/2020
Testmiljø	Java SDK 1.8 221 IntelliJ IDEA 2019.2.2 (Ultimate Edition) Build #IU-192.6603.28, built on September 6, 2019 Windows 10 Pro version 1909

Testcase	Brugertest af fængsel 2
Scenarie	Spilleren er havnet i fængsel og vil bruge løsladelseskort for at komme ud
Krav	K21
Preconditions	Spiller er i fængsel har et løsladelseskort
Postconditions	Spiller er ude af fængsel og har ikke længere et løsladelseskort
Testprocedure	Spillet startes, og initlialiseres som et normalt matador-spil. Spillet spilles som normalt indtil en spiller modtager løsladelseskortet og kommer i fængsel. Der holdes der øje med spillerens balance. Spilleren skal vælge at komme ud af fængslet ved at benytte kortet. Det tjekkes om spillerens nye balance den samme som før, og om spilleren næste gang han ryger i fængsel stadig har kortet.
Testresultat	Forventet scenarie stemmer overens med spilscenarie
Testet af	Ida
Dato	16/01/2020
Testmiljø	Java SDK 1.8 221 IntelliJ IDEA 2019.2.2 (Ultimate Edition) Build #IU-192.6603.28, built on September 6, 2019 Windows 10 Pro version 1909

9 Vejledning

9.1 Simpel

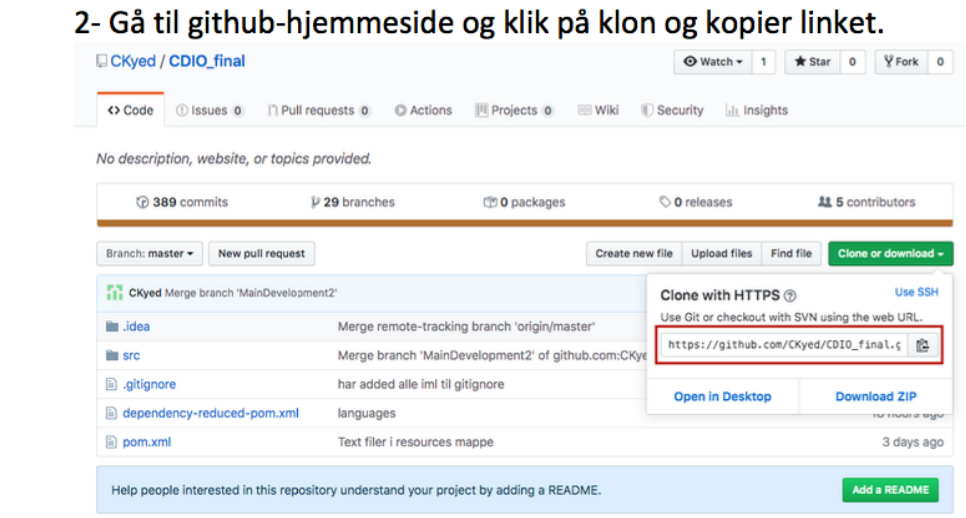
1. Download programmet
2. Find Jar-filens placering
3. Dobbelt-klik på Jar-filen

9.2 Avaranceret

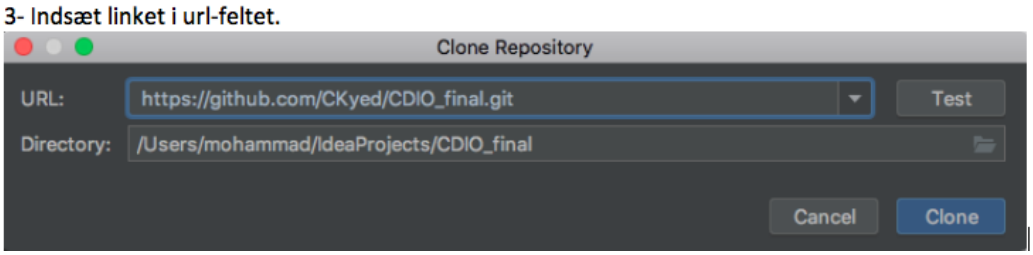
9.2.1 Import fra GitHub

En kort beskrivelse af fremgangsmåden, når et projekt skal importeres fra GitHub. Man kommer til projektets GitHub side vha. dette link: CDIO_finalGruppe 20.

Figur 16: Trin 1



Figur 17: Trin 2



Figur 18: Trin 3

4- Tjek at IntelliJ-projektets SDK stemmer overens med kravene under konfigurationsstyring.

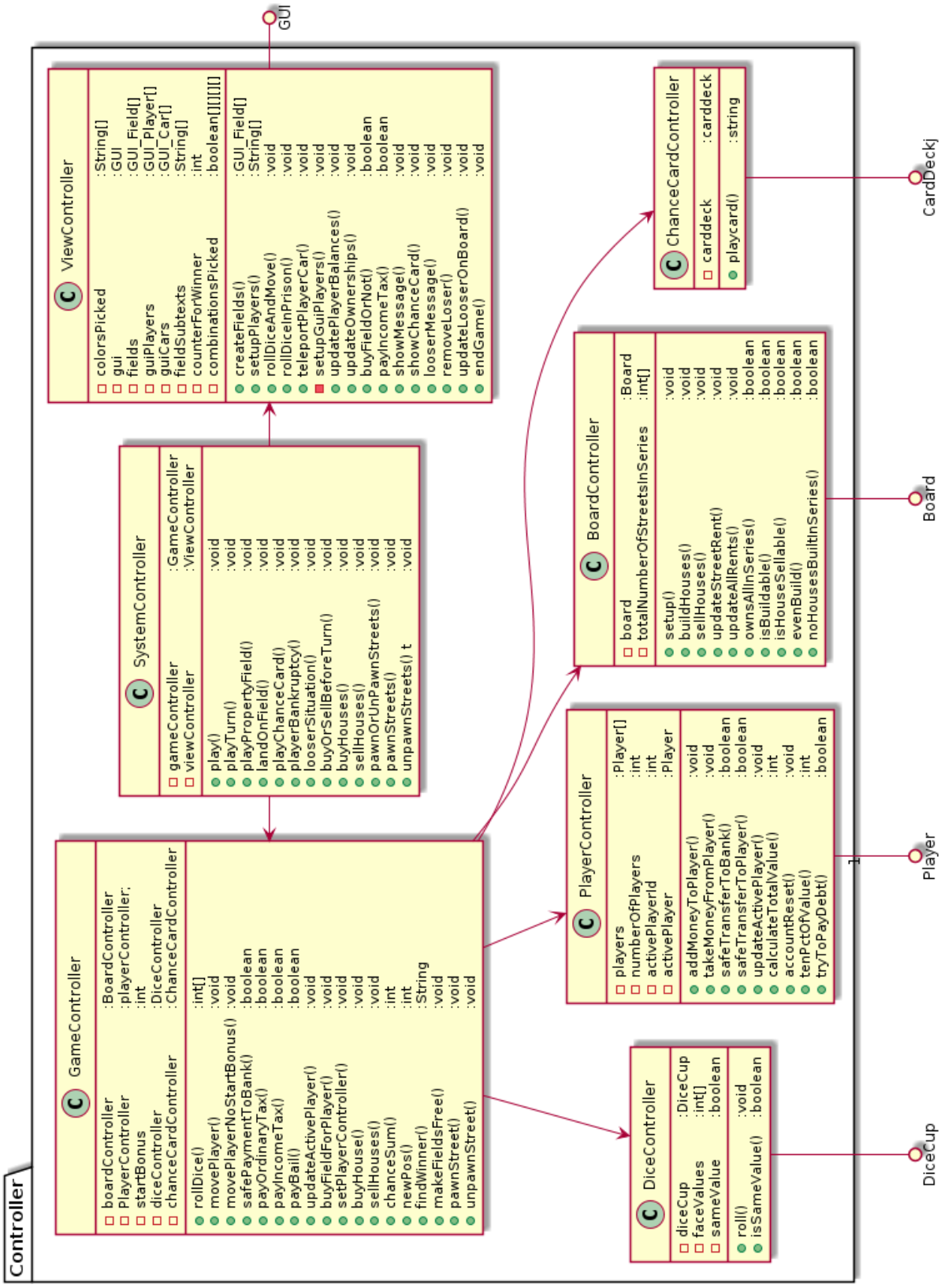
10 Konklusion

Alt i alt må det siges at have været et vellykket projekt. I vores kravanalyse tog vi aktivt stilling til hvilke funktionaliteter, vi ville implemtetere i spillet. De vigtigste fravalg, vi gjorde os fra et fuldt implementeret matadorspil var, auktioner og intern handel mellem spillere. Derudover blev de fleste funktioner taget med. Især var det at købe og sælge grunde og huse en høj prioritet for os.

I analysefasen undersøgte vi hvordan et rigtigt matadorspil fungerer, og hvordan vi kunne overføre dette til vores software-design. Modellaget i vores softwaredesign tager derfor stærkt udgangspunkt i vores domain-model, som repræsenterer et fysisk matadorspil. Vores design bærer også stærkt præg af vores forsøg på at implementere model-view-controller mønstret samt et antal GRASP-patterns. Vi har forsøgt at implementere et controllerpattern og at opnå høj samhørighed og lav kopling i designet. Vi har generelt forsøgt at overholde disse mønstre, men har visse steder været nødt til at prioritere i mønstrene og desuden at nedprioritere at holde fast i strukturen på grund af tidspres. Overordnet må vi dog sige, at vi med succes har oveholdt mønstrene i den udstrækning vi ønskede.
Implementeringen af koden har udgjort størstedelen af tiden, vi har brugt på projektet. I afsnittet om implementering har vi forsøgt at beskrive, hvordan de vigtigste klasser og deres vigtigste metoder fungerer. Dette afsnit afspejler på de fleste måder de beslutninger, vi har taget om vores design.

Alt i alt er vi tilfredse med resultatet. Generelt kan man sige at vi har strukket os langt i et forsøg på at opfylde MVC designprincipper, som går ud på at holde modellaget adskilt fra viewlaget, kun bundet sammen af et controllerlag. På den ene side kunne man have gjort mange ting med færre linjer kode, men på den anden side har det været en god øvelse for os.
Vores tidsforbrug har fungeret godt, løbende har vi forudset halve og hele dage, og hvor mange opgaver vi ville nå på den tid. Det har stemt nogenlunde overens. Et enkelt kritikpunkt kunne være at vi ikke har haft en særlig iterativ proces, men derimod stort set udviklet ud fra en vandfaldsmodel.

11 Bilag



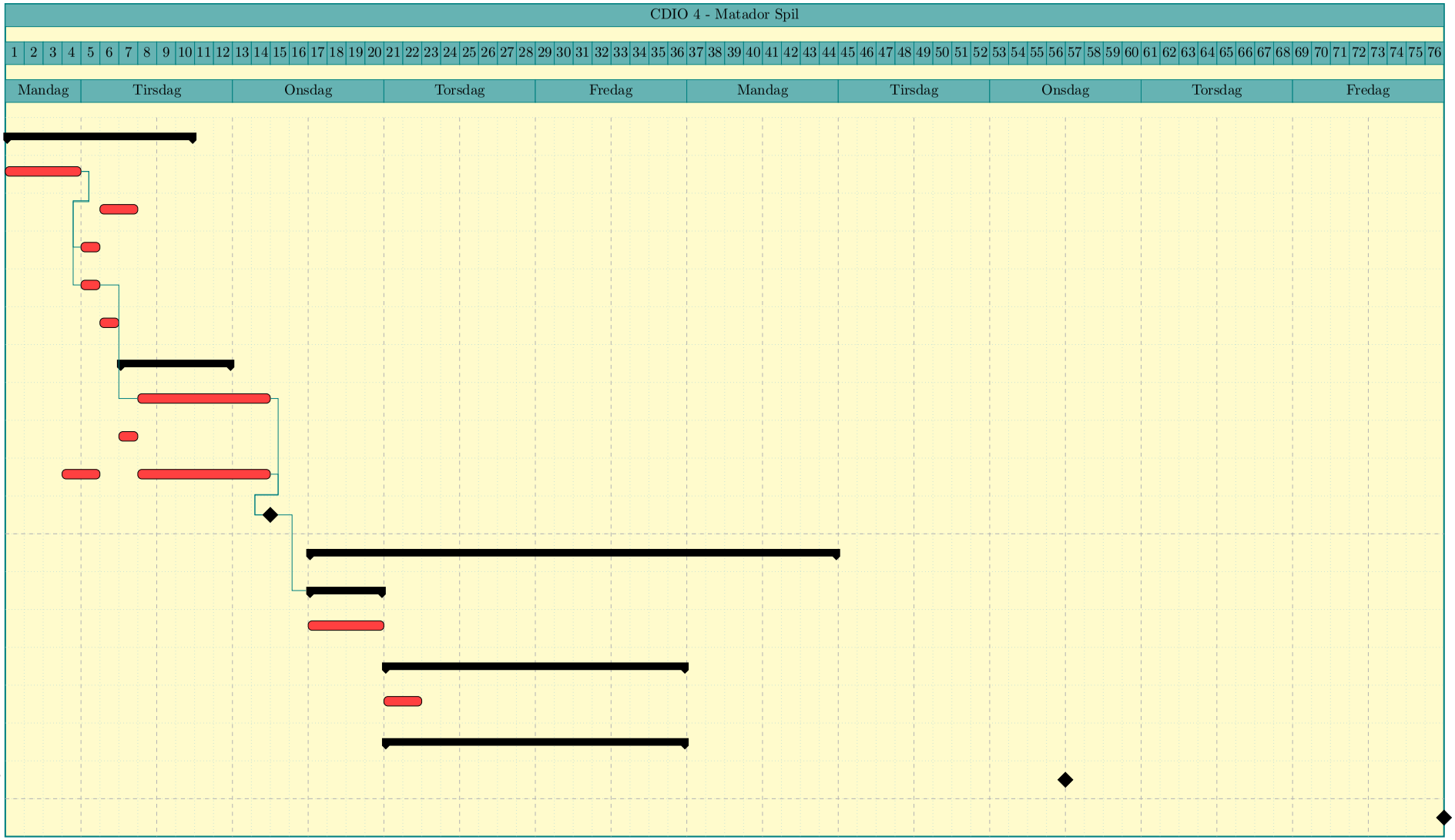
Figur 19: Det fuldsætndige klassediagram over Controller-pakken. Controllerdiagrammet viser pakken med controller klasserne som styrer model klasserne. Mange af vore controllere fungerer også som creatorer for modellerne.

11.1 Gantt diagram

I dette diagram aflæses hvordan vi gennem forløbet disponerer over tiden og ressourcerne i forbindelse med projektet. De turkis farvede pile angiver afhængigheder, eksempelvis var udformningen af domænemodellen nødvendig for klassediagrammet. De sorte barrer viser en gruppe af opgaver og de røde barrer viser de enkelte opgaver.

Vi arbejdede med gantt diagrammet ind til M1 og derefter begyndte fik vi ikke brugt det så meget, men vi har lært at det er en god måde at strukturere sit projekt på og vi vil klart bruge det i fremtiden.

I stedet for gantt diagrammet har vi haft scrumm møder og haft en tavle hvor vi har skrevet de nuværende opgaver ned og hvilke opgaver vi manglede de forskellige dage i projektet.



Figur 20: Gantt diagrammet over hele projekter med Milestones, grupper, afhænmgigheder og de enkelte opgaver.