

62532 - Versionsstyring og testmetoder E21

Rapport CDIO Del 3

Projektgruppe: 11

November 30, 2021



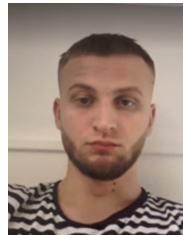
Sofie Groth Dige
[s211917](#)



Anshjyot Singh
[s215806](#)



Nick Tahmasebi
[s195099](#)



Miljkov Hansen
[s194302](#)



Technical University
of Denmark

1 Timeregnskab

Timeregnskab

Nick	25 timer
Marco	25 timer
Ansh	30 timer
Sofie	30 timer

Indholdsfortegnelse

1 Timeregnskab	2
2 Indledning	4
3 Projektplanlægning	4
4 Guide til Github repository	4
5 Analyse	7
5.1 Use-case diagram	7
5.2 Kravspecifikationer	7
5.3 Riskmanagement	13
5.4 P*I Matrix	14
6 Dokumentation	15
6.1 Arv	15
6.2 landOnField og Polymorphism	15
6.3 Abstract	15
7 Konfigurationsstyring	16
7.1 Maven	16
8 Design	17
8.1 Domænemodel	19
8.2 Design klassediagram	19
8.3 System Sekvensdiagram	21
8.4 Sekvens diagram	22
9 GRASP-PATTERNS:	22
10 Implementering	24
10.1 Kode:	24
10.2 GUI-implementation	25
11 Test	27
12 Konklusion	30

2 Indledning

I denne CDIO opgave har vi udviklet et program kaldet Junior Monopoly til vores kunde, som vi har kodet i programmeringssproget Java, i IDE-programmet IntelliJ. Der vil i dette tværfaglige projekt med kurserne Indledende programmering, Versionsstyring og testmetoder, Udviklingsmetoder for IT-systemer tages udgangspunkt i udviklingen af et brætspil. Det er et 2-4 player spil hvor det handler om at være den sidste tilbage som ikke er bankerot. Spillebrættet består af 32 felter. Hver spiller slår med en terning, og rykker antallet af øjne hen til et felt. Felterne påvirker spillerens bankkonto. Der er chancekort som kan påvirke spillerens bankkonto men som også kan påvirke spillerens position på spillebrættet. Formålet med dette projekt er at implementere så mange regler fra monopoly som muligt, spillet skal kunne køre og det er et springbræt til det næste projekt som er det fulde matador-spil.

Kundens vision:

Kunden har en vision om, at der skal udvikles på et Monopoly Junior spil. Der skal implementeres de væsentligste elementer for at spillet kan spilles, regler må udelades. Spillet skal dog kunne spilles mellem 2-4 personer.

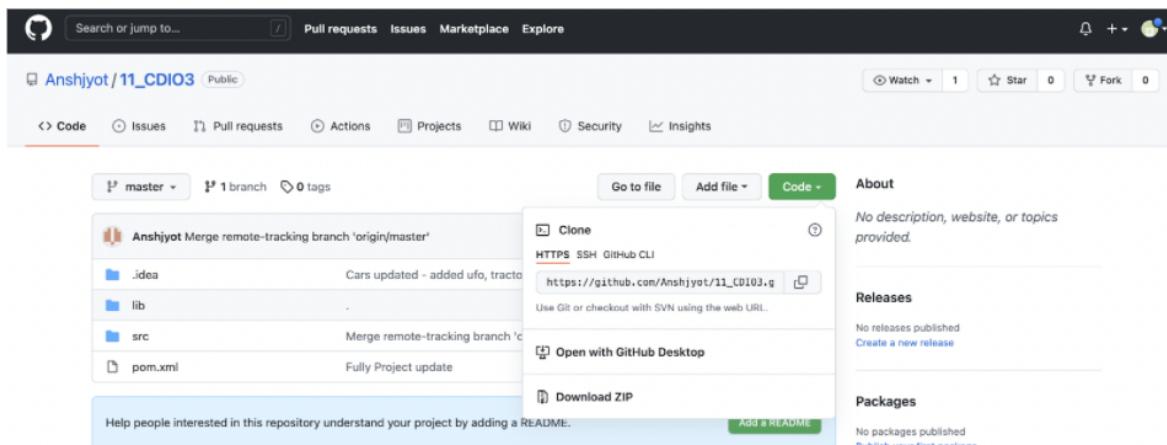
3 Projektplanlægning

Før vi, i gruppen gik i gang med selve projektet, så sad vi i fællesskab og sørgede for at klargøre en plan for hvordan vi skulle tilgå denne opgave. En måde hvorpå vi kunne tydeliggøre prioriteringerne i vores projekt, samt lave en intern aftale som lød på hvilke krav der først skulle løses før vi fortsatte arbejdet mod andre krav eller forespørgsler der var til projektet. Herunder ses en NeedToHave og NiceToHave liste som beskriver hvilke krav der SKAL løses og hvilke krav der kunne være FEDT at løse, hvor ”Need to have” kravene naturligvis var øverst i vores prioriteringsliste. Vi mente også at planlægningen for arbejdet skulle forbedres, forbedres ift. ansvarsområder og bedre kontakt mellem gruppemedlemmerne.

4 Guide til Github repository

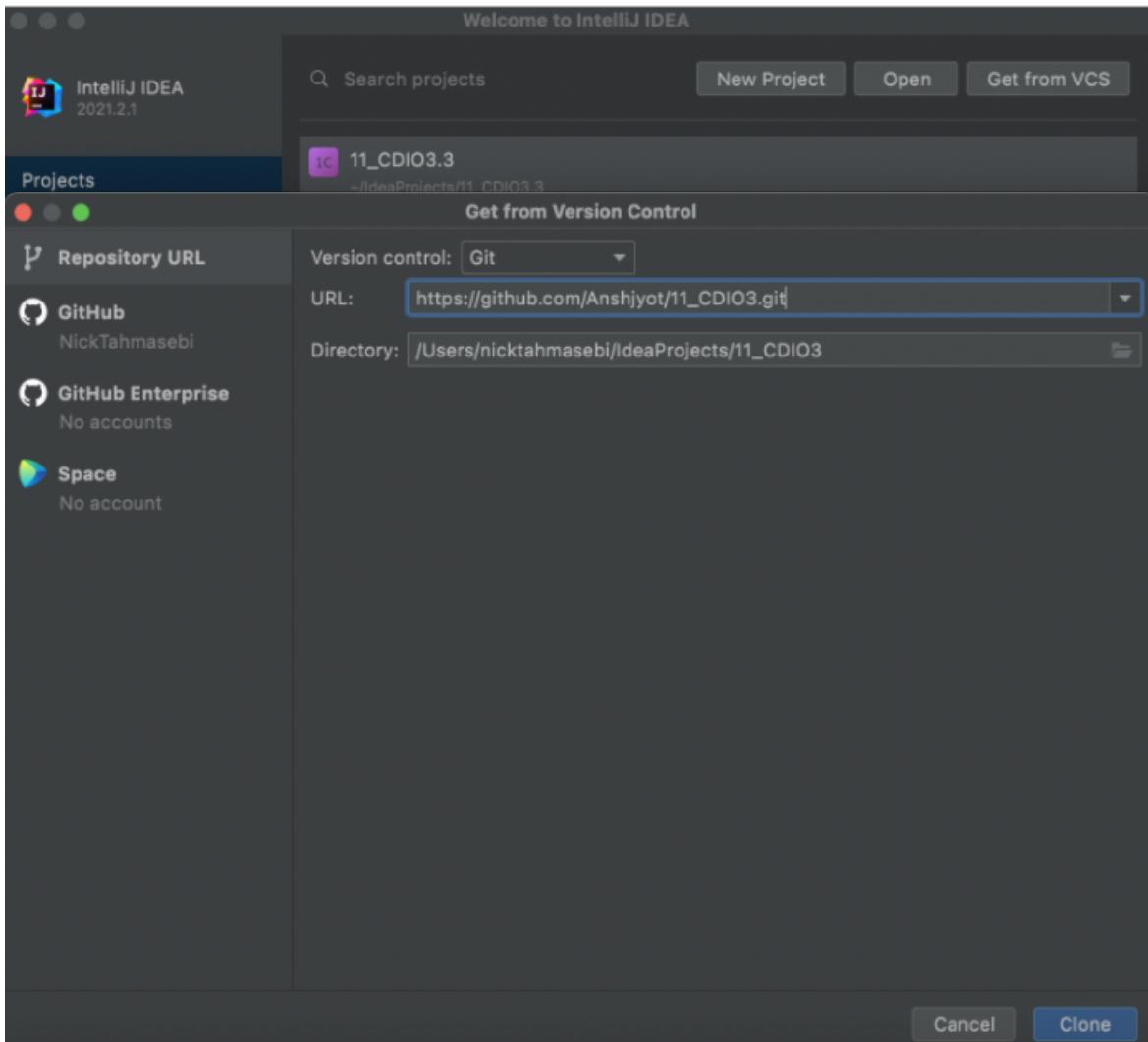
Dette er linket til vores github: https://github.com/Anshjyot/11_CDIO3.git Her kommer der to metoder på hvordan man kan importere projektet.

Metode 1) URL fra GitHub 1.a) På GitHub findes der den rigtige repository, hvor man derefter trykker på ”Code” og kopierer den viste URL.

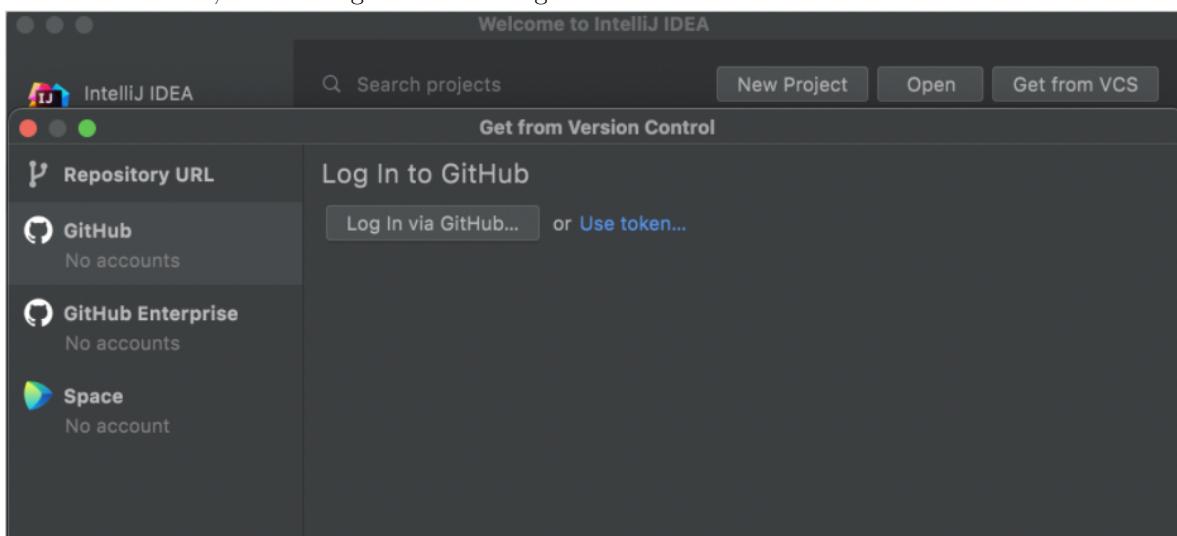


Figur 1: Viser vores Git-depository

1.b) Her åbnes IntelliJ, hvor man trykker på ”Get from VCS (Version Control)”, og derefter indsætter det kopierede link fra 1.a). Man skal dobbelttjekke at der ikke allerede ligger en fil på sit Directory med samme navn. Derefter trykker man ”Clone”. Derefter er projektet importeret.



Metode 2) At forbinde GitHub til IntelliJ 2.a) Hvis vi trykker på “Get from VCS (Version Control)”, og derefter “GitHub”, får vi muligheden for at login.



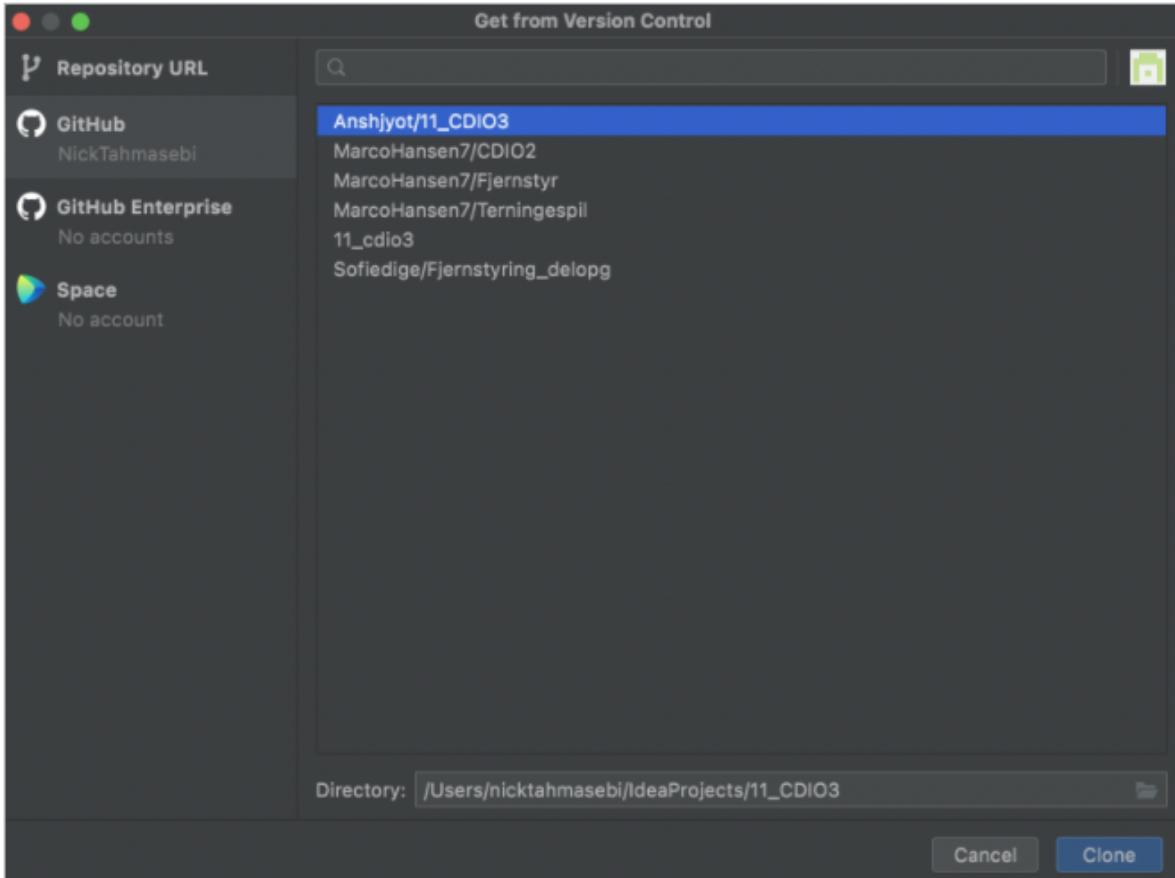
2.b) Efter login er trykket bliver man ført videre til en hjemmeside, hvor man skal “Authorize in Github”, Her logger man ind via sit GitHub login, og derefter får man besked på at man nu kan lukke siden.

Please continue only if this page is opened from a [JetBrains IDE](#).

[Authorize in GitHub](#)

You have been successfully authorized in GitHub. You can close the page.

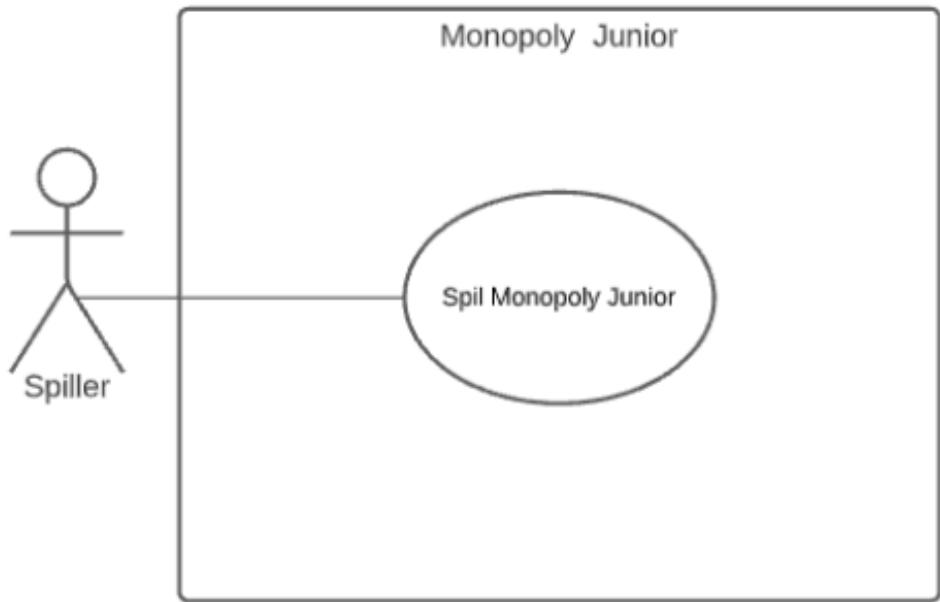
2.c) Når man vender tilbage på IntelliJ kan man nu se de følgende projekter der er delt med ens konto. Her vælger vi så “11_CDIO3”, og derfter åbnes projektet som ønskede.



Figur 2, 3, 4 5: Viser Guide til importering af vores projekt

5 Analyse

5.1 Use-case diagram



Figur 6: Viser vores Use-case diagram

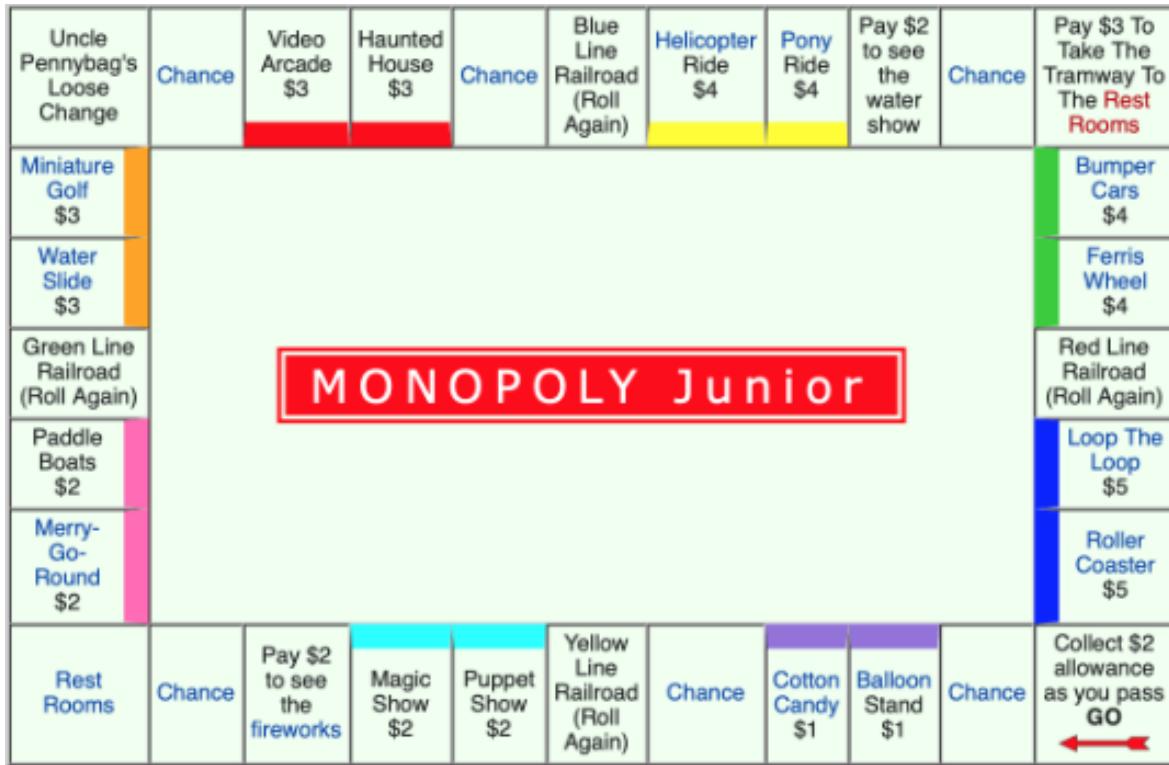
Use-case diagrammet består af en primær aktør, som er ‘Spilleren’ og et enkelt use-case “Spil Junior Monopoly”. Det er kort og kontant, da der kun er én use-case.

5.2 Kravspecifikationer

I denne del af rapporten beskrives vores krav yderligere, samt en risikoanalyse som tager udgangspunkt i de risici som kan forekomme under udviklingen af vores projekt.

Spillepladen vi vil gå ud fra er et Monopoly junior board fra 1990. De eneste ændringer vi har lavet på brættet er at fletet “Pay 2 dollars to see the water show” og “Pay 2 dollars to see the fireworks” er slettet og der er tilføjet et “Jail” og “Visit Jail” felt. Ellers følger den distribueringen som nedenfor.

Linket til spillebrættet: https://en.wikipedia.org/wiki/Monopoly_Junior



Figur 7: Viser det spillebræt vi går ud fra i vores GUI

Nedenfor ses alle de 32 felter. Felter:

- Felt 1: "Start"
- Felt 2: Chancen bullet.
- Felt 3: Balloon stand
- Felt 4: Cotton Candy
- Felt 5: Chance
- Felt 6: Yellow Line Railroad
- Felt 7: Puppet show
- Felt 8: Magic show
- Felt 9: Jail visit
- Felt 10: Chance
- Felt 11: Rest rooms
- Felt 12: Merry-Go-Round
- Felt 13: Paddle Boats
- Felt 14: Green Line Railroad
- Felt 15: Water Slide
- Felt 16: Miniature Golf
- Felt 17: Uncle Pennybags Loose Change
- Felt 18: Chance

- Felt 19: Video Arcade
- Felt 20: Haunted House
- Felt 21: Chance
- Felt 22: Blue Line Railroad
- Felt 23: Helicopter Ride
- Felt 24: Pony Ride
- Felt 25: Jail
- Felt 26: Chance
- Felt 27: Go to Restrooms
- Felt 28: Bumper Cars
- Felt 29: Ferris Wheel
- Felt 30: Red Line Railroad
- Felt 31: Loop the Loop
- Felt 32: Rollercoaster

Der er sådan set 4 forskellige typer af felter. Den første er de typiske købe-felter, som er defineret ved GUIStreet. Et felt hvor man har mulighed for at købe, samt leje/rent feltet når det er købt af en anden spiller.

Feltet VisitJailSquare gør sådan set ikke noget, dens funktionalitet er bare at være tilstede - da det er bare et besøgssted. Dog bliver man i JailSquare sendt en del felter tilbage, tilbage til VisitJailSquare som en konsekvens. Dog ville vi ønske at man mistede sin tur, når man røg i fængsels - så sammenligningen med et rigtigt Jail-felt er tilpas. Dette formået vi ikke pga. tidspres.

Vores sidste type felt er Chance-feltet, som desuden var ret essentielt at få implementeret - da det fylder en stor del af spilformatet. Vi har implementeret 4 forskellige typer af Chance-kort som også blev fremvist tidligere i rapporten.

Prioriteringsliste

Vi har disse 3 krav som er nødvendige for at spillet kan spilles:

1. A Basic Turn - Spillet bliver spillet på tur
2. Winning the game - Når en spiller ikke har flere penge, vinder spilleren med flest penge.
3. Setting up the board - Sætte brættet op, vælge - en spiller som bank, - bilfarve, - pengebeløb fra start, starte på feltet "Go".

Disse 3 er en del af MVP, altså "minimum viable product", for at spillet overhovedet kan fungere.

Andre vigtige regler, som vi har prioritert er listet nedenfor.

1. **Amusement** - Når man skal betale modstanderen for at lande på deres ejede felt.
2. **GO!** - Når du lander eller går forbi start, modtager du \$2 dollars fra banken. Denne feature er prioritert højt, da det er sådan man tjener penge.
3. **Railroad** - Du får en ekstra tur, som fungerer som om det var et enkelt terningeslag.
4. **Chancekort** - Chance Kortet er den regel, vi prioritere næst lavest, da spillet sagtens ville kunne fungere uden, samt er det mange diverse kort der skal laves og dette er meget tidskrævende. Men vi har valgt at prioritere den næst lavest, da vi kan implementere nogle få chancekort, til at gøre spillet mere sjov at spille.
5. **Loose change** - Jackpot. Denne prioritet er sat lavest, da den kræver hele 3 regler for at kunne lade sig gøre.
 - 5.1 **Pay \$2 Dollars spot** - koster \$2 for at se fyrværkeri eller et vandshow som betales til loose change
 - 5.2 **Restrooms** - Når man lander på feltet "Go to the restroom", betaler man \$3 til "loose change". Denne er nedprioriteret, da feltet ikke fungerer uden Loose change reglen.

Funktionelle krav

ID	Beskrivelse
k1	Et spil mellem 2-4 personer
k2	Slå med 1 terning mellem tallene 1-6. (Terningen virker korrekt) E.I (Bliver beskrevet i extensions i næste tabel)
k3	Terningerne skal hver have 6 sider, altså to "standard" terninger.
k4	Spillerne starter med \$35
k5	Når en spiller ikke har flere penge, vinder spilleren med flest penge. EV - Edge case, hvis alle spillerne har lige mange penge. (Beskrevet længere nede)
k6	Der skal være felter med numrene 1-32. Disse har en positiv, neutral eller negativ effekt på spillerens balance i banken.
k7	Spilleren skal kunne gå i ring på brættet. Spilleren skal altså kunne lande på et felt og fortsætte derfra.

Figur 8: Viser vores Funktionelle krav sorteret

Ikke-funktionelle krav

ID	Beskrivelse
k8	Spillerne skal kunne spille spillet uden brugsanvisning. (Almindelige mennesker) E.II (Bliver beskrevet i extensions i næste tabel)
k9	Systemet skal kunne køre på Windows maskinerne i databarene på DTU.
k10	Det skal gøres muligt for kunden at inspicere koden, og der skal derfor linkes til vores github repo.
k11	Der skal udføres test (JUnit), da det er et krav til reliability.
k12	En afprøvningskode, der sandsynliggør at balancen aldrig kan blive negativ, uanset hvad hæv og indsæt metoderne bliver kaldt med
k13	Spillet skal nemt kunne oversættes til andre sprog
k14	Det skal være let at skifte til andre terninger

Figur 9: Viser vores Ikke-Funktionelle krav sorteret Alle disse krav k1-k13 ovenfor er need-to-

have og er nødvendige i monopoly junior spillet, nice-to have kravene er beskrevet i tabellen nedenfor. Disse vil vi prøve at få med, hvis tiden rækker til det. Der er tilføjet extensions til vores krav, for at bevise at kravene er målbare.

Extensions til vores krav

E.I.

Vedrørende krav **2**, hvor terningen skal virke korrekt, så menes der, at terningen skal kunne generere et tilfældigt slag. Samt følge den teoretiske distribuering for 1 terning, som er en ligelig fordeling mellem alle terninge værdierne 1-6.

E.II.

Med "Almindelig mennesker" i krav **8**, menes der mennesker som ikke adskiller sig fra gennemsnittet, altså f.eks. uden påfaldende egenskaber, skal kunne spille spillet uden brugsanvisning.

E.III

I krav **13**, menes der at koden kan skrives på flere sprog i language klassen, så andre ikke behøver at ændre koden, men kun den tekst der bliver printet ud. For at gøre kravet målbart, kunne man fx. sige at programmet skal kunne oversættes inden for et vist tidsinterval.

E.IV.

I krav **14** menes der at man skulle ændre terningen fra en 6-sidet terning, til en hvilket som helst x-sidet trekant.

E.V.

I krav **5**, er der tale om en edge case. Det vil sige at der i spillet sker det at, når en spiller ikke har flere penge, vinder spilleren med flest penge. Det kan skabe et problem, hvis resten af spillerne har lige mange penge. Det problem kan man løse på forskellige måder. F.eks. ved at sige at de spillere med flest penge, skal slå med terningen om hvem der vinder. Vi har valgt at gøre det ved at sige at når alle andre spillere uddover én er bankerot, vinder den sidste spiller der er tilbage.

Figur 10: Viser extentions til de forskellige krav

ID	Nice-to-have
k14	Lave chancekort (prøv lykken)
k15	Du får en ekstra tur, når man lander på "Railroad"
k16	Når du lander eller går forbi start modtager du \$2 fra banken
k17	Får Jackpot, som er de penge der er samlet ind for nedenstående hændelser. <ul style="list-style-type: none"> - Skal betale \$2 for at se fyrværkeri - Skal betale \$3 for at gå på toilettet
k18	Når man skal betale de andre spillere for at lande på deres ejede felt.

Figur 11: Viser nice-to-have krav

Vi har tænkt os at tage udgangspunkt i disse fire chancekort.



Figur 12: Viser de chancekort vi vil implementere

5.3 Riskmanagement

Risici i sammenhæng med udviklingen af JuniorMatador:
<p>1. Udkiftning af medarbejdere</p> <p>1.1. Med medarbejdere menes der os som gruppe, der udvikler et program for en virksomhed. Vi kan risikere, at en af os bliver syge og kan misse udviklingen af projektet i et par dage. Dog er risikoen for at der er en reel udkiftning af gruppen meget lille, og vi har derfor valgt en sandsynlighed på 1.</p> <p>1.2. Men hvis det skulle ske, at der i virkeligheden var en udkiftning af en af os i gruppen, ville det påvirke helheden af projektet en del, da vi alle har betydningsfulde roller, samt antallet af vores gruppe: 5, som gør at der ikke er mange medarbejdere at tage af. Vi har derfor valgt en påvirkning på 3.</p> <p>1.3. Denne risiko har været svær at vurdere, da et gruppemedlem faktisk har forladt studiegruppen, samt at flere af vores medlemmer har været ramt af corona. Vi har dog stadig valgt en påvirkning på 3, da de resterende 4 medlemmer har arbejdet meget konsekvent.</p> <p>1.4. Denne risiko kan dog nemt forebygges, ved f.eks. at have en specialist på området og sørge for at flere medarbejdere kan varetage samme opgave. Derved bliver sandsynligheden også mindre.</p>
<p>2. Systemet underpræsterer</p> <p>2.1. Da vi alle i gruppen bruger MacBooks, samt er vant til det, kan vi have problemer i forhold til at skulle kode programmet på Mac. Kravet er nemlig at det skal fungere på en Windows maskine. Vi har derfor valgt en sandsynlighed på 3.</p> <p>2.2. Vi mener at påvirkningen er meget høj, da vi ikke kan levere et projekt til kunden, der ikke virker eller ikke lever op til kravspecifikationen om at det skal køre på Windows maskinerne i <u>databarerne</u>. Af den grund ligger påvirkningen på 5.</p> <p>2.3. For at forebygge dette kunne man udføre en masse test løbende og kontrollere at programmet virker.</p>
<p>3. Dårligt omdømme</p> <p>3.1. Virksomheder med et stærkt ry præstere generelt bedre, samt tiltrækker kvalificerede medarbejdere og øger deres samlede succes. Dog vil der nærmest altid være dårligt omdømme, eksempelvis ift. brugere der ikke</p>

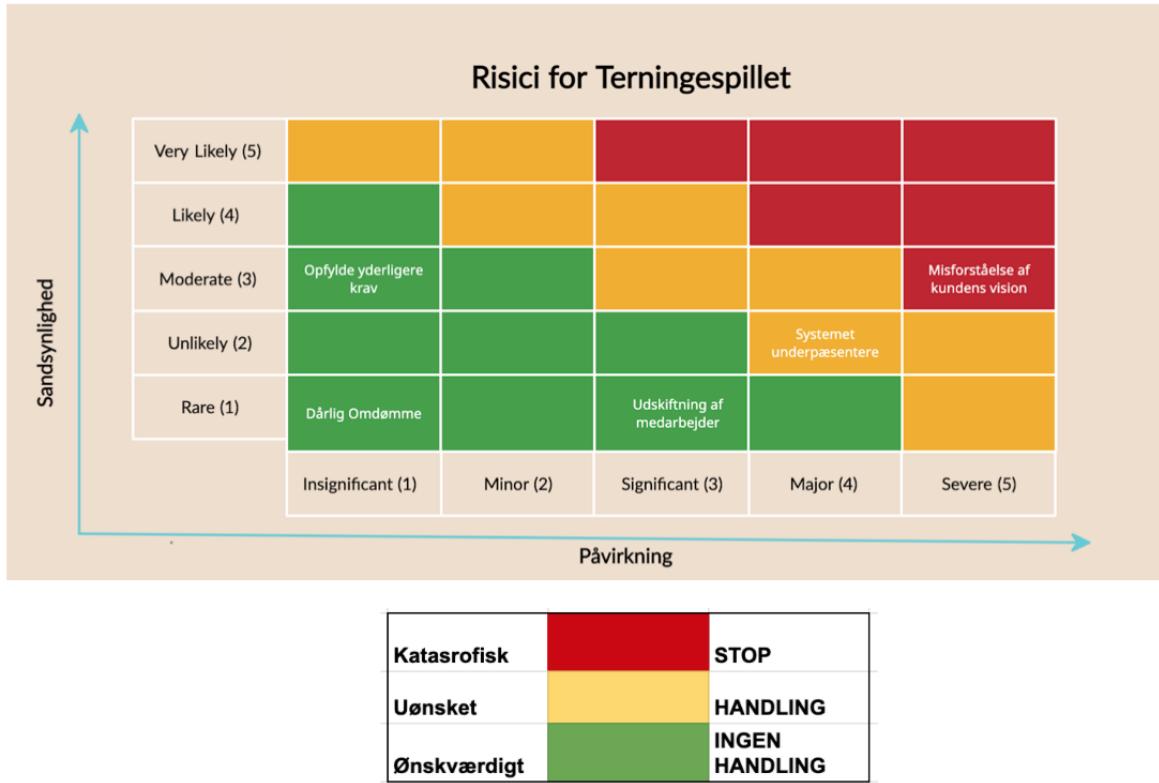
	<p>har særlig god værdsættelse for produktet. Men hvis der tages udgangspunkt i vores projekt, så mener vi ikke at der er "stor" plads til dårligt omdømme og derfor har vi sat sandsynligheden til 1.</p> <p>3.2. Hvis der fokuseres på selve påvirkningen af det dårlige omdømme, så vil påvirkningen være rimelig mild. Generelt vil dårligt omdømme ikke medvirke til den største påvirkningen på en virksom, og det vil det specielt heller ikke i vores projekt hvor sandsynligheden for at der forekommer dårligt omdømme er lav i forvejen. Netop derfor har vi valgt at sætte påvirkningen til 1.</p>
4.	<p>Misforståelse af kundes vision</p> <p>4.1. Vi har opdaget at kundens vision ikke har været specifik nok indtil videre, derfor har vi sat sandsynligheden til 2.</p> <p>4.2. Denne risiko kan dog forebygges, ved f.eks. at snakke med kunden dagligt. Derved bliver sandsynligheden for at der fremkommer færre misforståelser også forhøjet.</p> <p>4.3. For et hvilket som helst softwareudviklingsprojekt vil det være katastrofalt at misforstå kundens vision og deres kravspecifikationer, og det samme gælder for vores. Hvis vi ikke forstår hvad vi skal levere til kunden, er det praktisk talt umuligt at få en glad kunde. Af de grunde har vi vurderet påvirkningen til at være 5.</p>
5.	<p>Opfyldede yderligere krav</p> <p>5.1. Vi kan komme ud for at vi ikke har nok ressourcer og/eller tid til at udvikle ekstra funktioner for at tilfredsstille kunden udover kravene. Sandsynligheden for dette er middel, da det er vores 3 projekt, som kræver længere tid end sidst. Vi har derfor valgt en sandsynlighed på 3.</p> <p>5.2. Påvirkningen er derimod meget lav, da det ikke har en betydning for kunden om vi leverer ekstra-opgaver eller ej. Kunden bliver tilfreds med de grundlæggende krav, og vi har derfor sat påvirkningen til 1.</p> <p>5.3. For at forebygge dette, kan det være nødvendigt at være bedre til at prioritere vores tid, så man dermed har tid til at udvikle ekstra features til programmet. Dette sænker sandsynligheden for denne risiko og det kan give en mere tilfreds kunde.</p>

Figur 13: Viser vores Riskmanagement

5.4 P*I Matrix

Vi har tidligere beskrevet de adskillige risici der kan forekomme i vores projekt, og hvor stor sandsynligheden er for at den forekommer samt beskrevet om hvorvidt påvirkningen af risiciene vil have en stor indflydelse på vores projekt - ved at rangere dem fra 1-5. Herunder ses resultaterne af vores risici ift. hvordan de opstilles i vores P*I Matrix. Vi kan altså visuelt gennemskue hvilken slags handling vi

skal tage, hvis vi er ude for problemer.



Figur 14: Viser vores P*I Matrix

6 Dokumentation

6.1 Arv

Der er tale om Arv og nedarvning, når vi snakker om subklasser og superklasser. En superklasse har mindst en subklasse, dog kan en subklasse have én superklasse. Arv og nedarvning fortæller os, at klasserne deler deres mange metoder og attributter. I vores Monopoly spil, har alle vores felter noget til fælles. Derfor giver det bedst mening at bruge arv og nedarvning i dette tilfælde. Vi har en superklasse "Square", som nedarver til "ChanceSquare" (som indeholder vores 4 forskellige Chance kort), "JailSquare" (som sætter personen i fængsel) and "OwnedProperty" (som sørger for at købe/leje diverse felter).

6.2 landOnField og Polymorphism

Når man rykker rundt på boardet og lander på forskellige felter, der hver især udfører sit eget arbejde, og gør noget forskelligt, er der tal om konceptet Polymorphism. Her kunne der laves en "landOnField" metode, som gør brug af dette. Vi har valgt at bruge en Switch-Case til vores chance kort, her kunne vi f.eks. godt have brugt en "landOnField" metode, dog tænkt vi det ville være mere overskuelig med en switch case, til 4 forskellige Chance kort.

6.3 Abstract

Når der er tale om abstrakt, er det essentielt tale om man formindsker alle unødvendige data omkring et objekt, for at reducere kompleksiteten og øge effektiviteten i koden. En abstrakt klasse kan heller ikke instantieres, og hvis en klasse indeholder en metode som er abstrakt, (dvs. uden implementering), er klassen pr definition abstrakt. Når der er tale om diagrammer kan man nemt udpege en abstrakt klasse ved at se, om fonten er kursive, eller om det blot står `abstract`.

7 Konfigurationsstyring

7.1 Maven

Vi har brugt GUI'en der er blevet uddelt til os, og dermed bruges Maven til at køre GUI'en som en dependency. Vi har også en dependency til vores JUnit, som vi bruger til at teste vores kode igennem. Igennem vores JUnit vil vi også køre med code-coverage som beskrevet ovenfor. Ved brugen af Maven, så har vi alle udviklere mulighed for at downloade et projekt, hvor der ikke behøves at tilføjes en specielt version af nogle External-files, såsom i vores tilfælde matador-gui-filen eller JUNIT testen - da den medfølger gennem Maven. Dette problem blev vi ramt af i sidste projekt, men sørge denne gang for at vi konstrueret et Maven-projekt - hvor vores anvendte filer blev tilføjet gennem Maven.

Projekt kan meget nemt tilgås gennem github, hvor man kan "Download Zip" ud fra vores projekt, og dermed importere projektet i IntelliJ, da man har projektets folder. Da projektet køres gennem github, så er det yderst vigtigt at der er styr på brugen af merge, commit og push. Disse tre funktioner sørger for at tidlige versioner af vores opdateret projekt er gemt under git. Dette medfølger også, at der tit kan være merge-conflict, hvor to udviklere har redigeret under den samme klasse og dermed committed. Derfor er ansvarsområder under de forskellige klasser også vigtigt, så man ikke ender ud i disse typer af problemer.

8 Design

Nedenfor ses et fully dressed use-case over vores main flow i *Junior Monopoly*. Der ses altså en interaktion mellem aktøren og systemet. Det vil sige at når spilleren gør noget, retunerer systemet en handling.

Fully dressed use-case

Main flow:

1. Spiller trykker på “Start”
2. Systemet starter et nyt spil
3. Spiller 1-4 indtaster sit navn og vælger farve på bil
4. Systemet viser “Spiller” + “navn”
5. Spilleren trykker på “enter” for at slå med bægeret

5.1 Spilleren slår to tilfældige tal med terningerne.

5.2 Systemet viser resultatet af terningekastet

5.3 Systemet rykker spilleren summen af terningerne frem på brættet.

5.4 Spilleren lander på et felt.

5.4.a Spilleren lander på et amusement felt. Hvis ingen ejer feltet, skal spilleren købe det.

5.4.a.1 Spillet kræver et beløb for det givne felt

5.4.a.2 Spilleren betaler pengebeløbet til banken

5.4.a.3 Spillerens konto opdateres

5.4.a.4 Spillet giver spilleren ejerskab over feltet (spillerens farve kommer rundt om feltet)

Scenariet fortsætter fra punkt 6.

5.4.b Spilleren lander på et amusement felt, som er ejet af en anden spiller.

5.4.b.1 Systemet kræver et beløb for det givne felt

5.4.b.2 Spilleren betaler “x” antal kr til feltets ejer.

5.4.b.3 Spillernes kontoejendom opdateres.

5.4.c Hvis spilleren trækker et chancekort

5.4.3.1. Spilleren trækker en random case 1-4.

C1: Spilleren mister \$2, fordi spilleren har spist for meget slik.

C.II: Spilleren modtager \$2, fordi spilleren har lavet sine lektier

C.III: Spilleren rykker frem til start og modtager \$2

C.IV: Spilleren modtager \$1 fra alle spillere, fordi spilleren har fødselsdag.

Scenariet fortsætter fra punkt 6.

5.4.d Spilleren lander på feltet “Gå i fængsel”

5.4.d.I. Spilleren bliver rykket til feltet “Jail”

5.4.e Spilleren lander på feltet “Gratis parkering”

5.4.f Spilleren lander på feltet “Visit Jail”

5.4.f.I. Spilleren er bare på besøg i fængslet

5.4.g Spilleren lander på feltet “Start”

5.4.g.I Spilleren modtager \$2 for at passere eller lande på start.

6. Systemet afslutter spillerens tur.
7. Systemet beder den næste spiller om at slå med bægeret

Scenariet fortsætter fra punkt 5, indtil en spiller ikke har flere penge på deres konto.

8. Alle spillere undtagen én har ikke flere penge.

8.1 Systemet viser spilleren med flest penge: “Spiller “x” har vundet!”

8.1.1 Spiller vælger “Spil igen”

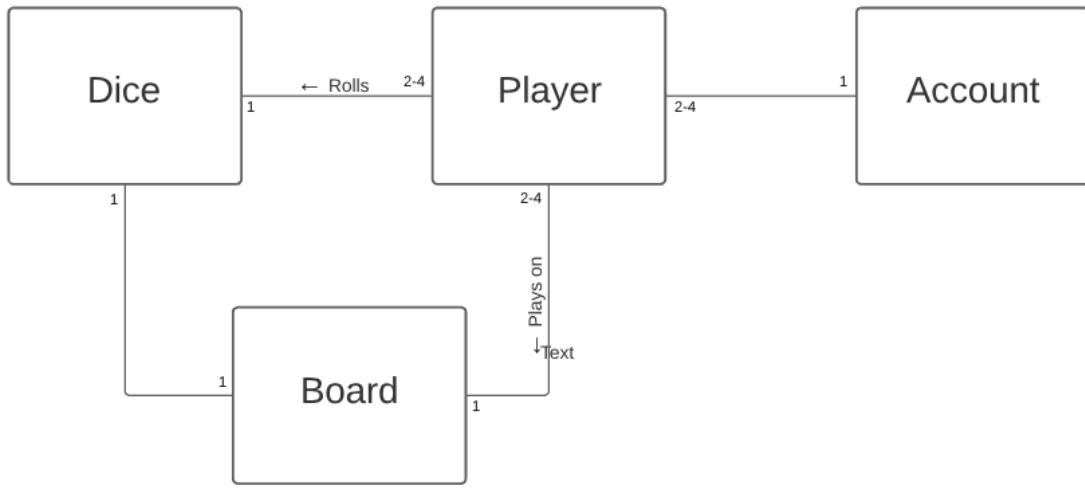
Scenarie fortsætter fra punkt 2

8.1.2 Spiller vælger “Afslut spil”

8.1.1.1 Systemet afslutter spillet.

8.1 Domænemodel

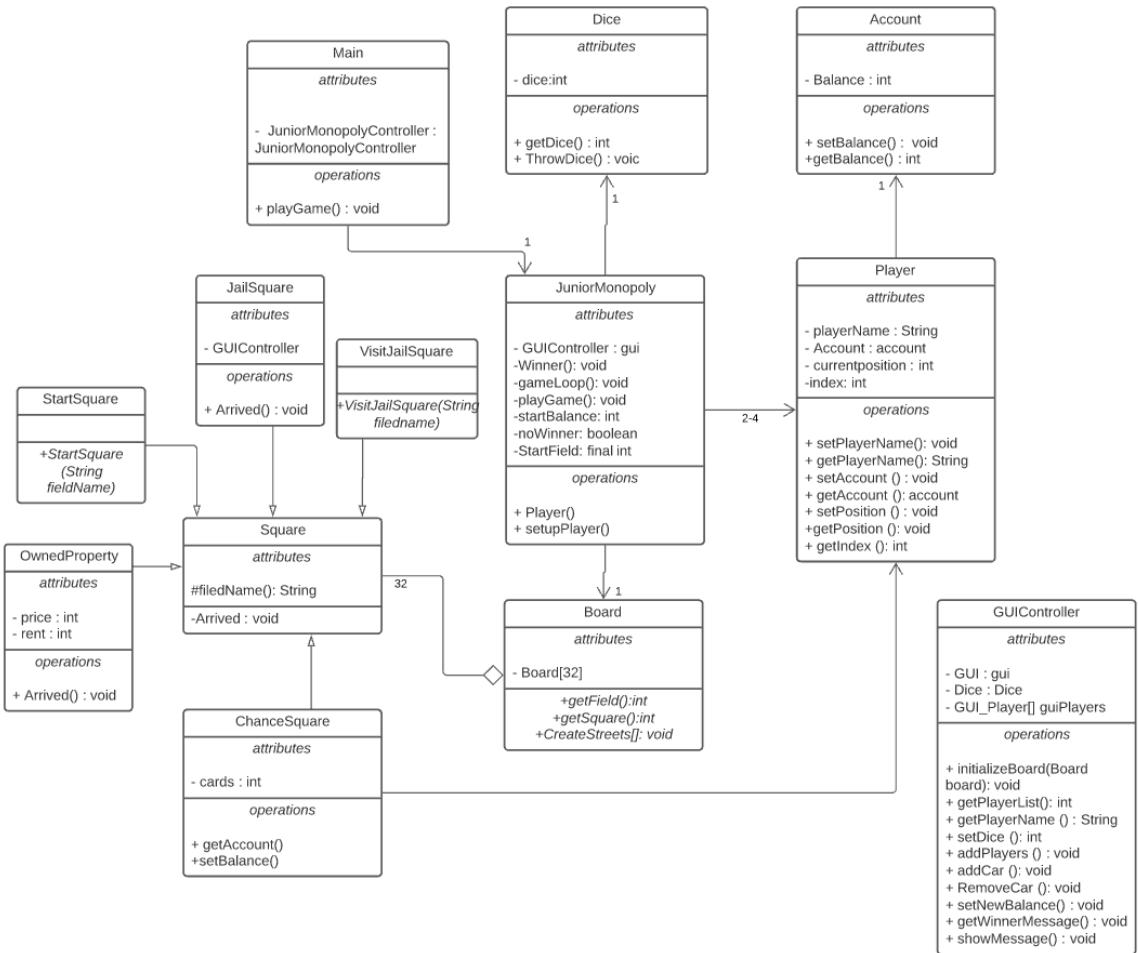
Vores domæne model består af 4 klasser, "Player", "Dice", "Account" og "Board". Spilleren ruller med en terning, og spiller på brættet (board). Spilleren er tilknyttet en account, da den holder spilleren pengekonto i spillet. Vores domæne model ser ud som følgende:



Figur 15: Viser vores Domænemodel

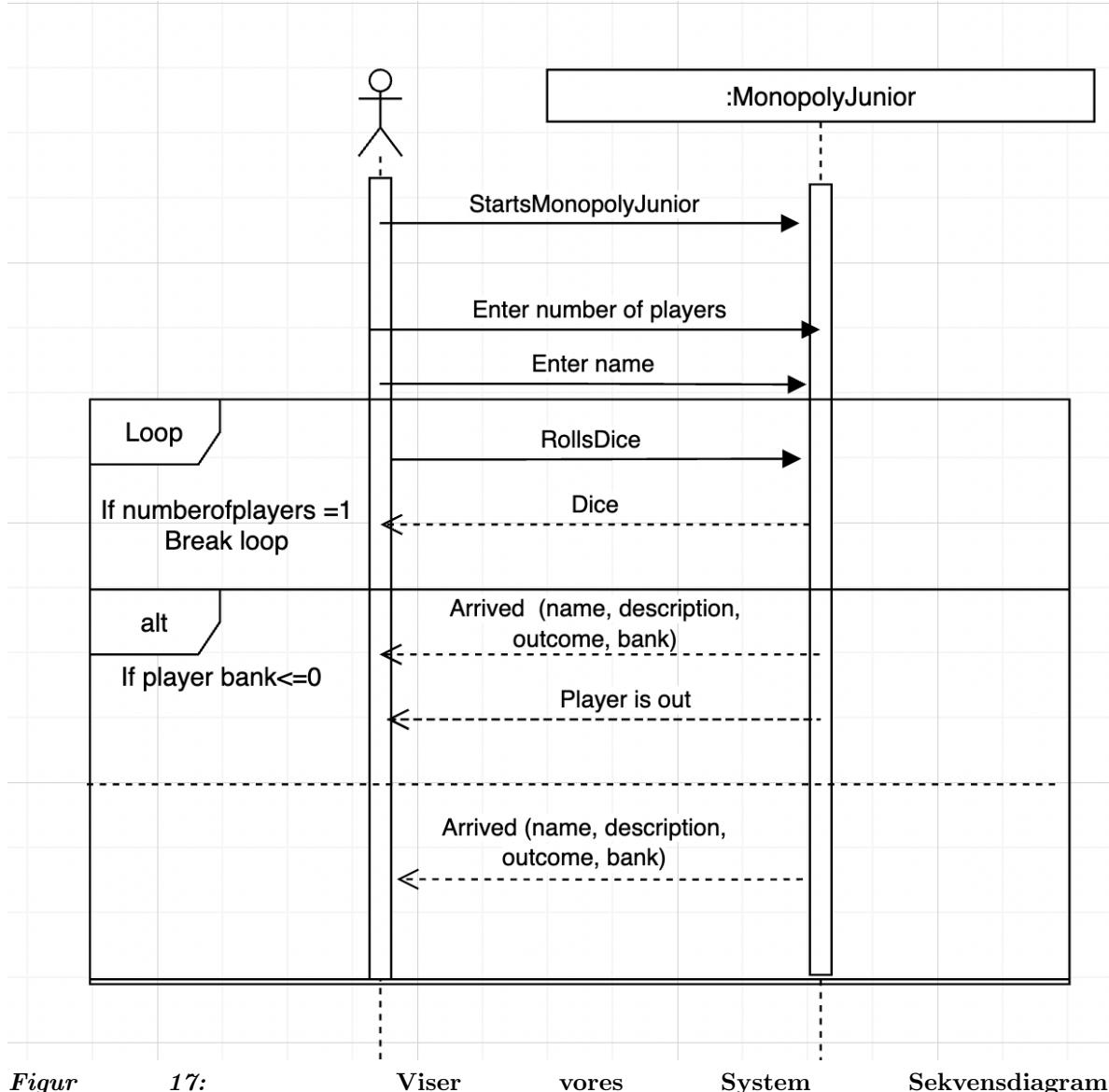
8.2 Design klassediagram

Vores design klassediagram er bygget op omkring vores JuniorMonopoly, som er tilknyttet board, dice, main og player . Det kan ses at Player-klassen er 2-4 spillere, der er koblet til banken, som har en bank til hver spiller. Klassen Board består af 32 felter, samt hvert felts beskrivelse, som er tilknyttet Square. Vi har 5 klasser som arver fra Square-klassen, disse 5 felter ses også på diagrammet. Disse er oprettet, da hver specielle felter har specifikke egenskaber.



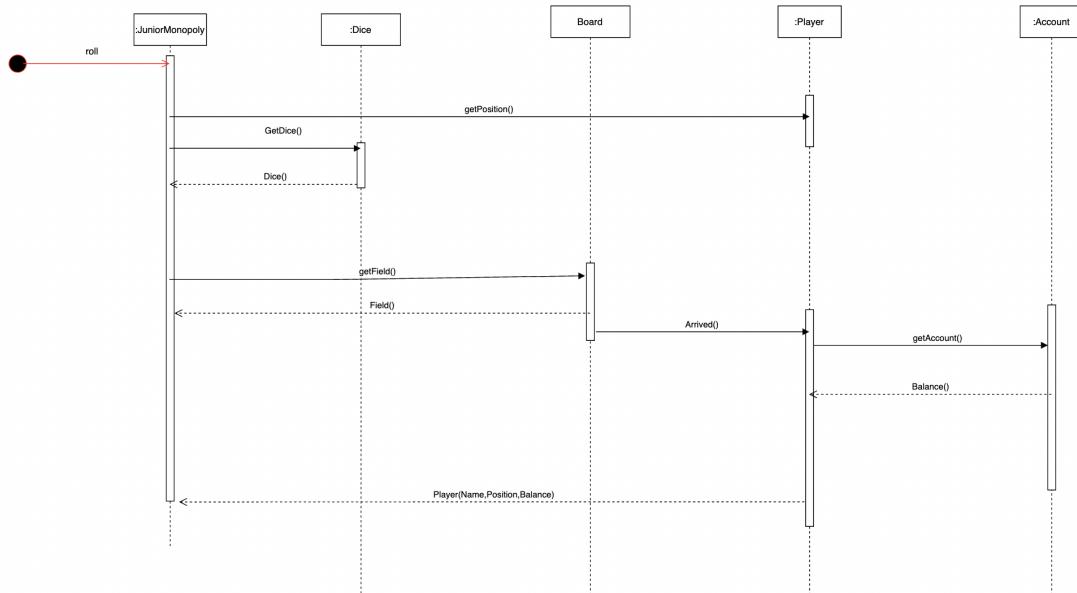
Figur 16: Viser vores Design klassediagram

8.3 System Sekvensdiagram



Her kigger vi på hvordan aktøren interagerer med systemet, systemet er simplificeret og centrale operationer er illustreret. Aktør starter spillet, indtaster antallet af spillere og navne. Derefter ruller aktøren terningen og modtager terningens øjne, samtidigt med at aktør får besked om hvad de er landet på og hvad der sker på spillerens konto. Vi har lavet et loop som breaker og stopper spillet hvis kun en spiller er tilbage. Vi har lavet et if-else statement som viser hvornår at en spiller er dømt ude.

8.4 Sekvens diagram



Figur 18: Viser vores Sekvens diagram

Sekvensdiagrammet er simplificeret da det ellers ville blive for uoverskueligt, vi har valgt at fokusere på terningeslaget hvor vi kan se hvordan nogle af klasserne kommunikerer med hinanden. Først ses initialiseringen af processen. Først henter systemet spillerens nuværende position som er sat i player klassen. Derefter kalder spillet på dice klassen som returnerer et tilfældige terningeslag. Positionen som spilleren havde ligges til terningeslaget og der bliver kaldt på board klassen som der returnerer os det felt som der landes på. Board klassen sender videre til spillerklassen hvad effekten af det givne felt er, og spillerklassen kalder på account klassen som sætter det nye beløb som spilleren har. Spillerklassen returnerer så udkommet til junior matador controlleren.

9 GRASP-PATTERNS:

GRASP-PATTERNS: Creator er et GRASP-pattern, som fortæller hvilken klasse der skal være ansvarlig for at oprette en ny forekomst af en klasse. I vores tilfælde kunne vores **Square**-klasse blive anset som en “creator”, da andre klasser såsom `ChanceSquare`, `JailSquare`, `VisitJailSquare` og `StartSquare` extender `Square`-klasser og kan dermed ikke være tilstede, hvis dens superclass (`Square`) ikke også er tilstede. Et andet eksempelvis kunne være, når **Player**-klassen formår at oprette `Bank`-klassen. Der kan ikke oprettes en `Bank`-klasse uden en `Player`, og dermed er dette en creator for klassen. Derudover kan vores `JuniorMonopolyController`-klasse anses som en creator for alle andre klasser, da inputs/methods fra alle andre klasser bliver sat i forbindelse sammen gennem vores `JuniorMonopolyController`. Denne forbindelse skaber funktionaliteten i programmet, og dermed ikke til at undvære - da den er med til ”oprette“ alle klasser.

Information expert er et af de GRASP patterns som også indgår i projekt ift. `OwnedProperty` og `Board` klasserne. Et princip, der bruges til at bestemme, hvor ansvarsområder såsom metoder skal fordeles til. Vores `OwnedProperty`-klasse initialisere netop vores `Board`-klasse som beskriver felternes price og rent price og dermed er det naturligvis også information-expert - da den burde have ansvaret for den klasse. Vores **JuniorMonopolyController**-klasse bør også anses som værende information-expert - da der som beskrevet også bør anses som en creator for de andre klasser og holder dermed på informationer fra alle klasser - hvilket også betyder at det har ansvaret for at programmet kan kører.

Low-coupling kunne muligvis ses i nedenstående eksempel. To elementer er netop “coupled”, hvis et element har en aggregation/composition “association” med et andet element eller hvis et element im-

plementerer/udvider et andet element. I vores OwnedProperty-klasse defineres der adskillige variabler som senere sammensættes med Board-klassen. Variablerne udvider netop beskrivelserne til hver af disse properties.

```

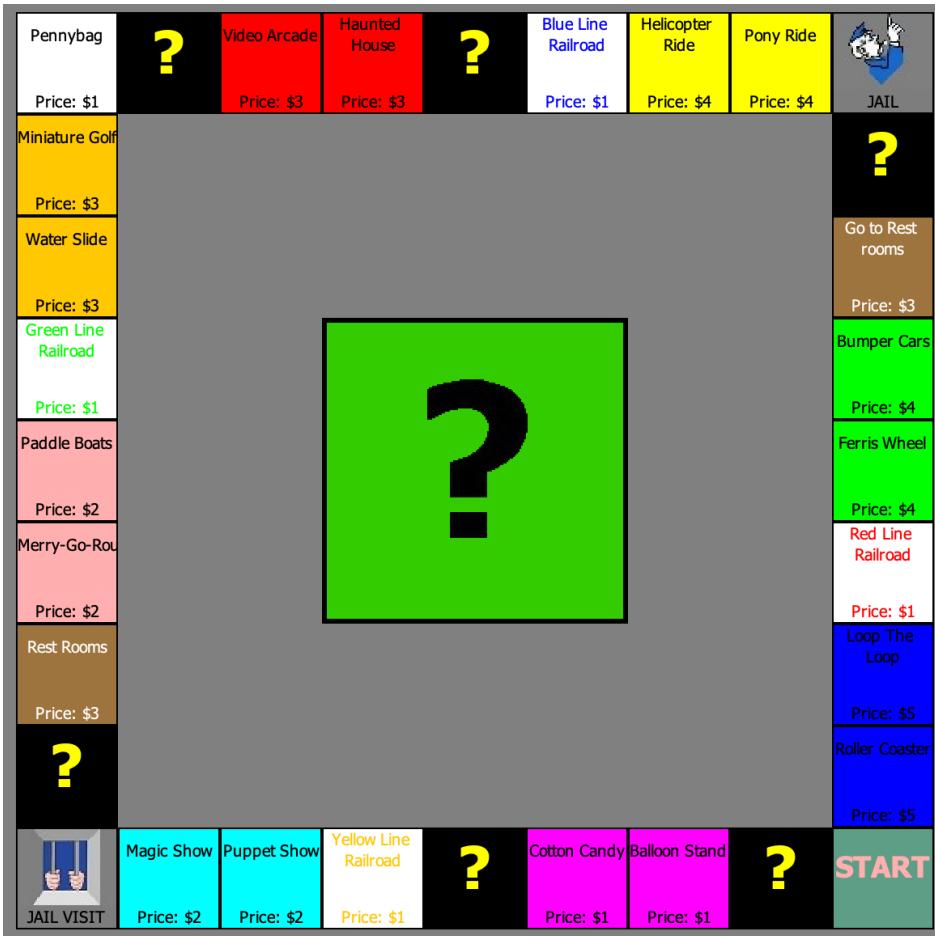
public OwnedProperty(String fieldname, int price, int rent, GUIController controller) {
    super(fieldname);
    this.price = price;
    this.rent = rent;
    this.controller = controller;
}

squares[26] = new OwnedProperty( fieldname: "Go to Rest rooms", price: 3, rent: 1,controller);
squares[27] = new OwnedProperty( fieldname: "Bumper Cars", price: 4, rent: 1,controller);
squares[28] = new OwnedProperty( fieldname: "Ferris Wheel", price: 4, rent: 1,controller);
squares[29] = new OwnedProperty( fieldname: "Red Line Railroad", price: 1, rent: 1,controller);
squares[30] = new OwnedProperty( fieldname: "Loop The Loop", price: 5, rent: 1,controller);
squares[31] = new OwnedProperty( fieldname: "Roller Coaster", price: 5, rent: 1,controller);

```

Figur 19: Viser dele af OwnedProperty og Board class

Polymorphism handler om, at objekter af forskellige klasser har operationer med den samme signatur/attribut, men forskellige implementeringer. Dette kunne sættes i forbindelse med vores felter. Felterne har samme signatur/attribut, men yder forskellige implementeringer til systemet ift. hvor meget pengesum der tilføjes eller trækkes fra ens bankkonto og generelt dens funktionalitet. Desuden kunne polymorphism også sammensættes med spillebrikkende, her er hver af spillebrikkende af forskellige type, forskellige farve og forskellige design. Desuden kunne polymorphism også sammenstås med vores chancekort, her har disse kort samme signatur/attribut - men har forskellige udfald.



Figur 20: Viser polymorphism ift. feltene

10 Implementering

Vi har valgt at tage udgangspunkt i udvalgte dele af vores projekt. Account og player: Account klassen er oprettet, så spillerne har en balance i spillet. Denne balance skal kunne ændres i løbet af spillet ved hjælp af getters og setters i "Player" klassen, hver gang en spiller lander på et nyt felt.

Square: Der bliver gjort brug af inheritance fra både "Chance Square", "Jail Square", "Visit Jail Square", "Start Square" til Square, da de alle extender til Square og arver alt fra Square-klassen. Det har vi gjort, da de alle har brug for metoden Arrived for at vide hvor i spillet de lander. I "Chance Square" har vi tilføjet de forskellige chancekort vi havde tid til at implementere.

Dice og Cup: Tidligere i programmeringen havde vi en cup-klasse, da den skulle indeholde to terninger, men i Junior Monopoly skal vi kun bruge en enkelt terning og har derfor valgt at udelade den denne gang, altså udelade Cup-klassen. Klassen dice står dermed for sig selv, som indeholder en sekssidet terning, der kaster et tilfældigt tal mellem 1-6.

JuniorMonopolyController og Main: Vores Main-klasse bliver brugt som en game-launcher til at køre spillet. Hvoraf vores JuniorMonopolyController er vores game-controller som indeholder nærmest alle metoder fra alle klasser og medvirker til at brugerne kan interagere med spillet. Metoder som er nødvendigt for at spillet kan køres.

GUIController: Som det også kan aflæses ud fra navnet, så er GUIControlleren selve klassen som skaber vores GUI og de tilhørende metoder. En GUI, der indeholder alle metoder som vi har konstrueret gennem de andre klasser - såsom et board, fields, players (spillebrik), terning osv. En betydningsfuld klasse i udviklingen af vores JuniorMonopoly-spil.

10.1 Kode:

Dice: Der er defineret en Dice-klasse som genererer et tilfældigt terningeslag i Dice klassen. Herunder gøres der brug af Math.random() der viser matematikken bag det tilfældigt genererede slag. Vi har fjernet vores Cup-klasse fra sidste opgave, da dette spil skulle indeholde 1 terning - derfor var slet ikke relevant at have den klasse implementeret.

```
package game;

public class Dice { // Creates a dice throw method for one dice only
    private int dice;

    public Dice() {
    }

    public void ThrowDice() { dice = (int) (Math.random() * 6) + 1; }

    public int getDice() { return dice; }
}
```

Figur 21: Viser vores Dice kode

Vi har valgt at implementere og designe vores **Board-klasse** ved at gøre brug af de eksisterende metoder i vores gui-jarfil. Der er i forvejen definere Chance-feltet, Jail-feltet og de generelle Street-felter som har sine parametre. Det frembringer et overskuelig billede af koden, at udvikle felterne gennem et array.

```

fields[24] = new GUI_Jail();
fields[24].setSubText("JAIL");
fields[24].setDescription("Locked up");

fields[25] = new GUI_Chance( title: "?", subText: "", description: "", Color.BLACK, Color.YELLOW);
fields[26] = new GUI_Street( title: "Go to Rest rooms", subText: "Price: $3", description: "You got the Tower",
fields[27] = new GUI_Street( title: "Bumper Cars", subText: "Price: $4", description: "", rent: "", Color.GREEN,

```

Figur 22: Viser vores Board kode

10.2 GUI-implementation

Vi har valgt at implementere en GUI i vores JuniorMonopolyController som skal fremvise et brætspil med specifikke felter. Vi har implementeret matadorgui-3.1.6.jar-filen som konstruerer den overordnede layout for vores brætspil.**GUIController:** Vores GUIController står for alt det visuelle, og kan dermed anses som værende et af vores allervigtigste klasser. En klasse der konstruerer vores dice, players og visualisere generelle metoder fra andre klasser såsom getPlayerList og getPlayerName.

```

public void addPlayers(Player[] players) { // Creates the player in the GUI
    players[0].GUICar = new GUI_Car(players[0].PlayerName, players[0].bank.amount, new GUI_Car(Color.RED,
    players[1].GUICar = new GUI_Car(players[1].PlayerName, players[1].bank.amount, new GUI_Car(Color.BLUE,
    board.addPlayer(players[0].GUICar);
    board.addPlayer(players[1].GUICar);
}

```

Figur 23: Viser vores getPlayerList og getPlayerName fra GUIController
Herefter gøres der desuden brug af setCar som er sat ind i forbindelse med felt-positionerne i en MoveCar metode, som ses herunder:

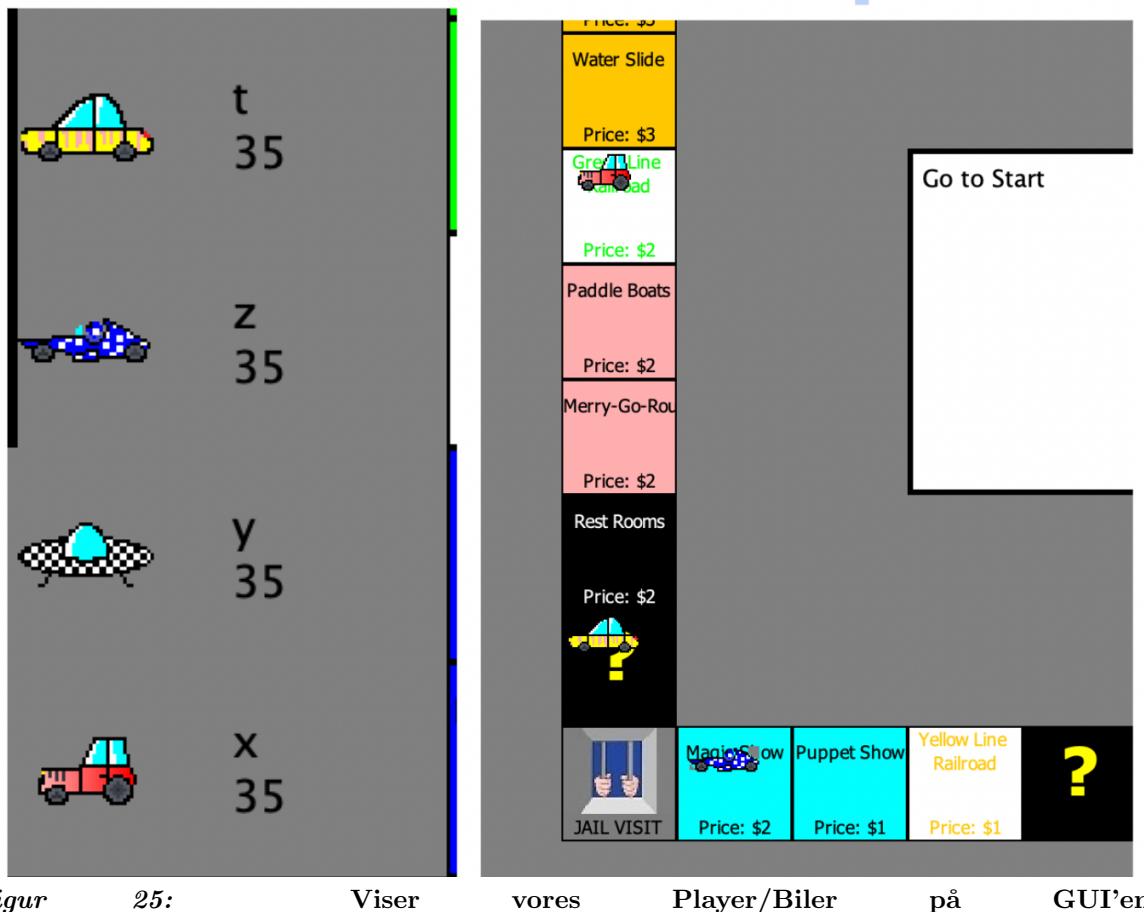
```

public int setDice() { // Creates the dice in the GUI
    gui.getUserButtonPressed( msg: "Throw Dice", ...buttons: "Throw");
    int x = (int)(Math.random()*7)+2;
    int y = (int)(Math.random()*7)+2;
    int rotation = (int)(Math.random()*360);
    gui.setDice(dice.getDice(),rotation,x,y,dice.getDice(),rotation, x,y); // Only one dice visible on the board
    dice.ThrowDice();
    return dice.getDice();
}

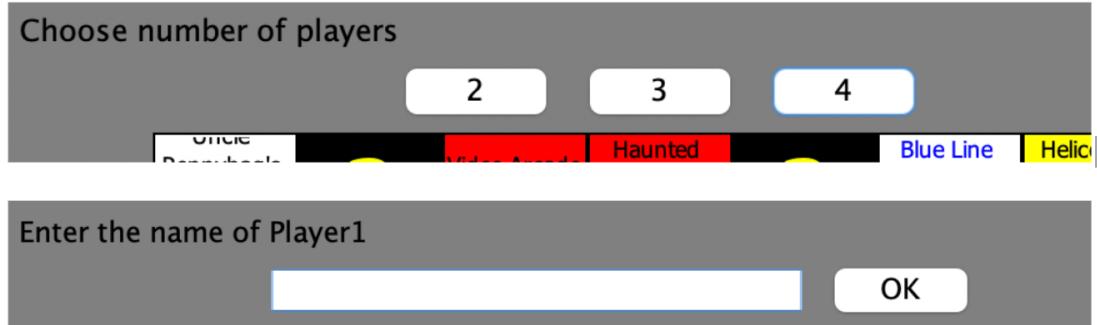
public void addPlayers(List<Player> players) { // Creates different types of game-pieces
    this.guiPlayers = new GUI_Player[players.size()];
    GUI_Car[] car_choices = {
        new GUI_Car(Color.PINK, Color.RED, GUI_Car.Type.TRACTOR, GUI_Car.Pattern.HORIZONTAL_GRADIENT),
        new GUI_Car(Color.BLACK, Color.WHITE, GUI_Car.Type.UFO, GUI_Car.Pattern.CHECKERED),
        new GUI_Car(Color.BLUE, Color.WHITE, GUI_Car.Type.RACECAR, GUI_Car.Pattern.DOTTED),
        new GUI_Car(Color.YELLOW, Color.PINK, GUI_Car.Type.CAR, GUI_Car.Pattern.ZEBRA)
    };
    for (int i = 0; i < players.size(); i++) { // Array of players in the GUI
        this.guiPlayers[i] = new GUI_Player(players.get(i).getPlayerName(), players.get(i).getAccount().getBalance());
        AddCar( position: 0, i);
        gui.addPlayer(this.guiPlayers[i]);
    }
}

```

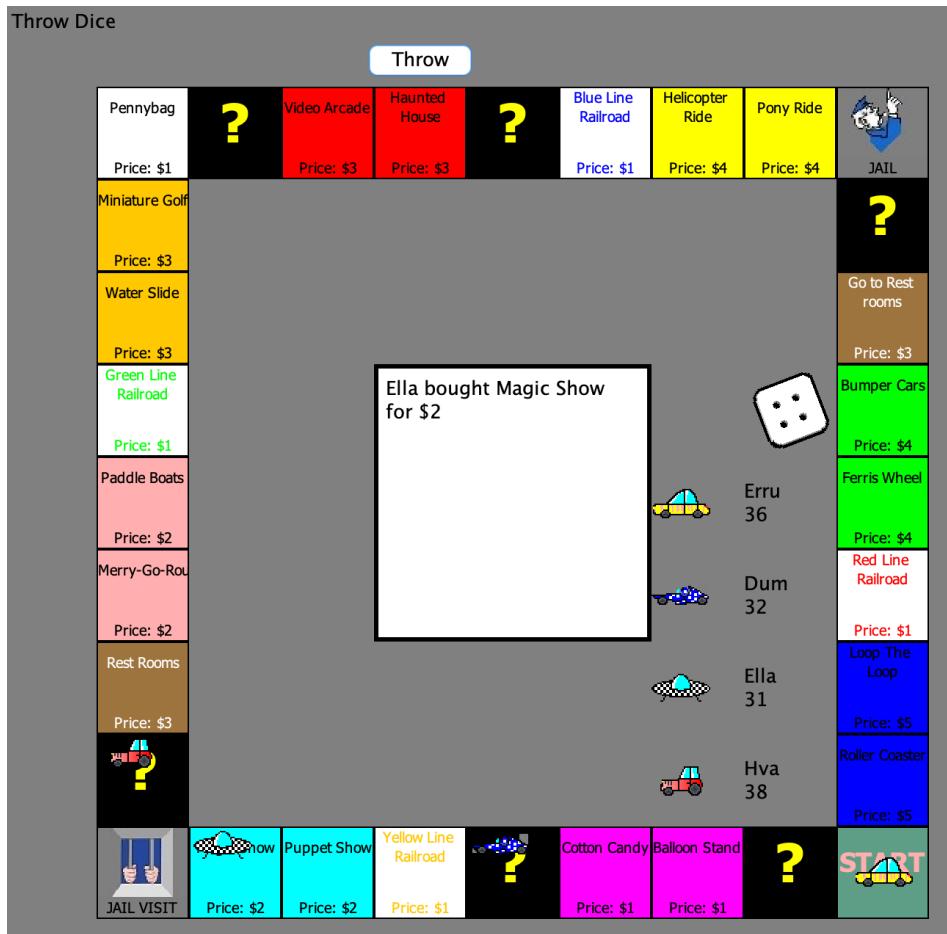
Figur 24: Viser vores setDice og addPlayers-GUI metode



Figur 25: Viser vores Player/Biler på GUI'en



Figur 26: Viser Choose NumberOfPlayers og EnterName metode



Figur 27: Viser et overordnet syn på vores GUI

11 Test

I vores 2 test, gøres der brug af JUnit med imports vedrørende assertEquals som bruges til at sammenligne vores "expected" og "actual" resultater.

Test T1: Der er blevet foretaget to forskellige JUnit test, den ene er en Account-Test og den anden er en PlayerPosition Test. Account-Test tester specifikt vores setBalance metode - hvor der tilføjes en amount (en penge sum) til ens konto - hvor der naturligvis gøres brug af vores setBalance-metode. Samt en metode der tester at ens amount ikke går under 0. Det ender med at min expectingResult og actualResult er ens, og dermed er testen korrekt.

```

@Test
public void testingSetBalance() {
    System.out.println("Set Balance");
    Account amount = new Account( balance: 0 );
    amount.setBalance(1000);
    int expectingResult = 1000;
    int actualResult = amount.getBalance();
    assertEquals(expectarResult, actualResult);
}

```

Figur 28: Viser setBalance-testen

Test T2: Ift. vores PlayerPosition Test, så gøres der brug af vores setPosition og getPosition metode fra vores program. En metode der positionere spilleren ved et specifikt felt. Det kan ses herunder.

```
public class PlayerPositionTest {

    Player player = new Player();
    @Test
    public void test() {
        int expectingResult = 32;
        player.setPosition(32);
        int actualResult = player.getPosition();
        assertEquals(expectingResult, actualResult);
        System.out.println("SetPosition virker korrekt");
    }
}
```

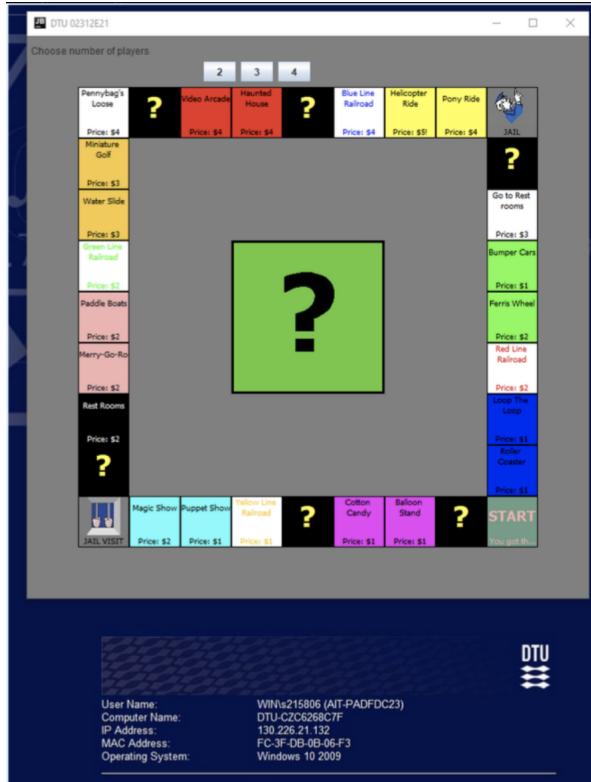
Figur 29: Viser PlayerPositionTest)

De to Junit test er desuden også sammensat med code-coverage test - hvoraf der ses hvor meget de fylder af vores game-package.

game 20% (2/10) 19% (6/31) 5% (10/1..)

Brugertest T3: Vi har lavet en brugertest på spillet. Testen er lavet for at se om spillet opfylder krav som ikke ellers er målbare, f.eks: "Spillet skal være godt" Vi fik en person uden programmeringserfaring til at prøve programmet og han blev bedt om at tænke højt. Uden hjælp kan bruger spille monopoly junior Bruger siger at det er nemt at starte spillet og knapperne er iøjenvældende så man ved hvad man skal gøre for at komme videre Bruger synes at det var svært at se hvad man landede på og det var først efter flere runder at han fandt ud af at man kan klikke på felterne og se prisen af hvert felt. Han synes at spillet kørte fint og at turene gik hurtigt men at det var lidt kedeligt og foretrak et fysisk brætspil så man lettere kunne følge med og kunne leve sig mere ind i spillet. Jeg spurgt om hvad der gjorde at det ikke var nemt at leve sig ind i, og han synes at der manglede en form for animation, at bilerne var svære at følge og se så man brugte tid på at finde ud af hvad der var sket og hvor man var rykket til.

Databar-test T4: Vi har yderligere importeret vores kode inde på databaren, på DTU. Det har tidligere været et krav, at vores program skulle kunne køre på Windows maskinerne i databarene. Dette krav medførte vi også i dette spil, og det kan herunder ses at spillet kan spilles i databaren.



Figur 30: Viser Databar-test

Vi har yderligere importeret vores kode inde på databaren, på DTU. Det har været et krav, at vores program skulle kunne køre på Windows maskinerne i databarene. Vores expected result, er at programmet ville kunne åbne GUI'en og køre, indtil en af spillerne har vundet. Det kan ses på figuren, at programmet opfylder vores expected result. Vi kan konkludere at programmet kører fejlfrit på databarene på DTU.

Traceability matrix

Traceability matrix:

Test case	Requirement number k1-k14													
	k1	k2	k3	k4	k5	k6	k7	k8	k9	k10	k11	k12	k13	k14
T1				x							x	x		
T2						x	x				x			
T3								x						
T4									x					

Figur 31: Viser Traceability matrix

Test cases

Nummer	Test	Forventede resultater	Resultater	Pass/Fail	Kommentarer
1	Køb og betal leje	Det forventes at spiller kan købe felter og skal betale leje af felter som spilleren ikke ejer	Spilleren betaler leje hvis en anden spiller har købt feltet, og ellers køber spilleren selv feltet	Pass	
2	Jail-field	Det forventes at man bliver rykket til jail-visit feltet og mister sin tur for næste runde	Når der landes på jail feltet bliver man rykket til jail visit men man mister ikke sin tur	Fail	Der mangler at du bliver sprunget over
3	Chancekort	Det forventes at der bliver genereret et tilfældigt chancekort ud af de muligheder som der er, og at chancekortet påvirker spilleren	Chancekortene fungerer og systemet vælger et tilfældigt.	Pass	

Figur 32: Viser Test-cases

12 Konklusion

Det kan nu konkluderes, at vi har opnået at udvikle et fungerende Junior monopoly der kan spilles af 2-4 spillere. Vi har valgt at fokusere på objekt-orienterede programmering og dermed består koden af tretten forskellige klasser der hver har en afgørende rolle for programmets kørsel. Vi har formået at opfylde de opstillede krav til projektet, såsom at spillet skal kunne bruges på Windows maskinerne i databarerne på DTU, at alle mennesker kan spille vores spil uden en brugsanvisning. Programmet er som nævnt i indledningen et springbræt til det næste projekt som bliver mere omfattende og vi føler at vi allerede er kommet langt og kan begynde at kigge på detaljer for at optimere spillet og brugerenes glæde ved at køre programmet.