

(02312 | 02314) 02313 02315

INDLEDENDE PROGRAMMERING, UDVIKLINGSMETODER TIL IT-SYSTEMER
OG VERSIONSSTYRING OG TESTMETODER

CDIO delopgave 3

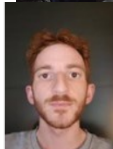
HOLD 12

Christine Reichenbach Nielsen
s141361



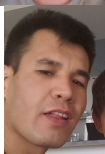
Theis Dahl Villumsen
s195461

Amer Abdulrahman Biro
s176356



Christian Francesco Notarmaso Pone
s195465

Zahra Soleimani
s170422



Mohammad Javad Zamani
s176485

29. november 2019

Indholdsfortegnelse

1	Introduktion	1
2	Krav	2
2.1	Funktionelle krav	2
2.2	Ikke-funktionelle krav	2
3	Analyse	3
3.1	Use Case diagram	3
3.2	Domænemodel	5
3.3	Systemsekvensdiagram	6
4	Design	10
4.1	Design klassediagram	10
4.2	Sekvensdiagram	11
5	Implementering	13
5.1	Die / Shaker	13
5.2	Player	13
5.3	Account / Brik	13
5.4	ChanceKort abstrakt	14
5.5	ChanceDeck	14
5.6	Felt	14
5.7	Felter	15
5.8	BoardController	15
5.9	GUI	16
6	Dokumentation	17
6.1	Arv	17
6.2	Abstract	17
6.3	GRASP	17
6.4	Konfigurationsstyring	19
7	Test	22
7.1	Junit test	22
7.2	Code Coverage	23
7.3	Brugertest	24
8	Konklusion	25
8.1	Tanker til næste projekt	25
	Figurer	26
	Tabeller	27

1 Introduktion

Dette er en rapport til programmet udviklet i forbindelse med projektet *CDIO delopgave 3*, hvor der laves en digital version af "Monopoly Junior"spillet.

Der bruges UML, the unified modelling language, til at designe og analysere spillet. Til analyse bruges Analyse modellerne, Use Case diagram, domænemodel og systemsekvensdiagram. MosCow-metoden bruges til at analysere krav for at vise, hvad der er vigtigst for vores produkt. Til design bruges modellerne, designklassediagrammer, Sekvensdiagrammer og designklassediagram.

Der bliver brugt git-versionskontrol-software i udviklingsfasen af projektet, så alle kan arbejde på samme tid på projektet, siden Github bruges til at dele koden. For at sikre, at vores system fungerer, bruges en kombination af manuelle og automatiske tests.

Koden til dette projekt kan findes via linket: <https://github.com/notarp1/CDI03>.

Vi ender med et produkt, der opfylder vores og kundens vigtigste krav og de fleste af 'bør have' kravene.

2 Krav

I dette tilfælde er en del af systemudviklernes opgave at finde ud af, hvad brugerne ønsker at det nye system kan. Krav er en specifikation af hvad der skal implementeres. En kravspecifikation opdeles efter funktionelle og ikke funktionelle krav.

2.1 Funktionelle krav

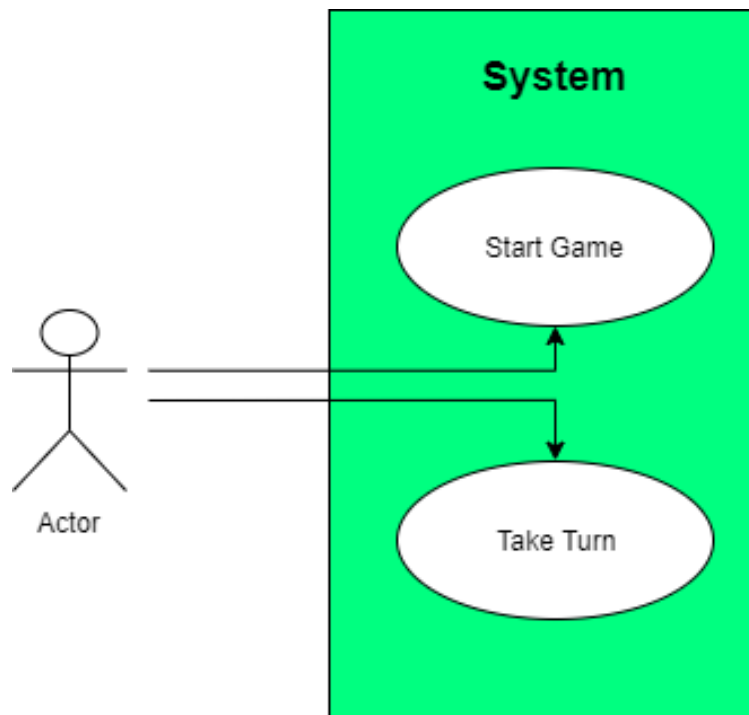
- Den yngste spiller starter! Spillet fortsætter mod venstre.
- Kast terningen og flyt brikken det antal felter, som øjnene viser
- Ryk altid frem, aldrig tilbage
- Hver gang du passerer eller lander på START, modtager du 2M
- Et ledigt felt: Hvis ingen ejer det, SKAL du købe det
- Hvis du lander på et ejet felt, skal du betale prisen af feltet til ejeren
- Hvis du lander på et ejet felt, hvor ejeren også ejer nabofeltet af samme farve, skal du betale det dobbelte af prisen til ejeren
- Man skal betale banken det beløb der står på feltet for at købe det
- Det skal være 2-4 spillere
- Spillerene skal kunne lande på et felt og så forsætte derfra på næste slag
- Man går i ring på brættet
- Når en spiller løber tør for penge, skal de andre tælle deres penge. Spillerne med flest penge vinder

2.2 Ikke-funktionelle krav

- Aktørene har en funktionel computer
- Computeren har installeret JAVA 8

3 Analyse

3.1 Use Case diagram



Figur 3.1: Use case diagram

Vi har lavet et usecase diagram for at få overblik over hvad vores aktør skal foretage sig. Da computeren foretager sig de fleste funktioner, som at købe grunde eller smide spilleren i fængsel, er vi kommet frem til at vi har to usecases. **Start Game** use casen refererer til at spilleren skal starte spillet. **Take Turn** refererer til at spilleren skal tage sin tur.

3.1.1 Brief beskrivelser af use cases

Start Game

- Spilleren starter spillet, spiller 1-4 indtaster deres navn og alder, samt vælger deres ønskede brik. Spillerene bliver præsenteret for deres aktuelle balance. Spillet kan gå i gang.

Take Turn

- Spillerne tager på skift deres tur rundt på brættet. Når en spillers balance går i nul eller minus er spillet slut. Spilleren med flest penge vinder.

3.1.2 Fully dressed beskrivelse af use cases

Use Case: Start Game
ID: 1
Brief Description: Spilleren starter spillet, spiller 1-4 indtaster deres navn og alder, samt vælger deres ønskede brik. Spillerne bliver præsenteret for deres aktuelle balance. Spillet kan gå i gang.
Primary Actors: Spillerne
Secondary Actors: -
Preconditions: Aktørene benytter sig af en funktionel computer som har JAVA 8 installeret
Main Flow: 1: Aktøren starter spiller 2: Aktøren angiver det antal spillere der skal være med (2-4) 3: Spiller x angiver navn 4: Spiller x angiver alder 5: Spiller x vælger sin ønskede brik 6: Systemet opretter en Account til spiller x 7: Spillerne bliver præsenteret for deres aktuelle balance 8: Den yngste spiller kan nu slå
Postconditions: Aktørene kan gå i gang med spillet
Alternative flows: <ol style="list-style-type: none"> 1. Aktøren starter spillet 2. Aktøren angiver det antal spillere der skal være med (2-4) <ol style="list-style-type: none"> a. Aktøren angiver et ugyldigt antal spillere og systemet beder aktøren om at indtaste et gyldigt antal spillere. b. Aktøren angiver et gyldigt antal spillere. 3. Spiller x angiver navn 4. Spiller x angiver alder <ol style="list-style-type: none"> a. Aktøren angiver en ugyldig alder og systemet beder aktøren om at prøve igen. b. Aktøren angiver en gyldig alder 5. Spiller x vælger sin ønskede brik <ol style="list-style-type: none"> a. Aktøren vælger en brik der allerede er blevet taget, systemet beder aktøren om at prøve igen. b. Aktøren vælger en gyldig brik. 6. Systemet opretter en Account til spiller x 7. Spillerne bliver præsenteret for deres aktuelle balance 8. Den yngste spiller kan nu slå

Tabel 3.1: Fully dressed beskrivelse af 'Start Game' use case

Use Case: Take Turn
ID: 2
Brief Description: Spillerne tager på skift deres tur rundt på brættet. Når en spillers balance går i nul eller minus er spillet slut. Spilleren med flest penge vinder.
Primary Actors: Spillerne
Secondary Actors: -
Preconditions: Aktørene benytter sig af en funktionel computer som har JAVA 8 installeret Aktørene har været i gennem use case 1.
Main Flow: <ol style="list-style-type: none"> 1. Den yngste spiller slår med terningerne først 2. Spiller x lander på et felt <ol style="list-style-type: none"> a. Spiller x lander på et frit felt og systemet trækker prisen fra spiller x konto og sætter spiller x som ejer b. Spiller x lander på et felt eget af spiller z. Systemet trækker prisen af feltet fra spiller x konto. Systemet lægger prisen af feltet til spiller z konto c. Spiller x lander på et chancekort felt og systemet præsenterer spiller x for et chancekort d. Spiller x lander på "På besøg i fængsel" og spiller x forbliver på feltet e. Spiller x lander på "Fri parkering" og spiller x forbliver på feltet f. Spiller x lander på "Gå i fængsel" og systemet placerer spiller x på "På besøg i fængsel" feltet og trækker 2M fra spiller x konto. Spiller x modtager ikke 2M for at passere start. g. Spiller x passerer eller lander på start og systemet lægger 2M til spiller x konto. 3. Spiller x konto går i nul eller minus. Systemet præsenterer vinderen af spillet.
Postconditions: Der er fundet en vinder
Alternative flows: <ol style="list-style-type: none"> 1. Spiller x lander på "Gå i fængsel" og systemet placerer spiller x på "På besøg i fængsel" <ol style="list-style-type: none"> a. Systemet trækker 2M fra spiller x konto. Spiller x modtager ikke 2M for at passere start b. Spiller x har "Du løslades uden omkostninger" chancekortet. Systemet trækker <i>IKKE</i> 2M fra spiller x konto. Spiller x modtager ikke 2M for at passere start

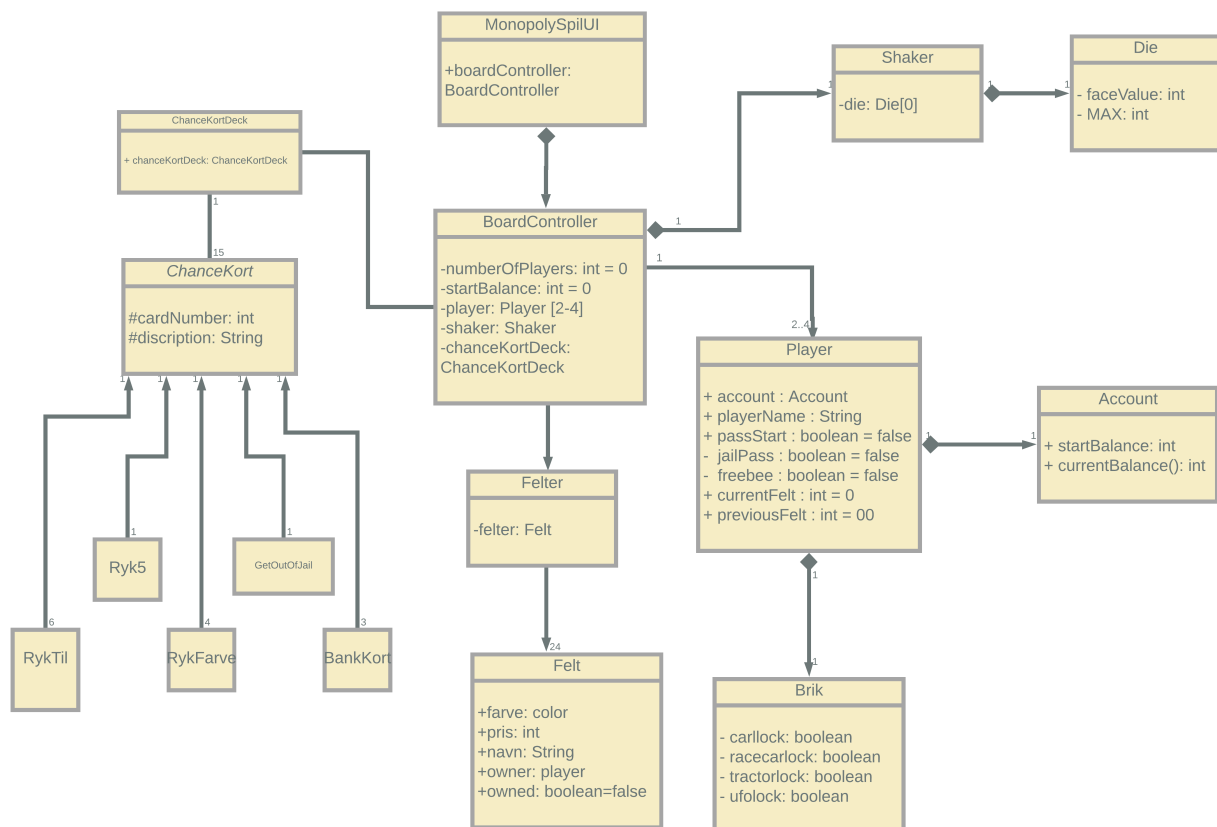
Tabel 3.2: Fully dressed beskrivelse af 'Take Turn' use case

3.2 Domænemodel

For at skabe et bedre overblik over vores use cases, laver vi en domænemodel, der indeholder de vigtige abstraktioner og informationer, der oprettes i analysedelen af udviklingsprocessen, og derefter bruges til at konstruere flere artefakter i designfasen. Domænemodellen til dette

projekt viser de klasser, forhold og attributter vi mener vil være dækkende for at implementere systemet.

Domæne modellen er en meget vigtig del i analyse, fordi ud fra domæne modellen, kan man se hvilke klasser man skal have med i projektet. Når man er i gang med at lave en domæne model, deler man hele projektet op i små mini projekter, så man kan implementere hver klasse for sig, og det er det der kaldes Unified Process. Man deler altså projektet op i små mine projekter, i modsætning til Vandfalds modellen. Efter vi har dannet et overblik over vores domæne model, kunne vi så småt gå i gang med at lave et design klasse diagram, som naturligvis indeholder alle klasserne samt deres attributter og metoder.

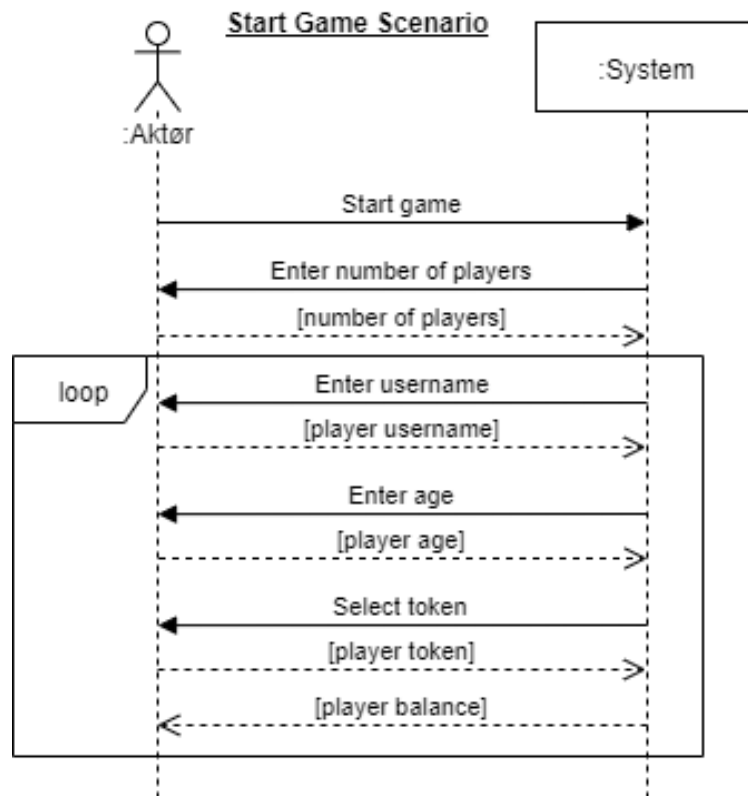


Figur 3.2: Domæne model

3.3 Systemsekvensdiagram

Ud fra vores to fully dressed beskrivelser kunne vi danne os et overblik over hvordan vores spil skulle fungere. Man skal kunne angive navn, alder og brik, samt se sin nuværende balance og spille spillet. Derfor opsatte vi to system sekvensdiagrammer der illustrerer, hvordan vores to use cases skal fungere.

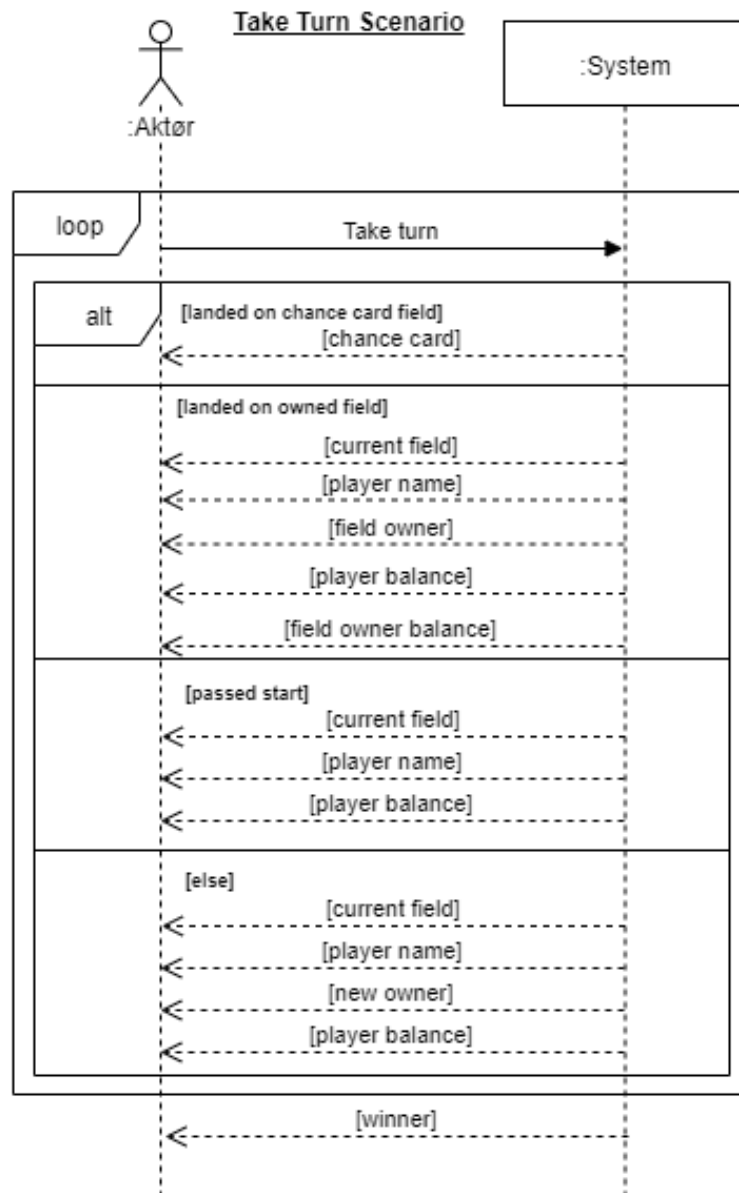
3.3.1 'Start Game' System Sekvensdiagram



Figur 3.3: System sekvensdiagram over use case 'Start Game'

I figur 3.3 har vi illustreret 'Start Game' use casen. Aktøren starter spillet, derefter beder systemet aktøren, om at angive hvor mange spillere der skal være med. Når systemet har registreret det bestemte antal spillere begynder et loop, som looper så mange gange der er spillere. I vores loop beder systemet spilleren indtaste deres navn og alder, samt vælge deres ønskede brik. Til sidst får spilleren returneret sin aktuelle balance. 'Start game' *system sekvensdiagram* slut.

3.3.2 'Take Turn' System Sekvensdiagram



Figur 3.4: System sekvensdiagram over use case 'Take Turn'

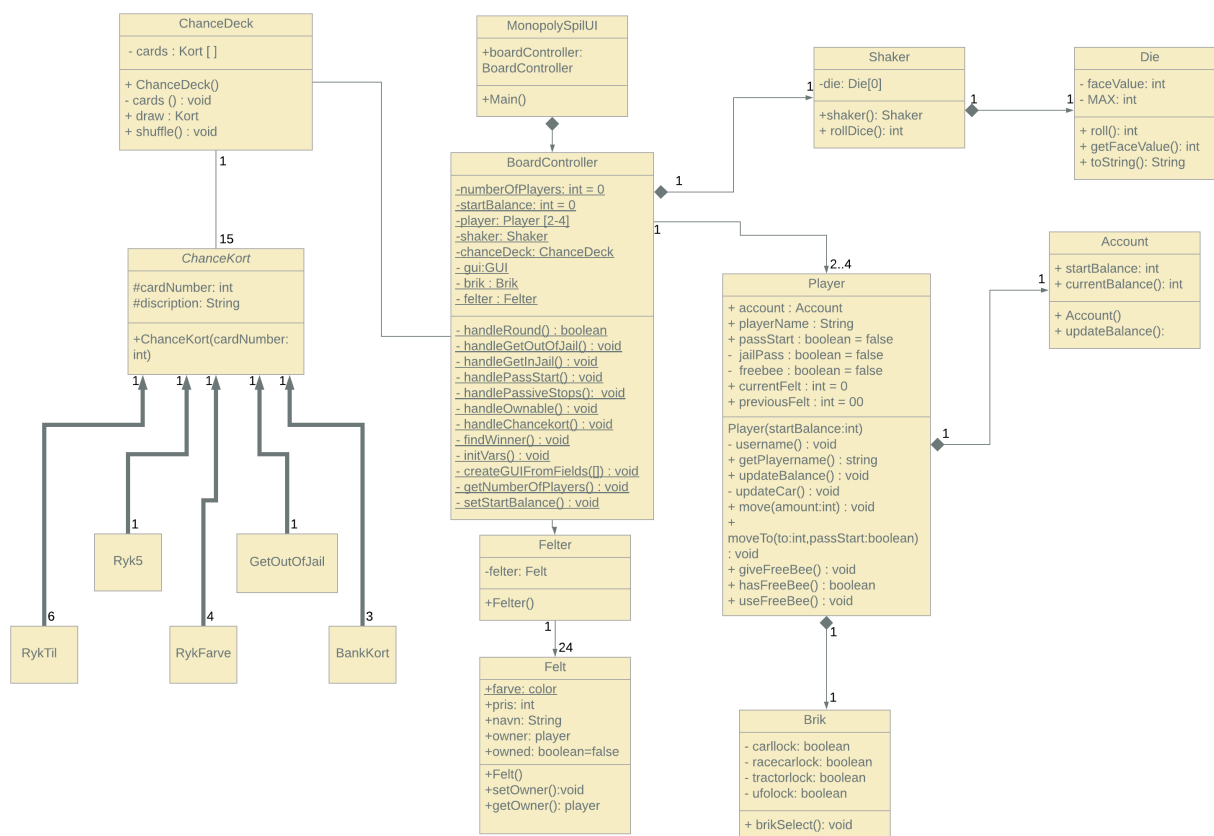
I figur 3.4 har vi illustreret 'Take Turn' use casen. Da dette er et system sekvensdiagram fokuserer vi på hvad selve vores use case indeholder. Man kan tænke at, da dette er et Monopoly spil burde spilleren både skulle købe og sælge ejendomme, samt trække diverse chancekort. Men i vores spil er det **systemet** og ikke **aktøren** der udfører handlingerne. Spillerne tager på skift sin tur og derfor er alle handlingerne sat ind i et loop der først slutter når der er fundet en vinder. Inde i selve loopet har vi en *alternative* som tjekker hvorvidt spilleren er landet på et chancekort felt, et eget felt (af en anden spiller) eller et normalt felt

som ikke er ejet. Hvis spilleren lander på et chancekort felt bliver spilleren præsenteret for et chancekort. Hvis spilleren lander på et allerede ejet felt returnerer systemet navnet på ejeren og deres aktuelle balance m.m. Hvis spilleren lander på et frit felt, køber systemet feltet til spilleren og returnere den aktuelle balance samt den nye ejer på feltet. Hvis Spilleren passerer start, får spilleren 2M ind på kontoen og systemet returnerer den nye balance. Når en spillers balance går i nul er spillet færdigt og systemet returnerer vinderen.

4 Design

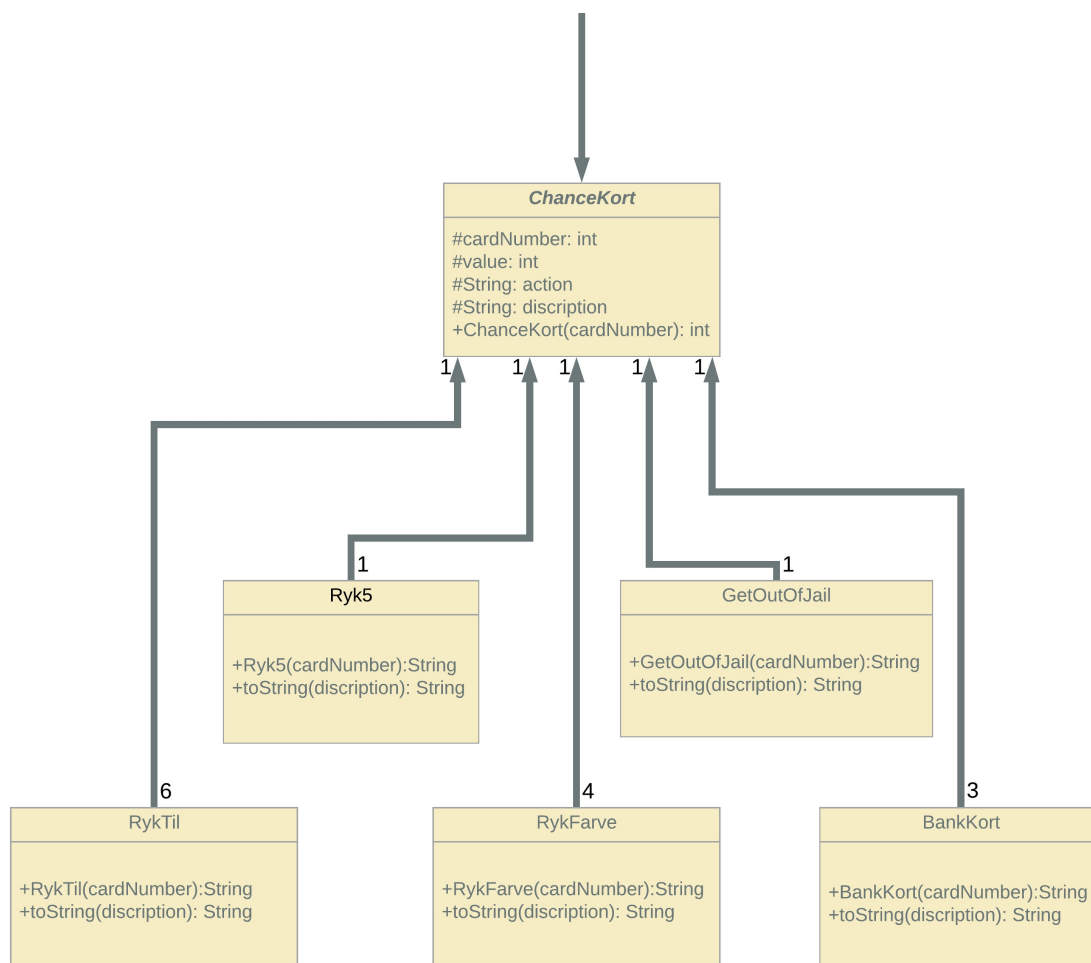
4.1 Design klassediagram

Ved at lave design klasse diagrammer, kan vi dele funktionerne op over nogle forskellige klasser, der hver har forskellige funktioner. At lave design klasse diagrammer, giver os det bedste overblik over hvilke klasser, underklasser, attributter og metoder vi skal have. Vi fik også en god forståelse for hvordan vi skulle komme i gang med implementering af projektet efter vi havde lavet design klasse diagrammer. F.eks. om nogle klasser skal nedarves fra andre klasser eller ej. Diagrammerne viser også associering mellem klasser og underklasser. Associering hjælper os senere i implementering ved at have styr på hvor mange arrays vi skal have af enhver klasse.



Figur 4.1: Design Klasse Diagram

For beskrivelser af klasserne se afsnit 5



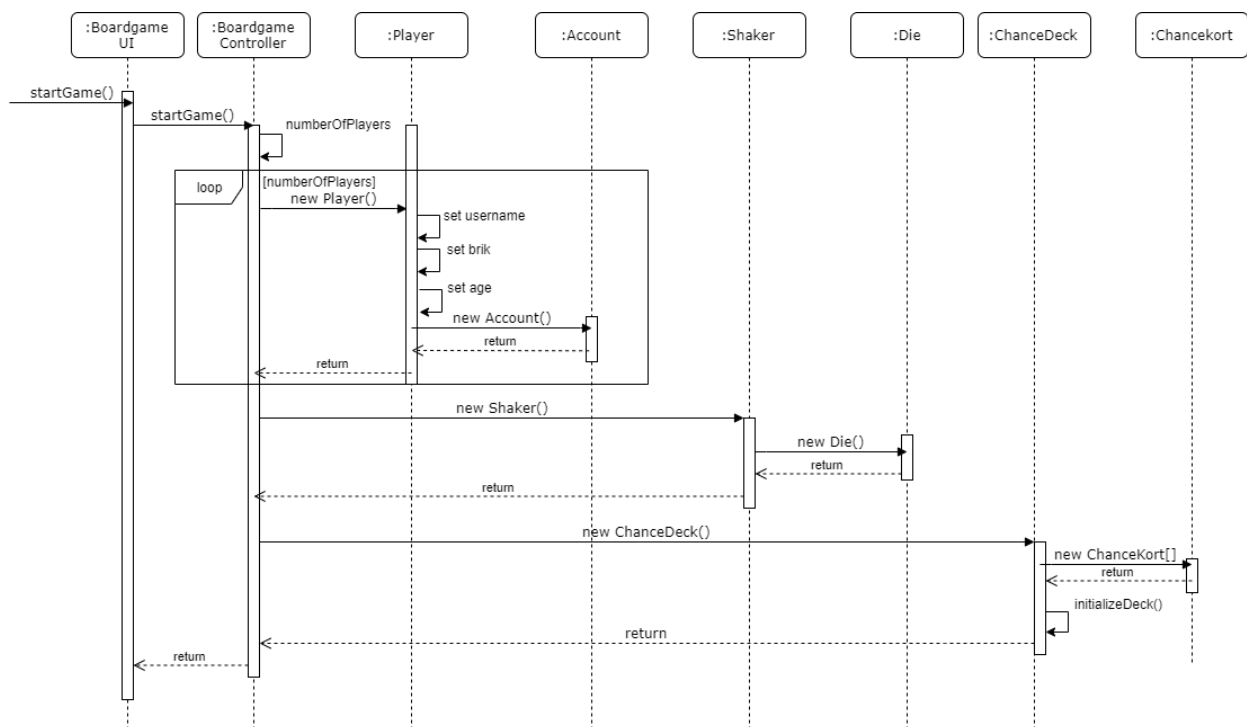
Figur 4.2: Design klassediagram over kort

For dybere forklaring af chancekort underklasserne se afsnit 5.4

4.2 Sekvensdiagram

Vi benytter os af et sekvensdiagram for at skabe sammenhæng mellem logikken og objekterne i vores spil og har lavet et sekvensdiagram til vores use case 'Start Game', som ses på næste side.

4.2.1 Sekvensdiagram af use case: START GAME



Figur 4.3: Sekvensdiagram af use case "Start Game"

Figur 4.3 viser vores sekvensdiagram af use casen "Start Game" og giver os et overblik over hvordan vores spil skal initialiseres inden det går i gang. Det betyder at figur 4.3 skal vise os hvordan instanser af spilleren, kontoen, rafflebægret, terningen, chancedækket og chancekortene skal initialiseres.

En spiller starter spillet, der indtastes hvor mange spillere der skal være med. Vi har et loop der looper så mange gange som antallet af spillere der er med. Det medfører at det korrekte antal Player instanser bliver oprettet. Hver Player instans har metoder som sætter navnet, brikken og alderen på spilleren, samt opretter en Account instans til hver Player.

Herefter bliver vores Shaker oprettet, som opretter et Die objekt. Herefter bliver en Chance-Deck (Chance kort dæk) instans oprettet, som opretter et chancekort array. Herefter bliver kortene initialiseret og spillerne kan begynde deres tur (Use case 2).

5 Implementering

I dette afsnit gennemgås den specifikke implementering af programmet. Klasserne, som blev designet (se afsnit 4.1) gennemgås en af gangen og til sidst gennemgås hvordan det primære loop fungerer og hvordan de tidligere nævnte småting (se afsnit 4.2.1) er implementeret.

5.1 Die / Shaker

Denne klasse er ret simpel, da det stort set bare er en pænere måde at generere et tilfældigt tal mellem to andre, definerede tal. Når klassen instantieres sætter den, dens `facevalue` til 1. Klassen har en metode, `roll()` som genererer et tilfældigt tal mellem 1 og 6, samt en `getFaceValue` som returnerer det tilfældige tal.

Denne klasse er, igen, ret simpel, da det bare er en pænere måde at lave en løkke som genererer tilfældige tal. Når klassen instantierer opretter den en instans af klassen `Dice`, men kun en, da der ikke vil være forskel på at slå en terning to gange eller to terninger en gang. Klassen har en metode `RollDice()` som slår terningen og returnerer *facevalue*en.

5.2 Player

Når `Player` klassen instantieres har den en `int` som parameter der sætter spillerens start balance. Her efter bliver `username`, `age` og `setBrik` metoden kørt, som giver spilleren muligheden for at indtaste sit ønskede navn, indtaste alder og vælge en brik. Ud over det har vi en `getPlayername()` der returnerer spillernavnet og en `updateBalance()` der opdaterer balancen.

`updateCar()` er en metode, som vores GUI skal benytte sig af for at opdatere brikkens placering på spillebrættet. `move(amount:int)` metoden hører til chancekort logikken og giver os muligheden for at rykke vores spiller et bestemt antal felter. `moveTo(to:int,passStart:boolean)` metoden er også chancekort logik og den giver os muligheden for at rykke spilleren til et bestemt felt og samtidig tjekke om spilleren har passeret start og skal modtage 2M eller ej. `giveFreeBee()`, `hasFreeBee()` og `useFreeBee()` høre alle sammen til logikken omkring hvorvidt en spiller ejer et felt eller ej. `useFreeBee()` bliver brugt i sammenhæng med de chancekort hvor man skal rykke frem til et bestemt felt, og hvis der ikke er nogle der ejer det får man det gratis. Altså giver chancekortet spilleren en "*FreeBee*", så når man lander på et fri felt, tjekker feltet om man har en freebee og man skal dermed ikke betale.

5.3 Account / Brik

`Account` klassen er forholdsvis simpel. Den indeholder en konstruktør, som har den ene funktion at den opretter en konto til spilleren. Derudover har `account` klassen også en `updateBalance()` metode, som kan opdatere balancen.

Vores Brik klasse skal indeholde de fire forskellige brikker man kan vælge imellem: en bil, en racerbil, en traktor og en UFO. Vores Brik klasse indeholder en enkelt metode `brikSelect()`, som har til formål at sørge for, at en brik ikke kan vælges igen når den er blevet valgt en gang.

5.4 ChanceKort abstrakt

Inden vi overhovedet gik i gang, valgte vi at gruppere kortene i forskellige kategorier alt efter hvad de har til fælles. Vi kunne se, at vi kunne dele kortene op i 5 forskellige grupper. Således fik vi følgende 5 forskellige underklasser, hvor hoved klassen har følgende attributter.

- `cardNumber` står for at give hvert kort et bestemt nummer, som man kan referere til.
- `description` angiver hver korts fulde beskrivelse.
- `ChanceKort()` er vores konstruktør, som indeholder en parameter `cardNumber` af typen `int`, som arves videre til underklasserne. Det vil sige at alle underklasserne benytter sig af denne konstruktør, således man nemt kan referere til alle underklasserne ved at lave et arraysæt.

Nedarvede klasser (se figur 4.2):

- `RykTil`, `RykFarve`, `BankKort`, `Ryk5` og `GetOutOfJail` har alle følgende metoder til fælles. Nemlig en konstruktør der repræsenterer et `cardNumber` og en `toString` metode der returnerer kortets beskrivelse.

5.5 ChanceDeck

Vores `ChanceDeck` klasse står for at oprette vores sæt med de forskellige chancekort. Vores konstruktør `ChanceDeck()` opretter et array af kort fra `Kort` klassen på 15 kort, som derefter kører vores `cards()` metode. `cards()` indsætter derefter instanser af de forskellige nedarvede klasse objekter fra afsnit 5.4 ind i `Kort` arrayet og vi har nu et dæk med kort.

`draw():void` metoden giver os muligheden for at kunne trække et kort og derefter placere det nederst i bunken. Vores `shuffle():void` metode giver os muligheden for at tilfældiggøre index positionerne i dækket ved hjælp af Javas 'Random' bibliotek.

5.6 Felt

Felt klassen er en meget essentiel del af vores spil. Felt klassen har en konstruktør som definerer hvilket type felt der skal oprettes i vores GUI. Konstruktøren `Felt()` består af et navn, pris og farve, som benytter `Felter` klassen til at danne et array af felt typer.

Ud over det indeholder `Felt` klassen to metoder `setOwner()` og `getOwner()`, der står for logikken omkring hvilken spiller der skal eje et felt og hvilken spiller der ejer et felt.

5.7 Felter

Felter klassen er nødvendig for at kunne vise en korrekt spilleplade i vores GUI. Vores **Felt**- og **Felter**klasser arbejder tæt sammen. Felter klassen har en konstruktør **Felter()** som opretter et felt array på 24 felter. I selve konstruktøren har vi et for-loop der indsætter felternes farve, pris og navn i den ønskede rækkefølge. Vi har nu et array med vores felter vi kan implementere i vores GUI.

5.8 BoardController

BoardController klassen er vores controller der står for at samle alt logikken og sørger for at at vi har et funktionelt spil. Konstruktøren samler alle nedstående metoder, som vi så kan benytte os af i vores GUI for at have et visuelt fungerende spil.

handleGetOutOfJail og **handleGetInJail** metoderne er logikken som tjekker hvorvidt en spiller skal i fængsel eller om spilleren har et frikort og kan undgå nogle af konsekvenserne af at ryge i fængsel.

handlePassStart metoden tjekker hvorvidt spilleren har passeret start og skal modtage 2M i sin balance.

handlePassiveStops metoden holder styr på at der ikke sker noget når spilleren lander på 'Fri parkering' eller 'På besøg i fængsel'.

handleOwnable metoden indeholder alt logikken omkring hvorvidt en spiller skal købe et felt eller betale ejeren af feltet. Den tjekker også om en spiller x, ejer et eller begge felter af samme farve. Således at, hvis begge felter ejes af den samme spiller, så skal spiller y betale det dobbelte beløb af feltet.

handleChanceKort tjekker om en spiller er landet på et chancekort felt og trækker derefter et kort.

handleRound samler alle **handle** metoderne, så vores konstruktør bliver mindre forvirrende at se på.

initVars ligesom **handleRound** segmenterer koden, hvilket gør konstruktøren mere overskuelig, samt bidrager til at problemer/ændringer nemmere kan findes og løses/udføres.

findwinners() tjekker hvorvidt en spillers balance er gået i nul eller er en negativ værdi. Den sørger også for at hvis to spillere har den samme balance, er de begge vindere.

getNumberOfPlayers() benyttes til at tjekke det antal spillere der skal være med.

setStartBalance tjekker ud fra **getNumberOfPlayers** hvad spillerenes startbalance skal være, ud fra om der er 2, 3 eller 4 spillere.

createGUIFromFields arbejder tæt sammen med vores GUI og står for at oprette et korrekt visuelt og funktionelt spillebræt ud fra vores felter.

5.9 GUI

Det er kendt, at vi skal bruge GUI til udførelse af spillet ved hjælp af maven (se afsnit 6.4 for bredere forklaring).

Uden et GUI (Grafical User Interface) ville vores spil skulle spilles i en terminal, med et GUI har vi lige pludselig et spillebræt med navne, balancer, felter og brikker. Vi fik udleveret en allerede færdig GUI, et bibliotek (dependency) af metoder, et visuelt spilbræt og diverse sprites. For at gøre brug af GUI'et lavede vi vores *MonopolySpilUI* klasse, som tager alt vores logik og sammenkobler det med grafik. Klassen indeholder vores `main()` metode, som tager brug af vores `BoardController` til at starte et fungerende spil med et grafisk interface.

Metoder fra GUI bibliotek vi har benyttet er:

`new GUI(fields)` som opretter vores spilbræt ud fra vores `Felt` array.

`getUserButtonPressed()` opretter en box med tekst og en knap. Dette har vi benyttet os meget af, da det gav os muligheden for at vise en besked, samt pause spillet og starte spillet når en spiller klikker på den nævnte knap.

`displayChanceCard()` gav os muligheden for at vise et `chanceKort`.

`setDie()` sætter den grafiske terning til det man har slået.

`showMessage()` viser en besked.

6 Dokumentation

Fortæl hvad det hedder hvis alle fieldklasserne har en `landOnField` metode der gør noget forskelligt. Det kaldes polymorfi.

6.1 Arv

Arv er en software udviklingsteknik, som anvendes i objekt orienteret programmering. Konceptet i processen er at en ny klasse (en subklasse) bliver afledt fra en anden klasse, altså der nedarves. Når der nedarves kommer subklassen til at indeholde variablerne og metoderne fra klassen der nedarves fra. Man kan derefter foretage ændringer i subklassen, for at få en mere specifik version af den originale klasse. Ved anvendelse af nedarvning får man altså genbrugt noget af den kode man allerede har skrevet, hvilket kan være mere effektivt.

6.2 Abstract

Abstrakte klasser er klasser som ikke bliver instantieret. Den abstrakte klasse bliver oftest brugt til nedarvning. I dette tilfælde er den abstrakte klasse en superklasse der indeholder nedarvede klasser, der har en bestemt funktion. Man kan bruge abstraktion til f.eks. at lave en bestemt klasse, der indeholder nogle nedarvede klasser, hvor man kun kan lave instanser af de nedarvede klasser, men ikke af hoved klassen.

Dette kan ses i vores `ChanceKort` klasse diagram, hvor `ChanceKort` klassen er en abstrakt klasse, der bliver brugt til at nedarve i 5 klasser, som er `RykTil`, `Ryk5`, `GetOutOfJail`, `RykFarve` og `BankKort`. På den måde behøver vi ikke at skrive koden igen, i de 5 subklasser. (figur 4.2) Vi har nemlig brugt abstraktion i dette tilfælde, fordi vi skulle have en abstrakt kort klasse, som indeholder 5 forskellige nedarvede klasser. Så vi kunne lave et arraysæt af kort, således vi kunne give hvert array en bestemt værdi, alt efter hvad dette array skal repræsentere. På den måde har vi også brugt polymorphism til at generere alle kortene. Attributter i hoved klassen er `protected`, fordi de kun bruges i nedarvede klasser. Når man bruger abstraktion, skal det abstrakte klasse navn stå i kursiv, og det gør vores `ChanceKort` klasse også.

6.3 GRASP

GRASP er et rigtigt vigtigt begreb i implementering. Når man skal lave et nyt projekt, er det meget vigtigt at dele ansvar og funktioner op i så mange nødvendige "klasser" så muligt. GRASP står 'General Responsibility Assignment Software Patterns' og indeholder følgende begreber:

Creator

Information Expert

Main klasse kender til de andre under klasser, mens de under klasser ikke kender til main klassen.

Low Coupling

Betyder at der skal være mindre afhængighed mellem klasserne. Opret aldrig en associering bare for blot at genbruge kode fra en anden klasse .

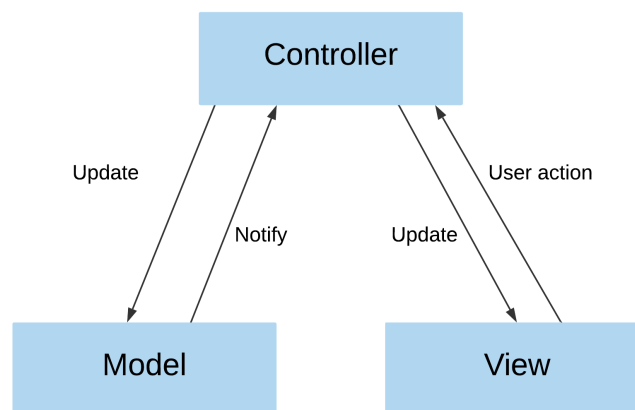
Vi har anvendt low coupling i projektet, som giver os muligheden for at kunne ændre koden i en

klasse, uden at den ændring påvirker andre dele af vores projekt.

High Cohesion

betyder at enhver klasse ikke skal have for meget ansvar. Så det relaterer til Creator, nemlig når vi deler vores projekt op i små bider, så én klasse ikke skal stå for det hele. Polymorphism
Vi har også anvendt polymorphism i vores projekt, som det er blevet forklaret under abstraktion.

Controller, Model og View



Figur 6.1: Controller, Model og View

Controller, Model og View packages er en stor del af GRASP, som sørger for programmeret bliver mere overskueligt og nemmere at ændre. De medvirker begge til brug af 'Low Coupling'. I vores projekt har vi placeret vores GUI i 'View', vores BoardController i 'Controller', da den er ansvarlig for håndteringen af user inputs og alt logikken, samt alle vores resterende klasser ind i 'Model'. figur 6.1 illustrerer hvordan disse 'packages' samarbejder. I 'View', vores GUI, laver aktøren en user action, som eksempelvis at slå med terningerne. Denne user action bliver kørt i Controlleren, som manipulerer klassernes data i Model, der derefter giver Controlleren besked og til slut opdaterer GUI'en. Dette resulterer i low coupling (en uafhængighed) mellem model og view, som gør programmet meget nemmere at teste og ændre.

Segmentering

En anden ting der bidrager til GRASP er segmentering af koden. Her har vi sørget for at vores BoardController konstruktør er hyper simpel. Det kode figur 6.2 indeholder er det eneste vores 'View' package skal kalde for at køre hele vores spil. Som set i afsnit 5.8, har vi samlet en masse metode i enkelte metoder og det er hvad der kaldes segmentering. Segmentering resulterer i at det blive meget nemmere at indskrænke hvor problemer opstår eller hvor ændringer skal foretages.

```
1 package Controller;
2
3 public class BoardController {
4     // [...]
5     public BoardController(){
6         initVars();
7
8         boolean playing = true;
9         while (playing) {
10             for (Player player : players) {
11                 playing = handleRound(player);
12                 if(!playing)
13                     break;
14             }
15         }
16         findWinners();
17         System.exit(0);
18     }
19     // [...]
20 }
```

Figur 6.2: Udklip af BoardController konstruktøren

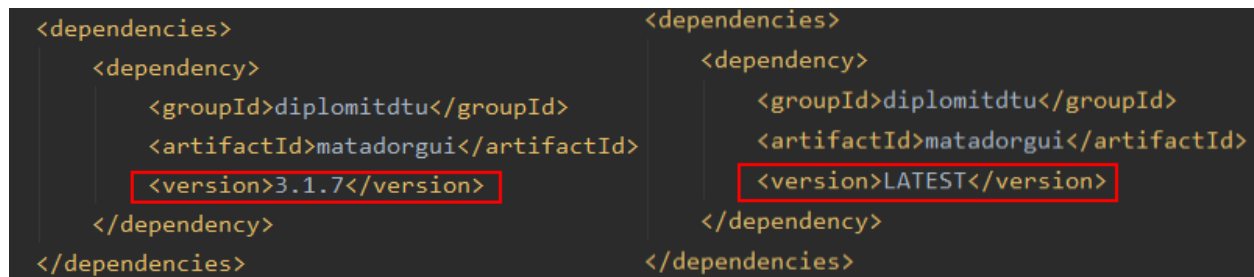
6.4 Konfigurationsstyring

Konfigurationsstyring er en vital del af helheden af ens projekt. Da projekter tit kan være meget store er det vigtigt at benytte sig af konfigurationsstyring for at undgå ting som misforståelser, fejl i versionstyringen eller måske bare miskommunikation.

Når man benytter sig af versionsstyring kan det hurtigt blive forvirrende hvis de forskellige udviklere, begynder at importere ukendte biblioteker i massevis. Derfor kan man heldigvis nu benytte sig af Maven, som hoster netop disse biblioteker og holder hele dit projekt opdateret.

Maven er som sagt en kæmpe internet cloud, der hoster tusindevis af forskellige Java biblioteker eller *packages* på engelsk. Hvis lige netop dit bibliotek ikke findes i Maven, kan du nemt tilføje det så alle dine kollegaer kan benytte sig af det.

Det smarte ved Maven er muligheden for at holde sine biblioteker up-to-date for at undgå fejl, og det gør netop Maven ved hjælp af en POM fil (*Project Object Model*), som bliver oprettet i projektet. POM er en .xml fil som indeholder alle projektets detaljer, så som projektets aktuelle version, repositories og biblioteker (*dependencies*). Så hvis man har tilføjet en ny dependency og *pusher* sin branch i IntelliJ, så vil ens kollegaer automatisk få opdateret sit projekt med de seneste tilføjede java-biblioteker. Hvis man vil være sikker på ens biblioteker er opdateret til den nyeste version kan man benytte sig af en 'LATEST' kommando se figur 6.3.



```
<dependencies>
  <dependency>
    <groupId>diplomitdtu</groupId>
    <artifactId>matadorgui</artifactId>
    <version>3.1.7</version>
  </dependency>
</dependencies>
```

```
<dependencies>
  <dependency>
    <groupId>diplomitdtu</groupId>
    <artifactId>matadorgui</artifactId>
    <version>LATEST</version>
  </dependency>
</dependencies>
```

Figur 6.3: Dependencies 'LATEST' i brug

6.4.1 Versionsstyring

For at holde styr på vores kildekode benyttede vi os af git. Vi benyttede os af GitHub som hoster vores repositories online og ved hjælp af det kunne vi nemt skabe overblik over alle ændringer foretaget i projektet.

Hvis vi kikker bort fra GitHub og ville importere vores projekt manuelt kan man benytte sig af forskellige commandlines i et git terminal. For at tilføje et git repository i ens projekt, finder man projektes filplacering og skriver følgende i terminalen

```
$ cd /c/user/my_project
$ git init
```

Efter man har tilføjet git er det vigtigt at man laver sit første commit med:

```
$ git commit -m 'commit besked'
```

Nu har man lavet sin master branch og kan derefter begynde at tilføje flere branches til evt. development og bug fixing. For at netop gøre det skriver man følgende

```
$ git branch branchnavn
```

Hvis man vil skifte den branch man arbejder på skriver man

```
$ git checkout branchNavn
```

Hvis man har et færdigt projekt og gerne vil bygge alle filerne som program i IntelliJ gør man følgende inde i IntelliJ

```
file > Project Structure > Artifacts > Add Jar > From modules with dependencies
```

Derefter vælger sin main klasse og klikker på

```
Build > Build Artifacts > Build
```

Man vil nu kunne finde sin eksekverbare fil under

<MitProjekt/out/artifacts/MitProjekt.jar>

7 Test

7.1 Junit test

I Java kan vi bruge et framework der hedder JUnit til automatisk test. I dette projekt er version 4 taget i brug. JUnit er blevet tilføjet via Maven, så alle har samme version.

Der er skrevet tests til de 5 mest simple klasser, da det ellers går hen og bliver for kompliceret, da det kræver brugerinput. Klasserne der er testet er: *Player*, *Account*, *Felt*, *Die*, *Shaker*.

Disse klasser er valgt, da de giver en stor del af basis funktionaliteten i koden og det er essentielt at de virker korrekt, da det går ud over en ellers korrekt logik, hvis der er fejl i de metoder og klasser der bruges.

For ikke at få en alt for lang rapport, vil udvalgte tests blive beskrevet.

```
1 package Model;
2 import org.junit.Test;
3 import static org.junit.Assert.*;
4
5 public class PlayerTest {
6     // [...]
7     @Test
8     public void move() {
9         Player player = new Player(1000, "P1");
10        player.move(10);
11        assertEquals(10, player.currentFelt);
12        assertEquals(0, player.previousFelt);
13        assertFalse(player.passedStart);
14        player.move(20);
15        assertEquals(6, player.currentFelt);
16        assertEquals(10, player.previousFelt);
17        assertTrue(player.passedStart);
18    }
19    // [...]
20 }
```

Figur 7.1: Udklip af unit test af *Player* klassen

Titel: Player - Flyt brik (se figur 7.1)

Beskrivelse: En spiller skal kunne rykke rundt på pladen og det skal huskes hvor de var før. Det skal også tjekkes om de har passeret start, så de kan få udbetalt den bonus.

Trin:

1. Opret en instans af en *Player*
2. Flyt spilleren mindre end 24 felter
3. Tjek resultatet
4. Flyt spilleren nok til at spilleren i alt har flyttet mere end 24 felter

5. Tjek resultatet

Forventet resultat: Første resultat forventes at være at spilleren er blevet rykket til det givne felt og at det afspejles i historikken. Spilleren skal ikke have passeret start. Andet resultat forventes at være at spilleren er blevet rykket igen og historikken afspejler dette og at spilleren nu har passeret start.

```
1 package Model;
2 import static org.junit.Assert.*;
3
4 public class AccountTest {
5     // [...]
6     @org.junit.Test
7     public void positiveUpdateBalance() {
8         Account account = new Account(1000);
9         account.updateBalance(100);
10        assertEquals(1100, account.balance);
11    }
12    @org.junit.Test
13    public void negativeUpdateBalance() {
14        Account account = new Account(1000);
15        account.updateBalance(-100);
16        assertEquals(900, account.balance);
17    }
18 }
```

Figur 7.2: Udklip af unit test af Account klassen

Titel: Account - Ret balance (se figur 7.2)

Beskrivelse: En spiller skal kunne bruge og modtage penge, det er derfor vigtigt at teste om Account klassen kan håndtere både positive og negative tal.

Trin:

1. Opret en instans af en Account
2. Tilføj eller fjern en mængde fra kontoen
3. Tjek resultatet

Forventet resultat: At kontoens balance er blevet justeret korrekt.

7.2 Code Coverage

Code coverage er en test der kan køres, for at se hvor meget af den udviklede kode der reelt bliver eksekveret.

IntelliJ har en indbygget funktion til at udføre denne test, mens koden eksekveres.

I dette tilfælde ser det ud til at den har fejlet, da det eneste der blev analyseret var filen *src/main/java/View/MonopolyUI.java*, som er stort set tom, da dens opgave er at starte kontrolleren, hvilket giver en coverage på 100%, hvilket antages ikke er helt korrekt.

7.3 Brugertest

Brugertest er en test som er udført af en slutbruger uden kendskab til indmaden i programmet. man giver en specifik opgave til brugeren og observere derefter brugerens handlinger. Man beder også brugeren om at tænke højt og fortælle om evt. frustrationer med brugen af programmet. Målet med denne test er at se programmet fra et eksternt synspunkt, hvilket ofte resulterer i at man opdager mangler og fejl i programmet som man ellers ikke ville have opdaget.

Titel: Brugertest - Kan en bruger finde ud af at spille spillet

Beskrivelse:

Antagelser: At der på forhånd er sat en computer op, hvor spillet allerede kører og at der er et sted mellem 2 og 4 brugere til stede til at teste spillet.

Trin:

1. Brugerne bliver placeret ved computeren
2. Brugerne får at vide at de skal spille spillet ind til en af dem vinder

Forventet resultat: At spillet bliver gennemført og brugerne ikke har behov for at stille spørgsmål.

8 Konklusion

Vi har udviklet et Monopoly Junior spil efter kundens krav. Vi har udarbejdet en analyse, der kunne hjælpe os med at komme i gang med at kunne forstå trin for trin hvordan vi kan bygge vores spil. Vi har benyttet os et use-case diagram, en domæne model, et system sekvensdiagram, et skevensdiagram samt et design klassediagram. Vi har opfyldt alle kravene undtaget at den "yngeste spiller starter først", da vi desværre ikke fik tid til at implementere det. Tilgængæld har vi overholdt vores tidsplaner i gruppen og afleveret vores produkt og rapport inden deadline. Vi har også implementeret GUI og samt udført test. Vi har lært meget igennem dette projekt, og har allerede fået nogle tanker til hvordan vi nemmere og hurtigere kan komme i gang med vores næste store projekt.

8.1 Tanker til næste projekt

Gruppen har lært meget, noget af det blev opdaget så sent i processen at der ikke var tid til at implementere det i koden. Disse eksempler vil blive nævnt her.

Abstrakte klasser og interfaces

Gruppen har læst noget teori om hvordan abstrakte klasser og interfaces virker. Dette havde været meget brugbart at have dybere kendskab til, da felterne og chance-kortene blev udviklet, da det ændrer drastisk på hvor komplicerede klasserne har behov for at være og hvor læsbar koden er.

Læs dokumentationen

I forbindelse med det GUI, der skulle bruges, har der været et par eksempler på "Hvordan er det lige det virker?", som kunne have været løst, hvis man læste i den dertilhørende java-doc. Givet, så var det meget svært at finde linket til java-doc, hvilket gruppen ikke rigtig har indflydelse på.

Figurer

3.1	Use case diagram	3
3.2	Domæne model	6
3.3	System sekvensdiagram over use case 'Start Game'	7
3.4	System sekvensdiagram over use case 'Take Turn'	8
4.1	Design Klasse Diagram	10
4.2	Design klassediagram over kort	11
4.3	Sekvensdiagram af use case "Start Game"	12
6.1	Controller, Model og View	18
6.2	Udklip af BoardController konstruktøren	19
6.3	Dependencies 'LATEST' i brug	20
7.1	Udklip af unit test af Player klassen	22
7.2	Udklip af unit test af Account klassen	23

Tabeller

3.1	Fully dressed beskrivelse af 'Start Game' use case	4
3.2	Fully dressed beskrivelse af 'Take Turn' use case	5