DANMARKS TEKNISKE UNIVERSITET

62531 DEVELOPMENT METHODS FOR IT SYSTEMS

62532 VERSION CONTROL AND TEST METHODS

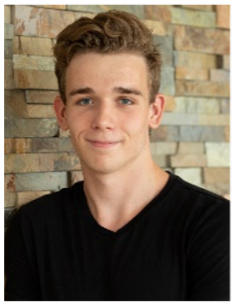
02312 INTRODUCTORY PROGRAMMING

GRUPPE 20

CDIO 3

Forfattere
Mads SØRENSEN
Bertram KJÆR
Lucas SCHOUBYE
Ismail ALI
Mark NIELSEN

Kursusansvarlig
Christian BUDTZ
Daniel Kolditz RUBIN-GRØN
Thorbjørn KONSTANTINOVITZ



Mads Sørensen
S215805

Mark Nielsen
S204434

Lucas Schoubye
S215809

Ismail Ali S190201

Bertram Kjær
S215775

26. november 2021

Indhold

Abstract

In this project our group was tasked with creating a virtual version of the Monopoly Junior boardgame. The group had 3.5 weeks to produce the most complete product with proper documentation. The project was conducted with focus on the agile method Rational Unified Process and programmed in Java. In the analysis the requirements were weighted, the structure inspected and risks were dissected. The end product fulfilled the vast majority of the requirements, with focus on the most impactful requirements. We learned the importance of documentation and investing resources in the design phase to minimize issues, while still keeping the development agile and feedback driven.

1 Indledning

I denne rapport vil vi dokumentere vores arbejde, herunder analyse, design og implementering af projektet. Opgaven består lave et junior matador spil med de danske spilleregler. Spillet går på tur hvor der skiftevis slås med terninger for at bevæge sin spillebrik. Når en spiller lander på et felt købes det for et specifikt beløb, hvis ingen anden spiller ejer det. Når en spiller lander på et felt ejet af en anden spiller skal de betale et beløb til spilleren. Spillet bliver ved indtil den første spiller er løbet tør for penge. I rapporten dokumenteres b.la. de forskellige artefakter, vi har udarbejdet, under de forskellige faser; analyse, design, implementering og test. Der er anvendt agile udviklingsmetoder, herunder at udvikle i små iterationer. Programmet er udviklet i Java med værktøjet IntelliJ, og skal bruges i DTU's databare.

2 Projektplanlægning

Tidsplan

Vores tidsplan blev styret iterativt. Vi fik opstillet Milestones i Kanban hvorefter der blev holdt korte gruppemøder ved hver grupperegning. I vores mødereferater blev der skrevet hvilke features og artefakter som skulle laves den givne uge. Disse gruppeopgaver blev opdateres i Kanban. Dette gav et dynamisk overblik over hvilke opgaver der skulle løses og minimerede hvor meget tid der skulle bruges på at opdatere selve tidsplan artefaktet.

Samarbejdsaftale

I dette projekt blev der arbejdet med samme gruppekontrakt som tidligere iterationer af CDIO projektet. Der blev taget højde for ambitionsniveau, gruppepligter og ansvarsområder. Der blev implementeret et klippekort system, hvorpå der ville være en økonomisk straf på 50kr hvis overtrædelserne akkumulere. Disse kunne være at møde for sent op, mangel på hjemmearbejde og mangel på kommunikation. Lucas var ansvarlig for at overholde dette klippekort system.

Gruppekonflikt

I begyndelsen af opgaven opstod en gruppekonflikt. Ismail vidste visse udfordringer med niveauet af programmeringen og havde derfor besvær ved at løse koderelaterede opgaver. Gruppen havde et møde om denne gruppekonflikt og indgik en aftale om at overlade disse arbejdsopgaver til resten af gruppen. Det vil derfor sige han overlod alle arbejdstimer til rapportskrivning og dokumentering, men ikke har deltaget i kodning.

Mødereferater

I dette projekt blev der overholdt mødereferater over gruppens Discord-server. Herunder blev der noteret dato, Dagens arbejde, arbejdsopgaver til næste gang, næstemøde dato og tidspunkt. Ændringer i mødetidspunkter og andre detaljer blev notificeret til gruppen messengergruppe specifikt til gruppearbejdet. Mads var ansvarlig for disse.

Værktøjer

I dette projekt blev der gjort brug af flere forskellige værktøjer. Der blev programmeret i Intellij, til versionstyring blev Git og Github brugt. Maven blev brugt samt det givne GUI. Rapportskrivning foregik i et LaTex dokument under vores projektmappe, derved også versionstyret. JUnit blev brugt til at foretage unittest af vores kode. Til vores tidplan blev der brugt Kanban hvor alle gruppemedlemmer kunne gå ind og updatere diverse issues.

Timeregnskab

Navn	Gruppetimer	Solotimer
Bertram Kjær	16 timer	28 timer
Ismail Ali	24 timer	15 timer
Lucas Schoubye	24 timer	23,5 time
Mads Sørensen	23 timer	23 timer
Mark Nielsen	24 timer	24 timer

3 Analyse

Analysen af opgaven tager udgangspunkt i følgende artefakter; domænemodel, kravspecificering, use case diagram, use case beskrivelser, systemsekvensdiagram, sekvensdiagram.

3.1 Kravspecifikation

Krav er opstillet ud fra den stillet opgave. Vi identificerede først kravene ud fra spilreglerne, som kan kategoriseres som funktionelle krav. Vi havde også ikke-funktionelle krav til b.la. brugervenlighed og konfiguration. Vi havde i projektet stort fokus på at fuldføre de krav, der bragt os tættest på det ønsket produkt, vi har derfor prioriteret vores krav i 3 niveauer, høj, middel, lav. Prioriteterne følger principperne "Must have", "Should have", "Could have". Vurderingen for de enkelte krav er bundet om på kriteriet om at spillet bliver mere eller mindre spilbart for brugeren - altså det skal minde mest muligt som et almindeligt Junior Matador spil.

Undervejs i projektet revurderede vi flere gange vores krav, så de undervejs har været opdateret ift. afklaring af ønsker fra kundens side. B.la. blev der skiftet om på spillebrættet, hvoraf der som konsekvens skulle omprioriteres.

Tabel 1: Funktionelle krav

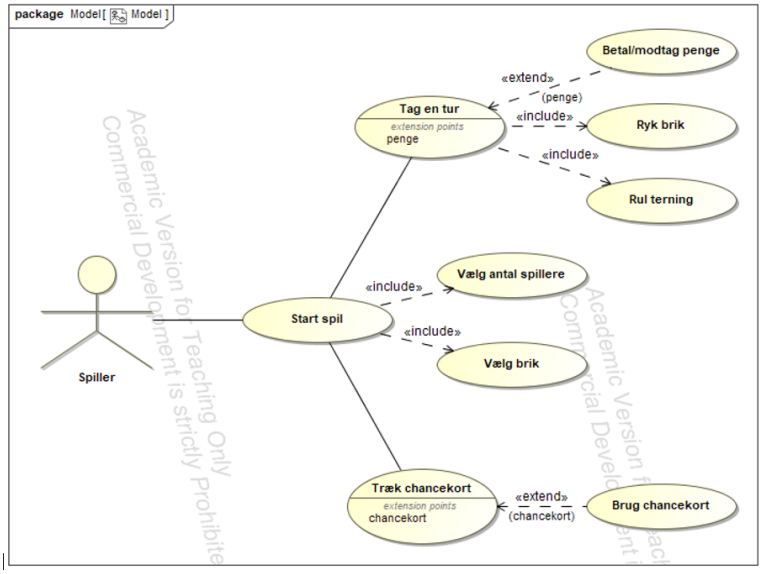
ID	**Krav**	**Prioritering**
R01	Spillet skal understøtte 2 til 4 spillere	Høj
R02	Hver spiller skal starte med 35	
R03	Hver spiller skal starte på startfeltet	
R04	Spilleren skal købe et felt, hvis de lander på det, hvis ikke ejet af anden spiller.	
R05	Spilleren skal betale ejeren dobbelt, hvis ejeren af feltet ejer begge felter af samme farve som det givne felt.	
R06	Spillet skal bruge GUI'en importeret fra Maven.	
06A	Brættet skal bestå af 24 felter.	
06B	Hver 3. felt der kan købes skal have en farve.	
06C	Hver farve skal have 2 felter som kan købes.	
R07	Spillet skal gå på tur, og spilleren slår med 2 terninger	
R08	Spillere skal rykke antallet af felter frem lig værdien af terninger.	
R09	Spillet skal have min. 3 forskellige chancekort	
R10	Spillere, der lander på et købt felt, skal betale feltets leje til ejeren af feltet.	Middel
R11	Spillere skal modtage $2 ved at passere start	
R12	Spillere betaler $3 ved at lande på "gå i fængsel"	
12A	Pengene betales til gratis parkering feltet	
12B	Spillere skal ændre deres position til "fængsel" når de lander på "Gå i fængsel"	
R13	Spillet bør have alle chancekort udleveret af kunden	Lav
R14	Spillet starter med at hver spiller skal rulle en terning, hvorpå den spiller med det højeste kast starter.	
14A	Hvis flere spillere slår den samme højeste værdi slår de om igen.	
R15	Spilleren skal få udbetalt alle opsparede penge når der landes på feltet Fri Parkering	
R16	Kortene skal blandes ved spillestart	
R17	Rækkefølgen på kortene er det samme gennem hele spillet	
R18	Spillere kan vælge brikfarve	
R19	En spiller vinder hvis pågældende har flest penge, når en anden spiller går fallit (ikke råd til at betale husleje, købe en ejendom eller betale en afgift fra et chancekort).	

Tabel 2: Ikke funktionelle krav

ID	Krav
Usability	
R20	En bruger skal starte spillet uden hjælp
Supportability	
R21	Mindst 1 JUnit test og 1 brugertest
R22	Mindst 1 JUnit test og 1 brugertest
R23	Lav mindst 3 testcases, testprocedure og testrapport
R24	Inkluder code coverage dokumentation
R25	Rapporten skal indeholde en vejledning i hvordan man importerer Git-repository i IntelliJ
R26	Koden skal versionsstyres lokalt med Git-repository
R27	Programmet skal kunne køres på DTU databar

3.2 Use case diagram

Der er lavet et use case diagram ud vores use cases, hvor at der har været fokus på at vise hvordan at brugeren interagere med spillet. Specielt kan der ses ved "tag en tur"use casen at brugeren kan interagere med spillet på flere måder. Der er valgt at inkludere use cases som har et lavt scope selvom det ville have været fint kun at have start spil use casen, men der er valgt at inkludere sub use casen for at illustrere hvordan at brugeren kan interagere med spillet.



Figur 6: Use case diagram

3.3 Use cases

Vi har identificeret og analyseret en række use cases i den udleverede opgave. De vigtigste use cases er: *Start spil, Tag tur* og *Træk chancekort*.

- **UC1:** Start spil
  - **UC1.1:** Vælg antal spillere
  - **UC1.2:** Tag en tur
- **UC2:** Tag en tur
  - **UC2.1:** Betal/Modtag penge
  - **UC2.2:** Ryk brik
  - **UC2.3:** Rul terning
- **UC3:** Træk chancekort
  - **UC3.1:** Brug chancekort

Brief

ID: UC1 Start spil
Actor: Spiller
Basic flow: Spiller starter spillet.

ID: UC1.1 Vælg antal spillere
Actor: Spiller
Basic flow: Spiller vælger hvor mange spillere der deltager i spillet.

ID: UC1.2 Vælg brik
Actor: Spiller
Basic flow: Hver spiller vælger en brik som de får tildelt resten af spillet.

ID: UC2 Tag en tur
Actor: Spiller
Basic flow: Spiller påbegynder tag en tur.

ID: UC2.1 Betal/modtag penge
Actor: Spiller
Basic flow Spiller betaler en modspiller (eller bank) penge.

ID: UC2.2 Ryk brik
Actor: Spiller
Basic flow Spiller rykker sin brik antal felter frem på brættet med det antal øjne, terningen viser.

ID: UC2.3 Rul terning
Actor: Spiller

Basic flow Når det er den pågældende spillers tur, så ruller spilleren terningen, hvorefter en terningsværdi fås. Værdien benyttes til at ændre spillerens position.

ID: UC3 Træk chancekort
Actor: Spiller
Basic flow Spiller trækker et chancekort, hvorefter et tilfældigt kort modtages.

ID: UC3.1 Brug chancekort
Actor: Spiller
Basic flow: Spiller benytter sit chancekort.

3.3.1 Fully dressed

ID: UC2 Tag en Tur
Actor: Spiller
Stakeholders. Spillerne: Vil gerne få en høj balance og vinde spillet. Vil gerne underholdes.
Pre-conditions: 1. Spillet skal være startet 2. Det skal være den nuværende spillers tur
Post-conditions: 1. Spillerens tur er slut eller spillet er vundet
Basic flow 1. Spiller modtager besked fra GUI'en: "Spiller X tur, klik på rul terning" 2. Spiller klikker på knap som ruller med terninger med resultat "X" 3. Spillerens brik rykkes "X" antal felter frem 4. Spillet tjekker hvilket felt de er landet på 5. Feltet er ikke købt 5.1.1: Derfor betaler spilleren for feltets pris 5.1.2: Feltet opdateres til at være ejet af spilleren 6. Spilleren passerede ikke start 7. Spillet tjekker om spilleren har en negativ account balance 7.1.1: Spilleren har en positiv balance 8. Spillet ændre hvilken spiller er aktiv
Alternative Flows 5.2.1: Feltet er købt 5.2.2: Spilleren betaler feltets lejepris til ejeren af feltet 5.3.1: Feltet er Chancen 5.3.1.1: Spilleren trækker et chancekort 5.3.1.2: Spilleren udfører chancekortets effekt 5.4.1: Feltet er "Gratis Parkering" 5.4.1.1: Spilleren modtager $ lig med værdien af "Gratis Parkering" 5.4.1.2: "Gratis Parkering" bliver sat lig med 0

5.5.1: Feltet er “Gå i fængsel”

5.5.1.1: Spilleren betaler $3

5.5.1.2: Spillerens position bliver sat til “Fængsel”

5.6.1: Feltet er “Fængsel”

5.6.1.1: Spilleren betaler ingen penge

6.1.1: Spilleren passerede startfeltet

6.1.1: Spilleren modtager $2

7.2.1: Spilleren har en negativ balance

7.2.1.1: Spillet tjekker hvilken spiller har den højeste balance

7.2.1.2: En enkel spiller har højeste balance

7.2.1.3: Den givne spiller vinder spillet og spillet stopper

7.2.1.1: Flere spillere har den højeste balance

7.2.2.2: De spillere har begge vundet og spillet stopper

4 Begrebsforklaring

4.1 Arv

Arv, i programmering, er når en klasse nedarver kode fra fra en anden klasse. Hvis fx. der skal programmeres et webshop-system hvor der skal bruges flere typer af kontoer. Disse kunne være en administrator og kunde. Disse kontoer skal dele egenskaber som adgangskode og e mail, men administrator skal have yderligere redskaber. Her kunne administrator være nedarvet fra kunde klassen. Hvis kunden skulle have egenskaber som administratoren ikke skulle have, vil man blive nødt til at strukturere arven anderledes, siden al information nedarves. Dette bringer os videre til udtrykket abstract.

4.2 Abstract

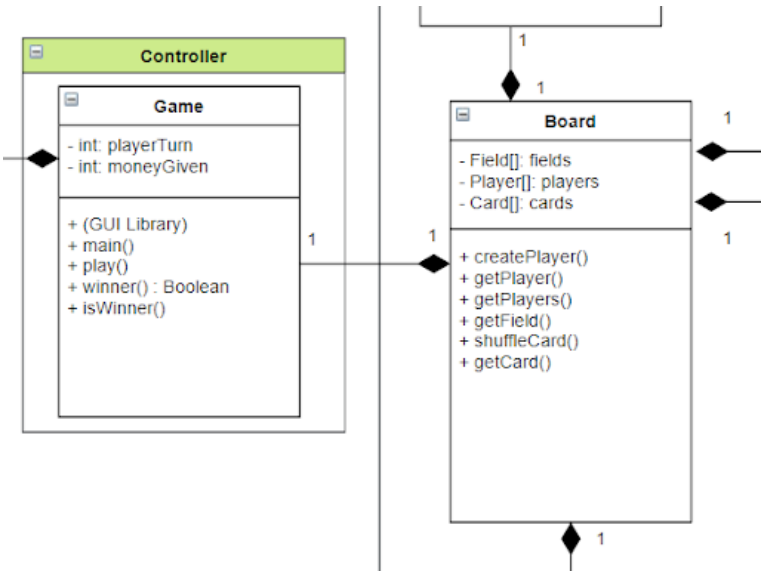
Abstract, i programmering, betyder at klassen ikke kan instantieres. Altså at der ikke kan laves objekter fra klassen. Denne egenskab bliver ofte brugt i sammenhæng med arv, som forklaret ovenover. Hvis der skulle programmeres et spil hvorpå der kunne spilles som forskellige dyr, ville abstract være brugbart. En bjørn og en flagermus ville ikke kunne nedarve fra hinanden da de begge deler visse egenskaber, men også har signifikante forskelligheder. Her ville man ikke opnå den ønskede funktionalitet. Her kunne der indføres en abstrakt "pattedyr" klasse som ikke kan instantieres, hvori der ligger den delte kode af bjørn og flagermus. Her vil man nemt og hurtigt kunne arbejde videre med at implementere flere forskellige pattedyr uden at copy-paste unødvendigt. Det skal også siges at det er en fordel at pattedyr-klassen ikke kan instantieres, da det ikke ville give mening i domænet at et pattedyr uden yderligere specialization ville finde sted, da det er en biologisk, kategoriserende klasse og ikke et dyr i sig selv.

4.3 Polymorphism

Siden at field klasserne bliver nedarvet ville dette kunne beskrives som polymorphism. Orden nedstammer fra græsk som betyder "mange former". I programmering ville det kunne understøtte metoder med forskellige implementationer.

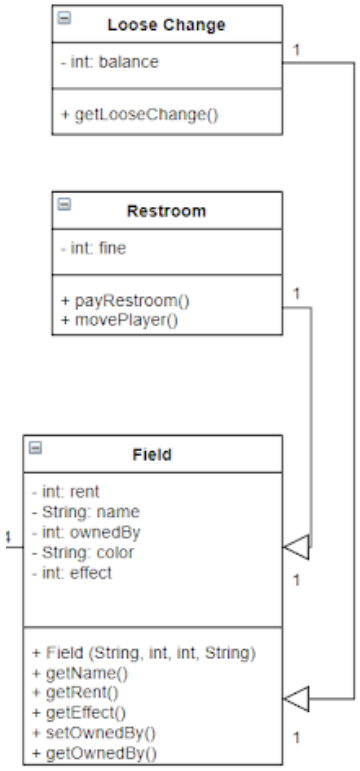
5 Design

5.1 Designklassediagram



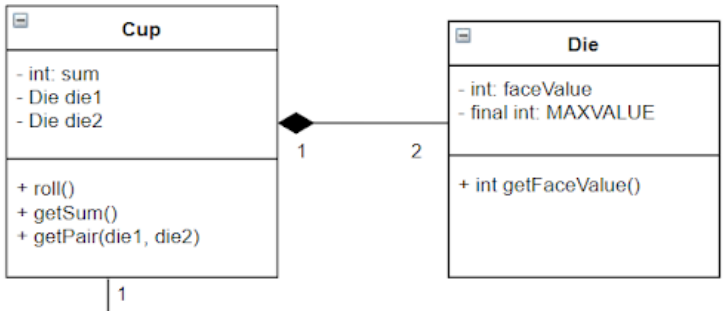
Figur 7: Design klasse diagram udsnit af Game og Board

Som der ses på billedet ovenover er vores main metode er i game klassen sammen med reglerne for logik, hvor board er en controller der udfører logik. Man kunne tage main-metoden ud i en separat klasse, men det har vi valgt ikke at gøre, da vi vil have vores main metode i vores game klasse. Der er valgt at lave nedarvning fra field klassen til Loose Change klassen samt Restroom klassen, da Loose Change klassen og Restroom klassen er deler flere attributter med field klassen dog er de begge lidt forskellige fra field klassen, som ses her:



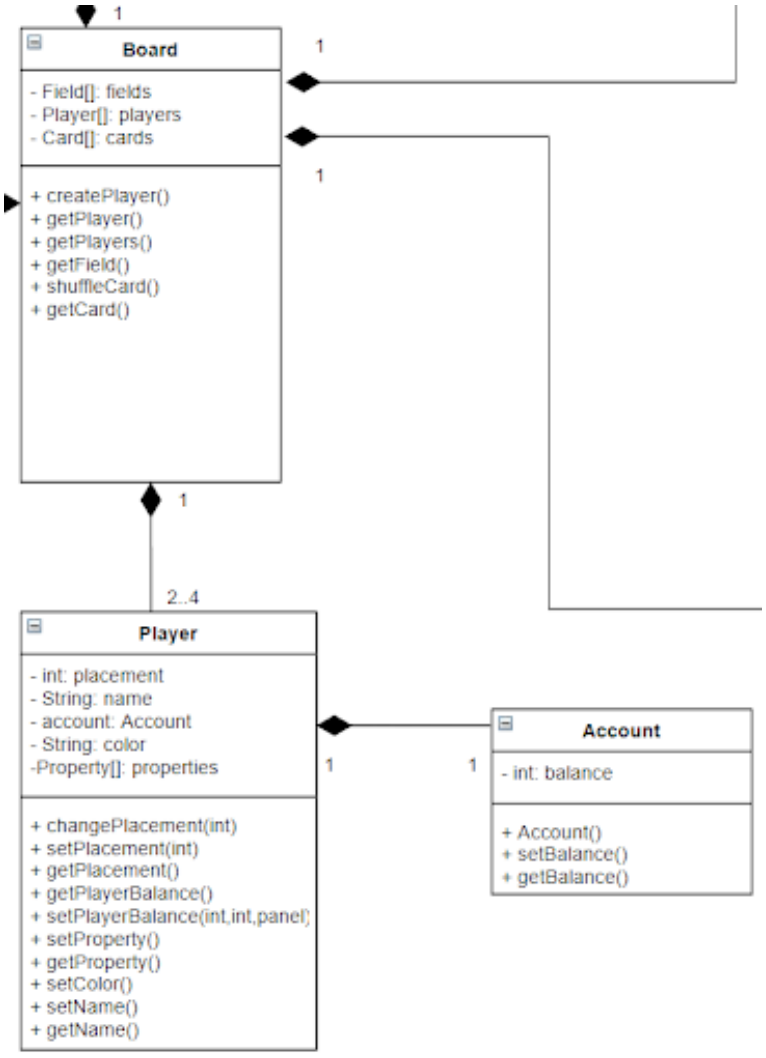
Figur 8: Design klasse diagram udsnit af Field, Restroom og Loose Change

Der er valgt at have en Cup klasse samt en Die klasse hvor Die klassen indeholder en metode der giver et tilfældigt tal mellem 1 og 6 som simulere en rigtig terning. Det er så Cup klassen simulere et raflebæger som slår med terningen ved hjælp af roll metoden hvor der tages to terninger fra Die klassen og skabes 2 værdier. Herefter er der getSum metoden der finder en sum af de to terninger, hvilket er det slag vores spiller har lavet. Dette ses her i design klassediagrammet:



Figur 9: Design klasse diagram udsnit af Cup og Die

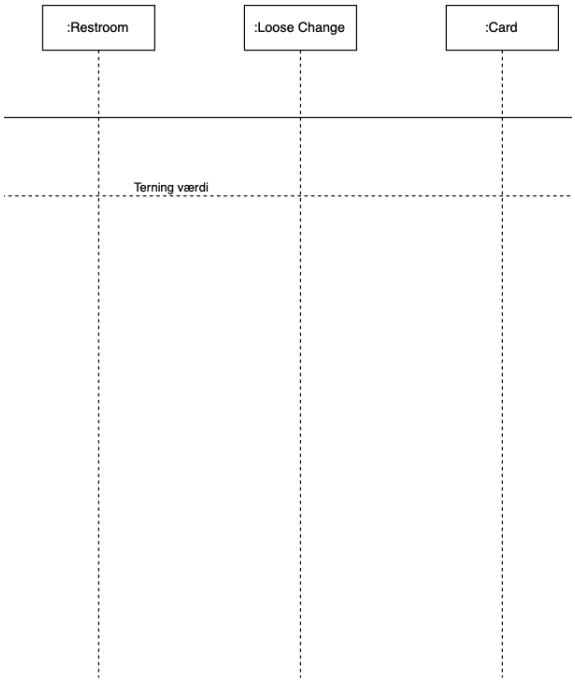
Der er valgt at have en player klasse som simulere vores spillere der bliver tildelt navn, farve, placering og ejede grunde med mere. Til player klassen er der koblet en account klasse som styrer spillerens balance, det er også denne attribut der er valgt til i Board klassen at bestemme om en spiller er gået bankerot og derfor har tabt spillet.



Figur 10: Design klasse diagram udsnit af Board, Player og Account

5.2 Sekvensdiagram

I sekvensdiagrammet vises en besked der beder spilleren at starte spillet. Dernæst trykker spilleren på knappen som ruller terningen. faceValue bliver kaldt af Die for at give værdien af tærningen. Værdien bliver omdannet til et antal af felter der bliver rykket på brættet. Feltet tjekker om den er ejet af en anden spiller. Hvis ikke, bliver spilleren bedt om at købe feltet, der også opdateres til ejet. Sidst bliver der tjekket spillerens balance. Hvis positivt, får spillet ændret spillerens aktivitet.



Figur 11: Udsnit af Sekvensdiagram

I figur 6 vises de klasser (Restroom/Loose change/Card) som bruges til alternative flow i sekvensdiagramet.

5.3 GRASP

5.3.1 Creator

Inden for programmering er GRASP-konceptet Creator den evaluering af hvilke klasser skal instantiere hvilke objekter, og derved kontrollere dem. I dette projekt kan denne structur ses flere steder i vores design. Vores Cup-klasse er controller af vores Die-object da der nemt kan tilføjes ekstra terninger i fremtidige projekter. Vores Board-klasse instantiere vores Field-array hvorpå Field objectet instantiere GUI-elementet. Board-klassen instantiere også vores Player-array som herefter instantiere GUI-elementet player. Denne struktør gør koden mere læselig og organizeret som derved minimere fejl.

5.3.2 Information Expert

Inden for programmering er GRASP-Konceptet Information Expert hvilken klasse som indeholder eller har størst adgang til brugbar data. Vores information expert er vores board-klasse. Da denne klasse har højeste coupling til de andre klasser ville vi få en mere overskuelig kode med mere professionelle metodekald som vores information expert.

5.3.3 Low Coupling High Cohesion

Konceptet om low Coupling omhandler at klasser har så få forbindelser til andre klasser som muligt. Dette gør koden mere organizeret da udviklerne ikke skal tage højde for hele systemet hvis en enkelt klasse skal udvikles. Dette kan ses på klassediagrammet hvor vi prøver at undgå unødvendige couplinger og redundant information. High Cohesion omhandler hvor præcis et fokusområde hver klasser er designet til at omfange. Dette kunne for eksempel ses flere steder i productet som Field-klassen der indeholder felt information eller Die-klassen som kun returnere en tilfældig integer fra 1 til 6.

5.3.4 Polymorphism

Inden for Programmering er GRASP-Konceptet om polymorphism at have flere elementer som minder om hinanden, dog med visse specifikke forskelligheder. Dette kunne fx være method overloading eller arv. I dette projekt bruges konceptet om polymorphism i vores field-typer. Der kan landes på en række af forskellige felter som arver kode fra en abstract klasse. Dette gør koden mere læselig og minimere gentagelser.

5.3.5 Pure Fabrication

Inden for Programmering er GRASP-Konceptet Pure Fabrication at indføre elementer som ikke indgår i domænet. men ville yde når domænet digitalizeres. I dette projekt ville vores Cup-klasse være Pure Fabrication samt vores main-klasse og account klasse.

6 Implementering

Vi lavede 2 iterationer af programmet i løbet af projektet. Det var vigtigt for os, at vi halvvejs i projektet havde en *proof of concept*, hvor de vigtigste krav skulle implementeres. Vores første prototype havde følgende funktionalitet.

1. Spillet skal understøtte 2 spillere og gå på tur.
2. Man skal kunne gå rundt på brættet.
3. Mist penge ved at lande på felt
4. Gøre brug af GUI'en og spilleren rykker visuelt rundt svarende til værdien af terningskastet.

Alle felter havde samme effect, og der blev trukket 1$ uanset hvilket felt man landede på. Under udviklingen af første prototype, var det sværest at løse IndexOutOfBounds fejl, når spillere gik en omgang på brættet. Den første fungerende prototype blev deludviklet fra d. 9/11-2021 til d. 15/11-2021, hvor vi havde en velfungerende med kun mindre bugs.

Under udvikling af 2. version af programmet implementerede vi b.la. features som

1. Chancekort.
2. Alle felttyper og spillere kan nu eje ejendomme.
3. Dobbeltejerskab af felter med samme farve.
4. Penge udveksling mellem spillere og spil.
5. Finde en vinder af spillet.
6. Andre simple spilleregler

Her var chancekort klart den største udfordring af implementere, som vi kommer til at uddybe i dette afsnit.

6.1 Version 1

I vores første prototype kan vi tilføje spillere og bevæge os på board med 24 felter. I udviklingsstarten oprettede vi først klasserne til at lave de basale objekter til at holde dataen. De fleste klasser havde vi fra tidligere CDIO-delprojekter herunder Cup, Die, Player og Account. Board klassen blev oprettet som informationsekspert, der holder de fleste objekter og arrays af objekter. I pseudo kode er de basale attributter som følgende.

```
1 public class Board() {
2     private GUI gui;
3     private Player[] player;
4     private ChanceCardDeck chanceCard;
5     private Field[] field = {
6         new Field(),
7         // 23 more fields
```

```
8       };
9   }
```

Derudover har vi Main klassen, der står for alt spillogikken. I første version af programmet følger hver spillertur følgende steps.

1. Rul terning
2. Fjern spiller GUI fra brættet
3. Check om spilleren er gået en omgang på brættet og korriger position, for at undgå Index-Out-Of-Bounds fejl
4. Opdater spillerens placering med værdien af terningkastet, og opdater spillerens position grafisk
5. Spilleren har landet på et felt - træk 1$
6. Ny spillertur

Vi løste problemet med at spillere kan gå rundt om brættet, ved at udføre check om summen af terningekastet og placering er over 24. Er den det korrigerer vi spillerposistionen.

```
10  if(placement + sum >= 24) {
11      player.setPlacement(sum - 24);
12      board.removePlayer(currentPlayer, placement);
13      sum = 0;
14  }
```

Her er pseudokoden over vores Main spillogik.

```
15  while(! winner) {
16      sum = cup.roll()
17      removeplayer()
18
19      //Check omgang
20      if (placement + sum >= 24) {
21          player.setPlacement(sum - 24)
22          sum = 0;
23      }
24
25      player.setPlacement(sum)
26
27      // Update GUI with new placement
28      board.movePlayer(placement);
29
30      player.setPlayerBalance(-field.getRent())
31
```

```
32      currentPlayer++;
33  }
```

6.2 Version 2

Som nævnt fik vi i 2. version af programmet udviklet feautures som f.eks. Chancekort og Felttyper, som vi vil gennemgå her. Disse har vi vurderet er de vigtigste features at gennemgå.

6.2.1 Felter

Vi implementede et klassehiearki af felter nedarvet fra superklassen Field. Subklasserne er Property, Jail, FreeParking, Start og ChanceField, som alle bruger nøgleordet *extends* i klasse deklarationen. Den vigtigste attribut i Field, som bliver nedarvet til subklasserne er objektet GUI_Field, da alle felttyper skal have en GUI-repræsentation. Vi tager udgangspunkt i subklassen Property, da denne er mest relevant. Nedenfor ses pseudo-koden.

```
34  public abstract class Field {
35      protected String name;
36      protected GUI_Field field;
37
38      // Getter og Setters
39  }
40
41  public class Property extends Field {
42      protected int rent;
43      protected Player owner;
44
45      Public Property(GUI_Street field, int rent) {
46          this.field = field;
47          this.rent = rent;
48      }
49
50      // Getters og Setters
51  }
```

I Property konstruktøreren giver vi Property objektet et nyt GUI objekt GUI_Street, der gør at vi kan tilgå GUI_Street's metoder, når spillere lander på dette felt. Field er *abstract* da denne ikke skal instantieres. Vi laver i klassen Board et array af typen Field, som har subklasserne til Field.

```
52  public class board {
53      private Field[] fields = {
54              new Start(new GUI_Start(),"Model.Start", Color.WHITE, "Startfeltet
55              new Property(new GUI_Street(),1, "Burgerbaren", Color.GRAY),
56              new Property(new GUI_Street(),1, "Pizzahuset", Color.GRAY),
57              new ChanceField(new GUI_Chance()),
```

```
58              new Property(new GUI_Street(),2, "Slikbutik", Color.CYAN),
59              // osv. 24 felter ialt
60      }
61  }
```

Når vi tjekker hvilket felt, spilleren lander på, kan vi derved tjekke på objecttypen med *instanceof*. Det viser vi senere.

6.2.2 Chancekort

Chancekort blev implementeret med 2 klasser ChanceCard og ChanceCardDeck. ChanceCardDeck udgør i dette tilfælde hovedsageligt et array af ChanceCards, og dens tilhørende metoder. Vi gennemgår her metoderne *useChanceCard()*, som er den der står for korteffekterne, og *drawCard*.

drawCard() returnerer det øverste ChanceCard i array'et og ligger det nederst i array'et. Vi bruger for-loopet til at flytte hvert Card ét index op i arrayet, antaget at index 0 er defineret som "øverst".

```
62  public ChanceCard drawCard() {
63          ChanceCard card;
64          card = deck[0];
65
66          for(int i = 0; i < deck.length; i++) {
67              if(i > 0) {
68                  deck[i - 1] = deck[i];
69              }
70              deck[i] = card;
71          }
72          return card;
73  }
```

useChanceCard() Når brugeren lander på et chancefelt, skal selve felteffekterne implementeres. Det har vi valgt at gøre med et switch-statement, hvor vi har nummereret hvert chancekort. Derefter udføres den effect vi har programmeret. Vi har kun nået at implementere 4 chancekort, som indebærer at rykke playeren eller modtag/betal penge.

```
74  ChanceCard card = drawCard();
75  switch(card.getNumber()) {
76      case 0:
77          //Card effect
78          break;
79      case 1:
80          //Card effect
81          break;
82      case 2:
83          //Card effect
84          break;
```

```
85      // osv.
86      default:
87          break;
88  }
```

6.2.3 Gamelogik

I 2. version blev vores Main metode, lidt for stor end hvad vi havde håbet. Vi havde desværre ikke tid til at enkapsulere og rydde op i den. Her skriver vi den i Pseuo kode

```
89  while (! winner) {
90      sum = cup.roll()
91      removeplayer()
92
93      //Check omgang
94      if (placement + sum >= 24) {
95          player.setPlacement(sum - 24)
96          sum = 0;
97      }
98
99      player.setPlacement(sum)
100
101     // Update GUI with new placement
102     board.movePlayer(placement);
103
104     Field field = board.getField(placement);
105
106     //Check Field type
107     if (field instanceof ChanceField) {
108         // Typecast to Model.Property
109         Property property = (Property) field;
110
111         // Check for Field owner
112             // Check for double color Field owner
113
114         // Check for ingen Field owner
115     }
116
117     if (field instanceof Chancefield) {
118         // Draw Chance Card
119                 board.getChanceCardDeck().useChanceCard();
120                 player.getPlayer().setBalance(player.getPlayerBalance());
121                 placement = player.getPlacement();
122                 field = board.getField(placement);
123     }
```

```
124
125     if (field instanceof Jail { // Do effect }
126     if (field instanceof FreeParking { // Do effect }
127
128
129     player.setPlayerBalance(-field.getRent())
130
131     currentPlayer++;
132 }
133
134     while (true) {
135         Player winner = board.getWinner();
136         Displaymessage(winner);
137     }
```

7 Konfiguration

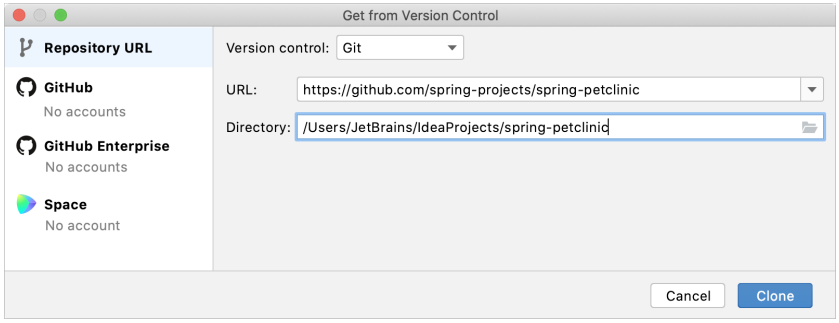
Alle filerne i vores projekt ligger under projektmappen. Rapporten ligger i et LaTex Dokument under projektmappen, samt alle figurer og modeller. Projektmappen er tilstrækkelig når det kommer til at finde alle relevante filer til projektet. Hele projekt bliver versionstyret af git. Derfor bliver ikke kun produktet, men rapporten versionstyret. Det burde derfor være håndterbart at finde den nyeste version, siden den ligger i vores Master-Branch. Hvis filerne ligger i Master-Branchen er det den nyeste funktionelle version. Derfor vil den nyeste version som passer sammen altid ligge i Master Branchen.

Importering af projektet i IntelliJ fra Git [1]

IntelliJ IDEA kan tjekke en eksisterende repository og oprette et nyt projekt baseret på de data man downloader derfra.

1. Fra menulinjen, vælg **Git | Clone**, eller hvis intet projekt er åbnet, bruges **Get from VCS** i **Welcome** skærmen.

2. I dialogboksen **Get from Version Control**, skal du angive URL'en på remote repository du vil clone lokalt.



Figur 12: Vinduet af Get from VCS i IntelliJ

3. Klik på **Clone**. Hvis du vil oprette et projekt baseret på de kilder, du har clone, skal klikker man på **Ja** i bekræftelsesdialogen. Git rodmappen vil automatisk blive sat til projektets rodmappe.

Hvordan kører man programmet?
Programmet køres ved at udføre følgende trin:

1. Hent seneste Java (JRE).

2. Download JAR filen og placer den i et let tilgængeligt sted.

3. Åbn Command Prompt/Terminal og skriv følgende:

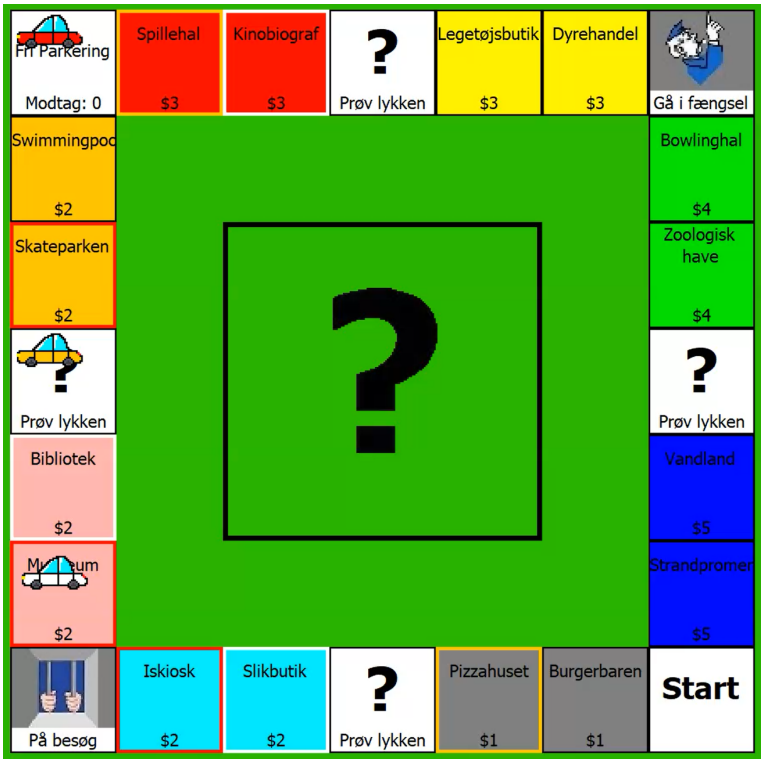
```
java -jar (filens placering.jar)
```

[1] Kilde JetBrains (Okt 2021): https://www.jetbrains.com/help/idea/set-up-a-git-repository.htmlclone-repo

8 Test

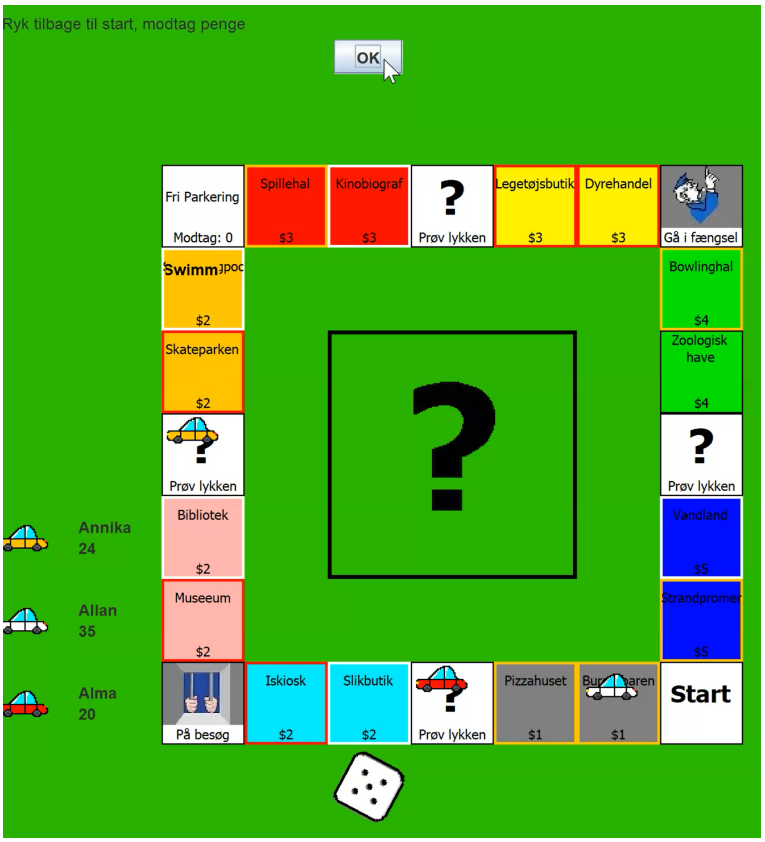
8.1 Brugertest

Vi har lavet en brugertest ud fra 3 forskellige slutbrugere uden kode erfaring, hvor vi har fået feedback som vi kunne bruge til at sammenligne om vores kravspecifikation er opfyldt tilstrækkeligt, og derudfra opdaget nogle bugs som vi selv var klar over.
Ud fra brugertestet blev der observeret, samt nævnt fra slutbrugerer at spillet var intuativt at komme i gang med, på trods af at der ikke var oplyst et regelsæt. I starten af testen var der lidt forvirring over hvordan man købte en ejendom, samt opdaterede balancen dog blev det hurtigt opdaget og var ikke senere et problem. Under brugertesten kom der nogle forslag fra slutbrugere til at øge brugervenligheden, for eksempel at lave farve kanten rundt om et felt man ejer tykkere så den nemmere kunne ses, samt at rykke brikkerne et felt ad gangen så de havde bedre styr på hvor de var på brættet.



Figur 13: Farvekant på felter fra brugertest.

Under brugertesten blev der også noteret et par defekter i spillet, på chancekortet “Ryk frem til start og modtag $2” stod der “Ryk tilbage til start og modtag $2”, når der blev passeret start eller der blev landet på gratis parkering opdaterede spillerens balance ikke altid med det samme men nogle gange først næste gang det var deres tur og i visse tilfælde kan fordelingen af felter kombineret med chancekort gøre at spillet ikke slutter da der fås flere penge end der betales.



Figur 14: Forkert titel på chancekort fra brugertest.

Den sidste defekt skyldes højst sandsynligt at der ikke var taget højde for hvilke chancekort der var blevet implementeret og derfor endte der med at være en større andel chancekort hvor man fik penge i modsætning til at betale.

8.2 Test cases

Tabel 3: Testcase01

Test case ID	**TC01**
Resumé	Test om spillerens placering er tilsvarende terningens værdi.
Krav	R08
Forudsætninger	Spilleren har startet spillet
Post betingelser	Spilleren spiller spillet
Test procedure	Her benyttes setPlacement() metoden til at ændre spillerens placering og der oprettes en terning, der giver en værdi som så gerne skal adderes til spillerens oprindelige position. Dette testes vha. JUnit-metoden assertTrue med betingelsen at getPlacement skal være lig den oprindelige placering plus den nye terningeværdi.
Test data	Terningeværdi (random fra 1-6).
Forventet resultat	True
Faktiske resultat	True
Status	Godkendt.
Testet af	Mark Nielsen
Dato	25/11-2021
Testmiljø	IntelliJ IDEA 2021 2.1 (Ultimate Edition)

Tabel 4: Testcase02

Test case ID	**TC02**
Resumé	Test hyppigheden af facevalues fra 1-6 over 50000 kast.
Krav	R06
Forudsætninger	Spiller starter spillet.
Post betingelser	Spiller spiller spillet
Test procedure	Kører programmet gennem et loop på 50000 kast, hvor der testes for at vores facevalue vil have en værdi fra 1-6, hvilket gøres vha. JUnit-metoderne assertEquals() og assertTrue().
Test data	Loop med 50000 kast for værdier fra 1-6.
Forventet resultat	16% ±1%
Faktiske resultat	16% ±1%
Status	Godkendt.
Testet af	Mads Sørensen
Dato	25/11-2021
Testmiljø	IntelliJ IDEA 2021 2.1 (Ultimate Edition)

Tabel 5: Testcase03

Test case ID	**TC03**
Resumé	Test om spiller ejer felt korrekt.
Krav	R01
Forudsætninger	Spilleren har startet spillet
Post betingelser	Spiller spiller spillet
Test procedure	Her testes der med en teststub, hvor der instantieres objekter af typen Player og Property, hvor setOwner metoden benyttes til at gøre spilleren owneren af property. Der bruges JUnit-metoden assertTrue med betingelsen at getOwner skal være lig spilleren.
Test data	Oprettelse af en Player og en Property. Tjekker herefter betingelsen getOwner er lig Player.
Forventet resultat	True
Faktiske resultat	True
Status	Godkendt.
Testet af	Mark Nielsen
Dato	25/11-2021
Testmiljø	IntelliJ IDEA 2021 2.1 (Ultimate Edition)

Tabel 6: Testcase04

Test case ID	**TC04**
Resumé	Test af at man får korrekte antal spillere når man opretter dem.
Krav	R04
Forudsætninger	Spilleren har startet spillet.
Post betingelser	Spilleren spiller spillet.
Test procedure	Testen blev udført ved at lave en test stub, da metoden createPlayers har funktionen at tilføje spillere til GUI. Derfor laves en metode som fungerer på samme måde, men kun giver logisk tilføjelse af spillere uden den grafiske del. Der benyttes assertEquals til at teste om den det givne antal af spillere som skal oprettes giver det faktiske antal af players.
Test data	Antal spillere sættes til 4.
Forventet resultat	4
Faktiske resultat	4
Status	Godkendt.
Testet af	Mark Nielsen
Dato	25/11-2021
Testmiljø	IntelliJ IDEA 2021 2.1 (Ultimate Edition)

Tabel 7: Testcase05

Test case ID	**TC05**
Resumé	Test af når der trækkes et kort fra chancekortene, så tager den kortet og placerer det trukket kort nederst i bunken.
Krav	R17
Forudsætninger	Spilleren har startet spillet.
Post betingelser	Spilleren spiller spillet.
Test procedure	Bunken fyldes med chancekort og tjekker om det første kort bliver placeret nederst korrekt.
Test data	Placerer første kort nederst i bunken.
Forventet resultat	Første kort bliver placeret nederst i bunken.
Faktiske resultat	Første kort bliver placeret nederst i bunken.
Status	Godkendt.
Testet af	Mark Nielsen
Dato	25/11-2021
Testmiljø	IntelliJ IDEA 2021 2.1 (Ultimate Edition)

Tabel 8: Testcase06

Test case ID	**TC06**
Resumé	Vi har lavet en brugertest ud fra 3 forskellige slutbrugere uden kode erfaring og derudfra opdaget nogle defekter som vi selv ikke var klar over.
Krav	R01, R02, R03, R04, R05, R06.A.B.C, R07, R08, R10, R11, R12.A.B, R15, R18, R19.
Forudsætninger	Spillet kan spilles.
Post betingelser	Spillet spilles færdigt.
Test procedure	Slutbrugere sættet foran et ikke igangsat spil og skal prøve at spille et spil igennem.
Test data	Spillet kan spilles men har små defekter.
Forventet resultat	Spillet kan spilles.
Faktiske resultat	Spillet endte i et loop hvor ingen kunne tabe.
Status	Godkendt.
Testet af	Bertram Kjaer.
Dato	24/11-2021
Testmiljø	64 bit Windows 10 laptop med intel core i5 1.8GHz processor og 8GB ram.

8.3 Test Planlægning

Planlægningen af vores test blev fastsat ud fra vores krav, således at at vi kunne teste hvert enkelte krav og de

8.4 Code Coverage

Element	Class, %	Method, %	Line, %
Account	100% (1/1)	33% (1/3)	33% (2/6)
Board	0% (0/1)	0% (0/20)	0% (0/47)
ChanceCard	100% (1/1)	25% (1/4)	57% (4/7)
ChanceCardDeck	0% (0/1)	0% (0/4)	0% (0/44)
ChanceField	0% (0/1)	0% (0/1)	0% (0/4)
Cup	0% (0/1)	0% (0/3)	0% (0/4)
Die	100% (1/1)	100% (1/1)	100% (4/4)
Field	100% (1/1)	0% (0/1)	50% (1/2)
FreeParking	0% (0/1)	0% (0/5)	0% (0/13)
Jail	0% (0/1)	0% (0/2)	0% (0/9)
Main	0% (0/1)	0% (0/1)	0% (0/83)
Player	100% (1/1)	33% (3/9)	50% (6/12)
Property	100% (1/1)	60% (3/5)	84% (11/...
Start	0% (0/1)	0% (0/1)	0% (0/8)

Figur 15: Tabel over Code Coverage

Det ses ud fra vores code coverage at vi har testet klasserne:

- Account: (Class: 100%, Method: 33%, Line: 33%)
- Chancecard: (Class: 100%, Method: 25%, Line: 33%)
- Die: (Class: 100%, Method: 100%, Line: 33%)
- Field: (Class: 100%, Method: 0%, Line: 33%)
- Player: (Class: 100%, Method: 33%, Line: 50%)
- Property: (Class: 100%, Method: 60%, Line: 84%)

Vi valgte at udføre tests på disse klasser, eftersom vi mente at det var de mest relevante. Derudover kan det selvfølgelig ikke lade sig gøre at teste 100% af programmet, selvom vi gerne ville have testet mere, så havde vi lavet vores Main-klasse med lav samhørighed, hvilket gjorde det svært at teste de forskellige dele.

8.5 Traceability matrix

Requirement number	Test case					
	TC01	TC02	TC03	TC04	TC05	TC06
RS01				X		X
RS02						X
RS03						X
RS04			X			X
RS05						X
RS06A						X
RS06B						X
RS06C						X
RS07		X				X
RS08	X					X
RS10						X
RS11						X
RS12A						X
RS12B						X
R16						X
R17					X	
R19						X
R20						X

Figur 16: Tracebility matrix

8.6 Test Planlægning

Planlægningen af vores test blev fastsat ud fra vores krav, således at at vi kunne teste hvert enkelte krav og dermed konkludere hvilke krav der blev opfyldt og verificeret, hvilket hører under positive tests. Vi benyttede JUnit- og brugertests til at teste vores krav. I forhold til testmiljøet så blev testene kørt i Intellij IDEA 2021 2.1.

8.7 Automatiserede Test

Eftersom vi ikke har lavet TDD, har vi derfor ikke kunne lave automatiseret tests løbende gennem vores udvikling af matadorspillet, da vi først til sidst udførte vores tests med JUnit.

8.8 White box/Black box

Vi har gjort brug af White box test ved at lave JUnit test af vores kode. I forhold til Black box test så har vi fået lavet brugertests, hvor vores produkt er blevet testet af slutbrugere, hvor ingen af dem havde nogen form for programmeringserfaring.

8.9 Integrationstest

Vores tests blev lavet rettere uformelt, altså lavet Big Bang test, hvilket er nogle ting vi vil forbedre os på til næste gang, og i stedet lave TDD, hvilket vil gøre vores fremtidige program mere robust og fejlsikret.

9 Versionsstyring

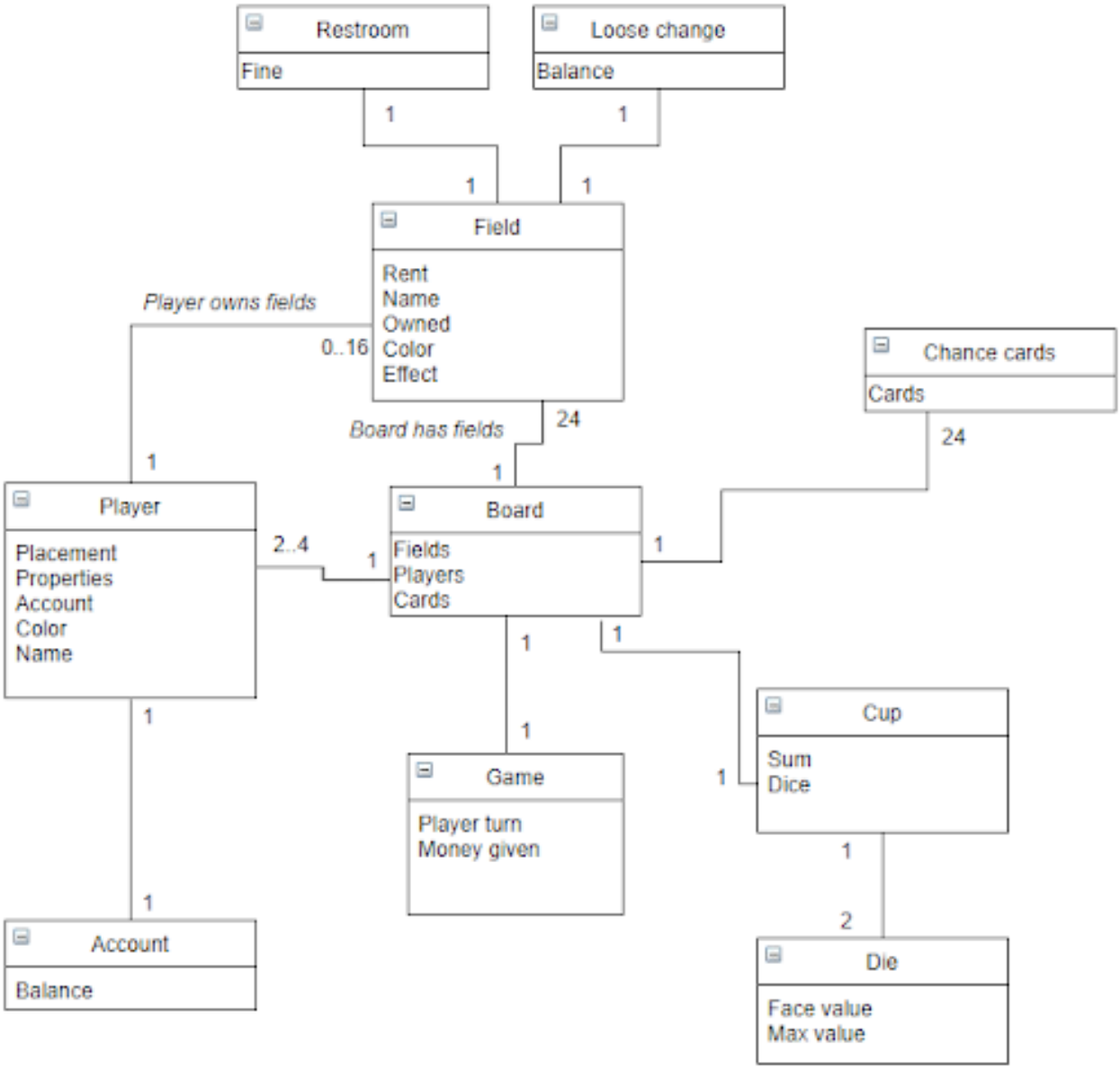
I dette projekt har vi brugt Git som versionstyringsværktøj. Der har minimum været 2 branches i vores projekt på ethvert givet tidspunkt. I vores Master Branch merger vi kun når vi har en prototype som er spilbar med tilstrækkelig nye features. Der har også været en Development Branch. Denne branch blev brugt så tilføjelser af features kunne opstå og eventuelle kode defekter kunne løses effektivt. Der blev også oprettet flere feature branches, test-branches som blev merget med development for at opnår optimal effektivitet af arbejdsopgaver.

Siden alle filer tilhørende projektet er placeret under projektmappen, blev alle vores artefakter versionstyret. De vil altså sige vores tidsplan, LaTex rapport, program og andre artefakter alle blev versionstyret. Dette gjorde vi både for at samle projektet under Intellij og for at effektivt dokumentere alle ændringer optimalt.

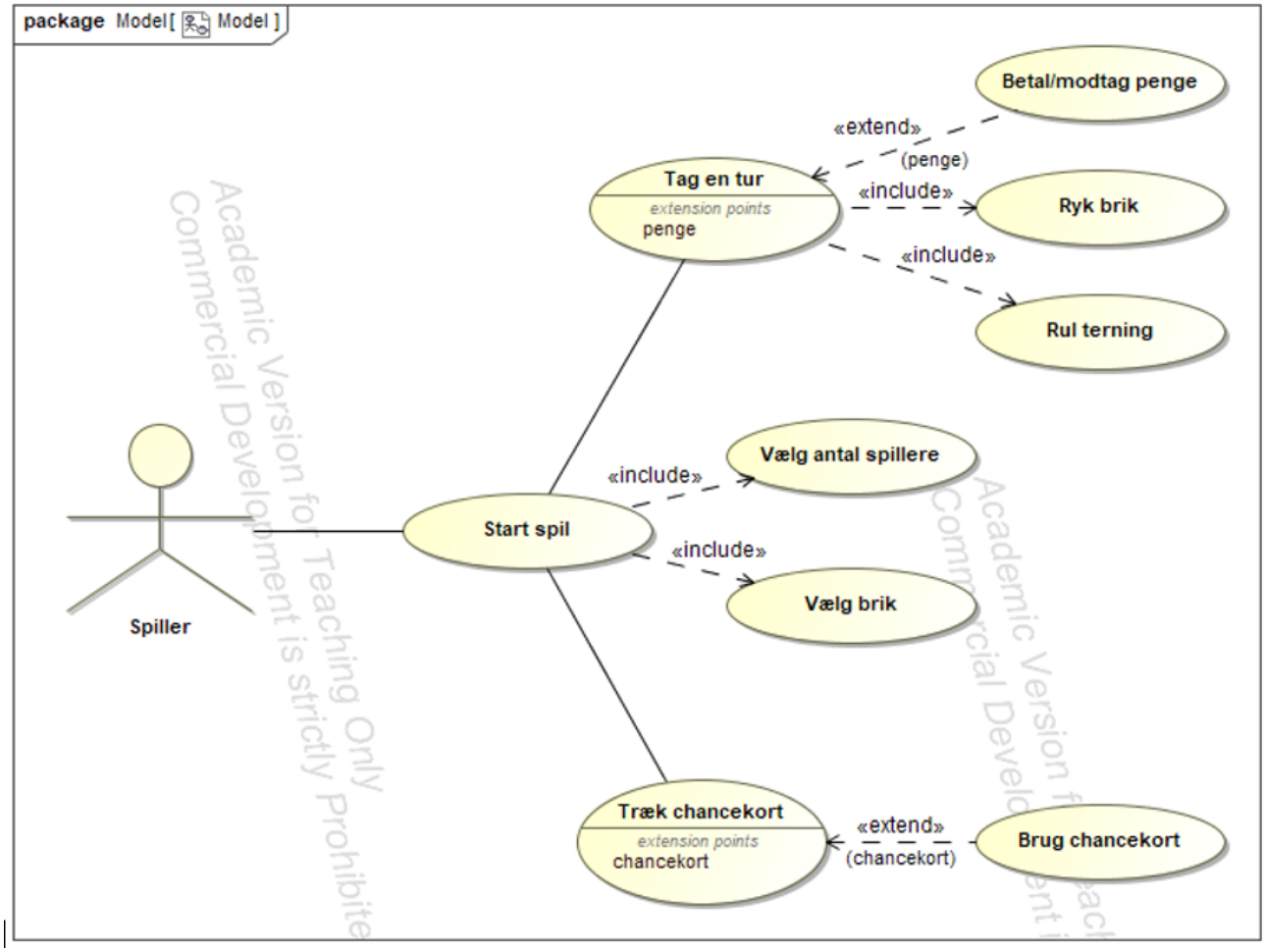
10 Konklusion

I dette projekt fik vi givet opgaven at programmere og dokumentere et junior matadorspil. Vi kan konkludere at vores krav med høj prioritering er færdiggjort. Dette konkludere vi ud fra vores test som grundlag. Vi konkludere på samme grundlag at alle vores krav med prioritering 5 er færdiggjort, samt R15, R16 og R17. Vi konkludere at krav R13 ikke er blevet færdiggjort, da spillet kun har 4 funktionelle chancekort. Vi konkludere at krav R14, A14 og R18 ikke er færdiggjort. Disse krav blev ikke færdigjort grundet deres evaluering af deres prioritering og vansklighed af implementering. Vi vurderede ud fra vores resterende tid at arbejdsresoucer skulle fokuseres på dokumentationen. Vi konkludere at vi at vi har udarbejdet alle artefakter givet i opgavebeskrivelsen under analyse- og designdokumentation. Herunder Kravliste, Use case diagram, use case beskrivelser, Fully dressed use case, Domænemodel, systemsekvensdiagram, sekvensdiagram og Designklassediagram. Vi konkludere at vi har overholdt kravene for implementering, dog har vi ikke toString metoder da vi ikke opprioriterede dem under det givne tidsrum opgaven var givet i. Vi konkludere at vi har overholdt krav til dokumentation givet i opgaven. Vi konkludere at vi har overholdt krav til test som givet i opgavebeskrivelsen. Vi konkludere at vi har løst opgaven i et tilstrækkeligt omfang, både dokumentations- og productmæssigt omfang.

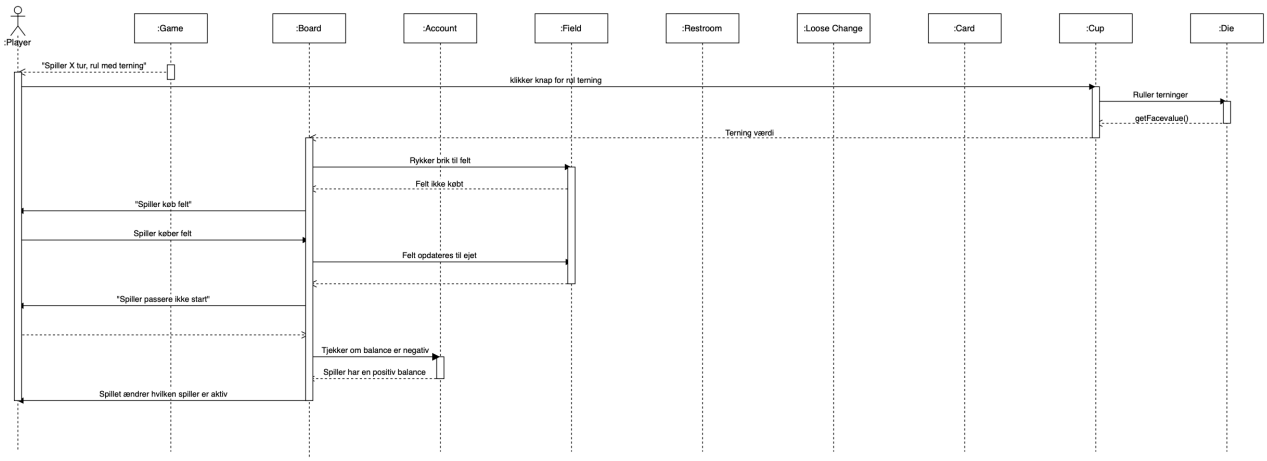
11 Bilag



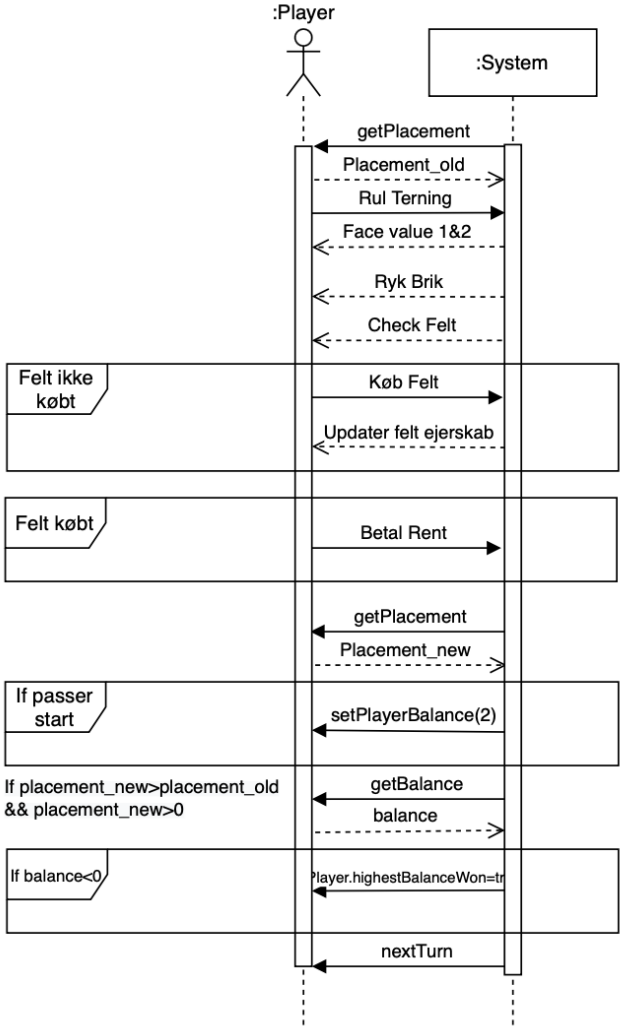
Figur 17: Domænemodel



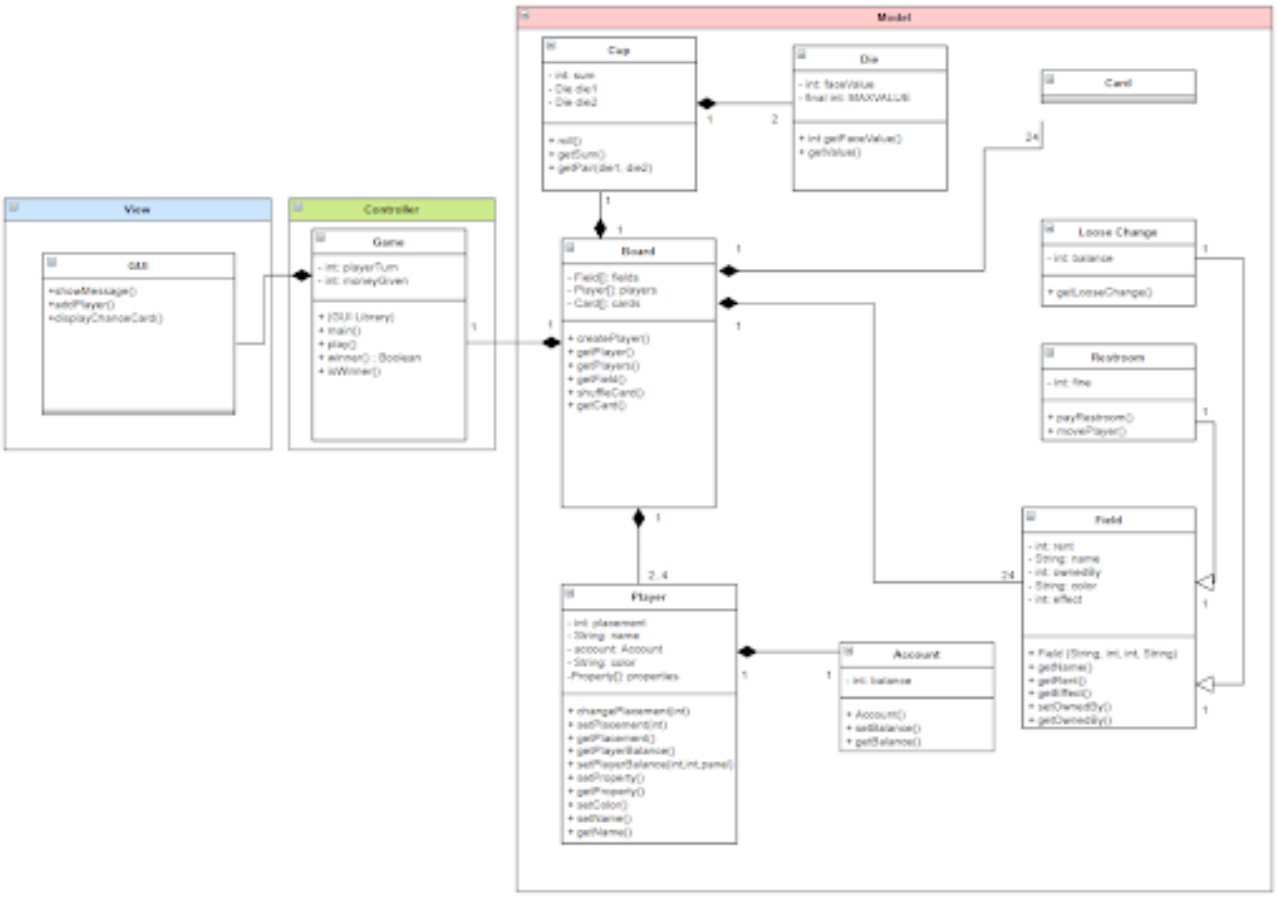
Figur 18: usecasediagram



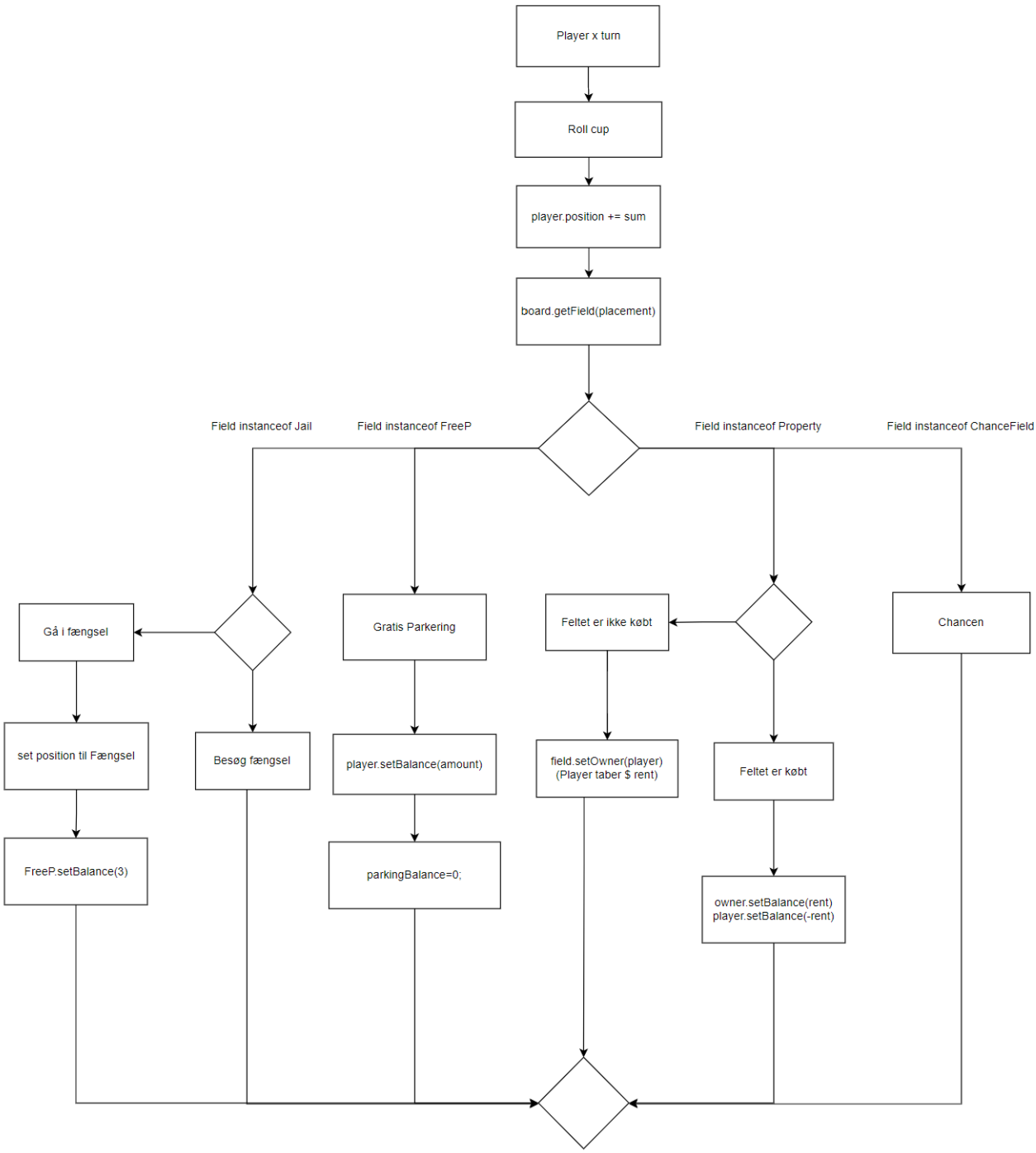
Figur 19: Sekvensdiagram



Figur 20: Systemsekvensdiagram



Figur 21: Desin klasse diagram



Figur 22: Aktivitetsdiagram