02312
Indledende programmering

Final - Gruppe 42

Emil Nymark Trangeled
s195478

Stig Bødtker Petersen
s186333

Mohammed Zahed
s186517

Mads Emil Kaas Hansen
s195159

Magnus Thorup Müller Larsen
s194056

Afleveringsfrist 18. januar 2021

Link til github repository der blev brugt under projektet:
https://github.com/DTU1semHackerChamps/42_Final

Timeregnskab

Dato	Emil	Stig	Zahed	Mads	Magnus
04-01-2021	3 timer	3 timer	3 timer	3 timer	3 timer
05-01-2021	4,5 timer	4,5 timer	4,5 timer	2 timer	4,5 timer
06-01-2021	2,5 timer	2,5 timer	2,5 timer	2,5 timer	2,5 timer
07-01-2021	6 timer	6 timer	6 timer	5 timer	6 timer
08-01-2021	6 timer	6 timer	6 timer	0 timer	6 timer
09-01-2021	5 timer	0 timer	2,5 timer	0 timer	5 timer
10-01-2021	2 timer	0 timer	0 timer	0 timer	2 timer
11-01-2021	7,5 timer	7,5 timer	7,5 timer	7 timer	7,5 timer
12-01-2021	12 timer	8,5 timer	8,5 timer	8,5 timer	12 timer
13-01-2021	8 timer	6,5 timer	6,5 timer	6,5 timer	7,5 timer
14-01-2021	7 timer	7 timer	7 timer	6 timer	6,5 timer
15-01-2021	6 timer	7 timer	7 timer	7 timer	7 timer
16-01-2021	6,5 timer	6,5 timer	6,5 timer	6,5 timer	6,5 timer
17-01-2021	11 timer	12,5 timer	11 timer	11 timer	11 timer
18-01-2021	4 timer	4 timer	4 timer	4 timer	4 timer

Indholdsfortegnelse

Resume

I dette CDIO Final projekt har vi udarbejdet et matador spil, med henblik på at man skal kunne være 2-6 spillere. Spillet indeholder 40 felter, med 30 forskellige chancekort, som har deres egne effekter. Når man bevæger sig rundt på spillepladen kan man købe forskellige grunde. Hvis man får alle grunde af samme farve, kan man bygge huse og hoteller. Alle spillere starter med 30000 kr. Spillet handler om, at få andre til at gå fallit ved at købe en masse grunde og bygge huse og hoteller, så man kan opkræve en stor leje fra andre spillere. En spiller går fallit, når de skylder flere penge end de ejer. Når alle på nær en spiller er gået fallit, er spillet slut, og den sidste spiller vinder.

Indledning

Gruppen har fået til opgave at skabe et matador brætspil målrettet efter kundes vision. Spillet skal være simpelt, da det skal spilles på DTU's databar, og der skal desuden heller ikke være særligt mange instrukser for at spille spillet. Vi anvender DTU's egen GUI til at vise, hvad der sker, så brugeren har let ved at forstå hvordan spillet spilles. Inden gruppen begyndte at skrive koden til spillet, kiggede gruppen først på de krav der blev stillet, og analyserede dem. Derefter kunne gruppen opstille modeller til at designe programmet, med det mål at gruppen skulle have et ordentligt grundlag til at kode. Når det var gjort, begyndte gruppen på koden, og ændrede modellerne hen ad vejen i takt med at arbejdet gik fremad. Koden bliver naturligvis også testet, her ved brug af JUnit, for at kvalitetssikre at vi levere et produkt, der er så tæt på fejlfrit som muligt. Der bliver versionsstyret ved hjælp af Github og rapportskrivningen foregår på Google Docs.

Kravspecifikation[1]

Høj prioritet:

1. Der skal være 1 spillebræt.
2. Der skal mindst være 2 spillere, men man skal have muligheden for at være op til 6.
3. Der skal slås med to terninger på en gang
4. Spilleplade med felter som har 40 felter.
5. Der skal være 32 huse
6. Der skal være 12 hoteller
7. Der er et skødekort til en hver grund
8. Værdien af terningerne afgøre, hvor mange felter spilleren rykker frem.
9. Spilleren skal indeholde en pengebeholdning.
10. Pengebeholdning pr. spiller skal vises på skærmen.
11. Alle spillere starter med 30.000 kr.
12. Vinderen er fundet når der er to spillere tilbage, og en af dem går fallit.
13. En spiller går fallit når, en spiller skylder mere end personen ejer.
14. Når en spiller lander på et ledigt felt, får spilleren muligheden for at købe det for det beløbet der står på feltet.
15. Når en spiller lander på et felt en anden spiller ejer, skal spilleren betale baseret på hvor mange huse eller hoteller der er bygget på grunden.
16. For at kunne bygge et hotel, skal der være bygget 4 huse på hver grund.
17. Der kan kun være et hotel per grund.
18. Når en spiller lander på et felt, som spilleren selv ejer, skal spilleren ikke betale noget.
19. Brættet skal indeholde følgende felter: 1 startfelt, 6 chance felter, 1 helle felt, 1 gå i fængsel felt, 1 fængsel felt, 2 betal skat felter og 28 ejendoms felter.
20. Når en spiller er nået hele vejen rundt, og lander på start feltet, modtager spilleren 4000 kr.
21. Når en spiller lander på "Prøv løkken" skal spilleren trække det øverste kort i bunken og udføre ordren der står på kortet.
22. En spiller kan komme ud af fængsel ved at betale 1000 kr inden man kaster terningerne.

[1]CDIO 3
https://docs.google.com/document/d/1dC61tPbIbAocAlL080VWjfUlORjD_ZSs9LUETMi-NyA/edit#heading=h.n0k5idfcsjpt

Mellem prioritet:

1. Spillet skal indeholde 36 forskellige chancekort.
2. Når en spiller går fallit skal personen overdrage alt sin kreditor efter at have solgt eventuelle bygninger tilbage til banken.
3. Når en spiller har købt en ejendom skal det felt markeres som solgt til den spiller.
4. Man må først bygge huse og hoteller, når man ejer alle grunde i samme farve.
5. Man skal altid have muligheden for at sælge til banken for det halve af købsprisen, for hvert hus.
6. Når en spiller lander på "Gå i fængsel" bliver spilleren placeret på fængsel feltet.
7. En spiller kan komme ud af fængsel ved at slå 2 ens terninger, og rykke den pågældende sum af terningerne, spilleren får også en ekstra tur.
8. Hvis man slår to ens i terningekast får man en ekstra tur. Man skal rette sig efter det felt man lander på efter første kast og efter ekstra kastet.
9. En spiller kan pantsætte en af sine ubebyggede grunde for den pris der står på skøderne.
10. Spilleren beholder skøde kortene til en pantsat grund.

Lav prioritet:

1. En spillers pengebeholdning må gerne komme under 0, så længe at spilleren sælger sin ejendomme for bevare sin positive pengebeholdning.
2. Hvis ikke der er bygget nogen huse eller hoteller på en grund, og den samme spiller ejer alle grundene i samme farve, skal man betale dobbelt i husleje.
3. Man skal bygge husene jævnt, så man først skal bygge samme mængde hus på grundene for at kunne bygge endnu et hus på grundene.
4. En spiller kan komme ud af fængsel ved at benytte et af løsladelses kortene.
5. Man kan ikke blive i fængslet i mere end 3 omgange, på 3. omgang skal man betale 1000 kr. og flytte summen af terningerne.
6. Den fængslede spiller har ret til at købe grunde, huse og hoteller.
7. Hvis man slår to ens i terningekast tre gange i træk kommer man i fængsel.

Prioritering forklaret

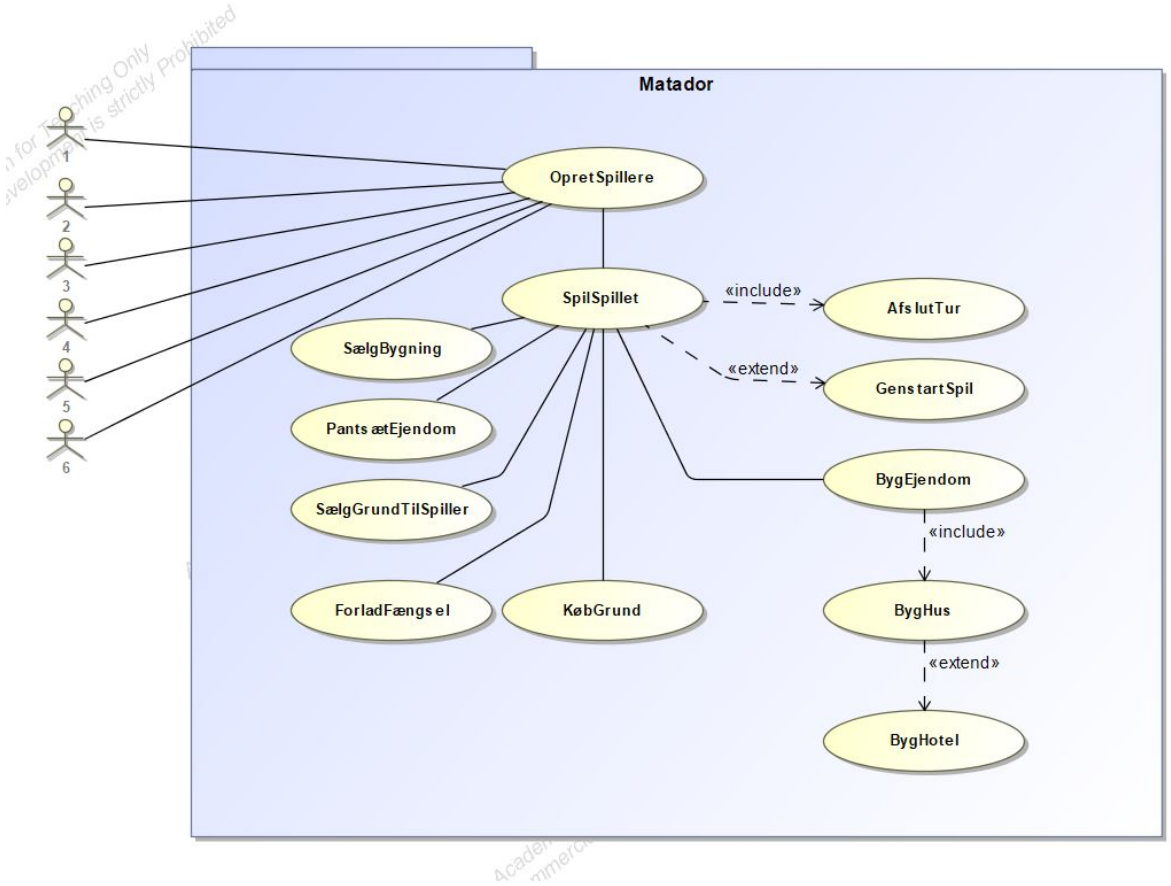
Vi har valgt vores prioriteringsliste ud fra, hvad vi følte der ville give det største udbytte for et matador spil set fra en spillers synspunkt. Derfor følte vi, at de krav i den højt prioriteret kategori, umiddelbart ville være de største features der udgør størstedelen af spillet. og hvis undladt ville føle som en stor mangel fra spillet.
I mellem-prioritets kategorien følte vi, at featuresene stadig gav udbytte for helheden af spillet, dog ville det ikke føles som en stor mangel, hvis de udgik fra spillet.
Lav-prioritets kategorien, er for features, som vi ikke følte ville give et væsentligt udbytte. Derfor ville det ikke føles en mangel, hvis de udgik fra spillet.

Kravspecifikationer, som vi ikke nåede

1. Når en spiller går fallit skal personen overdrage alt sin kreditor efter at have solgt eventuelle bygninger tilbage til banken.
2. En spillers pengebeholdning må gerne komme under 0, så længe at spilleren sælger sin ejendomme for bevare sin positive pengebeholdning.
3. Hvis ikke der er bygget nogen huse eller hoteller på en grund, og den samme spiller ejer alle grundene i samme farve, skal man betale dobbelt i husleje.
4. En spiller kan komme ud af fængsel ved at benytte et af løsladelses kortene.
5. Man kan ikke blive i fængslet i mere end 3 omgange, på 3. omgang skal man betale 1000 kr. og flytte summen af terningerne.

De fleste højt prioriteret kravspecifikationer blev implementeret. Dog er der nogle kravspecifikationer, som vi ikke nåede.
Vi implementerede de kravspecifikationer, som vi følte nødvendige at implementere for at skabe et fornuftigt og fyldestgørende matador spil.

Use cases[2]



Figur 1: Use case diagrammet.

I use case diagrammet har vi fundet alle de use cases det kunne opstå ved at bruge dette produktet. Vi har derefter valgt at lave en prioritering af de use cases.

Højt prioritet

- OpretSpiller
- KøbGrund
- KøbEjendom
    - KøbHuse
    - KøbHotel

Mellem prioritet

- SælgGrund
- SælgEjendom
- AfslutTur

[2]CDIO 3
https://docs.google.com/document/d/1dC61tPbIbAocAIL080VWjfUlORjD_ZSs9LUETMi-NyA/edit#heading=h.n0k5idfcsjpt

- ForladFængsel

Lav prioritet
- PantsætEjendom
- GenstartSpil
- SpilSpillet

Da det ikke er alle use cases der er lige vigtige, har vi valgt at fokusere på dem, der er mest essentielle for at kunne få et brugbart produkt i sidste ende. Gruppen har også identificeret use casen "KøbHuse", som er med til at beskrive use casen "KøbEjendom" og gør den nemmere at forstå. Dertil har vi valgt at fully dressed dem alle vores use cases, for at få en bedre forståelse for, hvad hver use case indebærer.

Vi har desværre ikke fået nået at lave følgende use cases: SælgGrund, PantsætEjendom og GenstartSpil. Man kan godt forlade fænget, men ForladFængel bliver ikke brugt som en reel use case, da brugeren ikke har noget med det at gøre.

Use case: SpilSpillet
ID: 1
Brief description: Efter man starter spillet, bliver man bedt om at vælge hvor mange spillere man er (2-6) og derefter bliver spillerne oprettet og så kan man gå i gang med spillet.
Primary actors: Spiller 2-6.
Secondary actors: Ingen.
Preconditions: Der skal være minimum 2 spillere for at spille spillet.
Main flow: 1. Man trykker spil på startsiden. 2. Derefter genererer spillet 2-6 spillere, så man kan skiftes om at slå med terningerne. 3. Man starter med 30000 kr. 4. Spillerne lander på et felt på brættet baseret på hvad de slog med terningerne. 5. Spillerne lander på et felt og effekt sker baseret på hvilket felt de lander på a. Spiller skal købe en ledig ejendom. b. Spiller skal betale skat c. Spiller skal trække et chance kort d. Spiller ryger i fængsel e. Spilleren lander på sin egen grund og skal ikke gøre noget. f. Når spilleren har gået spillebrættet rundt en hel omgang modtager spilleren 4000 kr. 6. Vinderen er fundet når der er to spillere tilbage, og en af dem går fallit.
Postconditions: Spillet erklærer en vinder, når den ene af de to spillere går fallit.
Alternative flows:

Use case: GenstartSpil
ID: 2
Brief description: Spillerne genstarter spillet, når der er fundet en vinder.
Primary actors: Spiller 2-6
Secondary actors: Ingen.
Preconditions: 1. Spillerne skal have startet spillet, før de kan genstarte spillet. 2. Der skal være erklæret en vinder, som sker når en person går fallit, så ser man hvem har flest penge. 3.
Main flow: 1. Use casen starter når spillerne er klikket til genstartspil. 2. Spillet starter helt forfra, når de klikker gestartspil.
Postconditions: Systemet sletter gamle score og starter et helt nyt spil. Der skal ikke oprettes spiller igen, hvis der er samme antal spiller som tidligere spil.
Alternative flows: Ingen

Use case: OpretSpil
ID: 3
Brief description: Inden spillet starter, opretter vores spil en spilleplade og spillere, som bruges til at spille spillet. Man bliver spurgt om hvor mange spillere man er (2 til 6).
Primary actors: Spiller 2-6.
Secondary actors: Ingen.
Preconditions: ● Man har åbnet spillet
Main flow: 1. Man klikker på det antal spillere, man har brug for. 2. Vælger hvilken karakter man vil være 3. Spillepladen bliver genereret 4. Spillet går i gang
Postconditions: ● Der vises deres karakter og pengebeholdning på skærmen. ● Spillerne og spillepladen er oprettet, og man er klar til at spille.
Alternative flows: Ingen.

Use case: ForladFængsel
ID: 4
Brief description: Betale sig ud af fængslet, eller brug forlad fængsel chancekort
Primary actors: Spiller 2 - 6.
Secondary actors: Ingen.
Preconditions: ● En spiller sidder i fængsel
Main flow: 1. Spilleren betaler sig ud af fængslet 2. Spilleren kaster sin terning
Postconditions: Spilleren kan fortsætte med spillet, efter løsladelsen.
Alternative flows: Hvis man har chance kortet, der får en ud af fængslet, kan man bruge kortet for at komme ud i stedet for at betale.

Use case: KøbGrund
ID: 5
Brief description: En spiller lander på en ledig grund, og skal købe den.
Primary actors: Spiller 2 - 4.
Secondary actors: Ingen.
Preconditions: ● Spillet er Startet ● Spilleren har landet på en ledig grund
Main flow: 1. Spilleren betaler den mængde af penge, som grunden kræver
Postconditions: Spilleren ejer den købte grund.
Alternative flows: 1. Hvis spilleren har ikke råd til grunden, er spilleren gået fallit 2. Hvis der trækkes et chancekort, som gør at man kan rykke hen til enhver plads, og der ikke er nogen ledige grunde. Kan man købe en grund fra en anden spiller.

Use case: KøbEjendom
ID: 6
Brief description: En spiller lander på en ledig grund, og skal købe den.
Primary actors: Spiller 2 - 6.
Secondary actors: Ingen.
Preconditions: ● Spillet er Startet ● Spilleren har landet på en ledig grund.
Main flow: 1. Spilleren betaler den mængde af penge, som grunden kræver 2. Spilleren har også mulighed for at låne penge fra banken.
Postconditions: Spilleren ejer den købte grund.
Alternative flows: 1. Hvis spilleren har ikke råd til grunden, er spilleren gået fallit

Use case: KøbHuse
ID: 7
Brief description: Spilleren køber et hus på sin grund, så han/hun kan opkræve husleje hvis en anden spiller lander på deres grund.
Primary actors: Spiller 2 - 6.
Secondary actors: Ingen.
Preconditions: ● Spilleren skal eje alle grundene af samme farve.
Main flow: 1. Spilleren køber alle grunde af samme farve 2. Spilleren køber huse til sine grunde
Postconditions: Nu har spilleren købt sine huse, og kan nu opkræve flere penge af spillere der lander på deres felter.
Alternative flows: Ingen.

Use case: KøbHotel
ID: 8
Brief description: Spilleren køber hoteller på sin ejendom.
Primary actors: Spiller 2 - 6.
Secondary actors: Ingen.
Preconditions: ● Spilleren ejer en grund og har bygget 4 huse.
Main flow: 1. spilleren køber huse til sin grunde. 2. spilleren køber hoteller til sin grunde
Postconditions: Spilleren eje hotellet og det skal der vises på skærmen. Spilleren er i stand til at opkræve leje fra de spillere der lander på grunden.
Alternative flows: Ingen.

Use case: PantsætEjendom
ID: 9
Brief description: Spilleren kan sætte sin ejendom som inaktiv og modtage et beløb for det. Hvis spilleren vil købe ejendommen tilbage, skal spilleren betale 10 % oven i.
Primary actors: Spiller 2 - 6.
Secondary actors: Ingen.
Preconditions: Spilleren ejer en grund, som ikke har nogen bygninger. Spillerens tur
Main flow: 1. Spilleren klikker på pantsæt ejendom 2. Spilleren vælger hvilken grund der ønskes at blive pantsat 3. Spilleren modtager de penge der står på skødet 4. Grunden bliver sat som inaktiv
Postconditions: Spilleren har modtaget penge svarende til beløbet der er på skødet grunden er pantsat.
Alternative flows: 1. Spilleren trykker på ophæv pantsætning 2. Spilleren betaler pantsætnings værdien + 10% tilbage til banken 3. Grunden bliver sat som aktiv igen.

Use case: SælgBygning
ID: 10
Brief description: Spilleren sælger sin bygning når, spilleren har lyst til.
Primary actors: Spiller 2 - 6.
Secondary actors: Ingen.
Preconditions: ● Spilleren har bygget en bygning. ● Det er spillerens tur. ● Spilleren skylder ikke penge til banken, hvis bygningen var bygget ved hjælp af banken.
Main flow: 1. Spilleren klikker på knappen SælgBygning for at sælger sin bygning. 2. Derefter vælger spilleren hvilken bygning spilleren vil sælge.
Postconditions: ● Spilleren ejer ikke længere bygningen. ● Beløbet skal tilføjes til spillerens pengebeholdning. ● Spilleren kan ikke modtage husleje længere.
Alternative flows: ● Spilleren sælger sin bygning til banken, hvis spiller skylder penge til banken.

Use case: SælgGrundTilSpiller
ID: 11
Brief description: En spiller sælger en af sine grunde til en anden spiller for en aftalt pris
Primary actors: Spiller 2 - 6.
Secondary actors: Den købende spiller.
Preconditions: ● Den nuværende spiller ejer en grund ● Der skal ikke være nogen huse eller hotel på grunden
Main flow: 1. Spilleren klikker på “Sælg til anden spiller” knap 2. Spilleren vælger hvilken spiller ejendommen skal sælges til 3. Spilleren der modtager ejendommen indtaster et beløb, som der er aftalt individuelt mellem spillerne, og klikker ok. 4. Det aftalte beløb bliver trukket fra den modtagne spiller, og tilføjet til den sælgende spiller.
Postconditions: Grunden er blevet solgt til den anden spiller.
Alternative flows: Ingen.

Use case: AfslutTur
ID: 12
Brief description: Spilleren afslutter sin tur.
Primary actors: Spiller 2 - 6.
Secondary actors: Ingen.
Preconditions: Spilleren har påbegyndt sin tur, og har slået med terningerne.
Main flow: 1. Spilleren klikker på “Afslut tur” knappen
Postconditions: Spilleren har afsluttet sin tur, og turen går videre til den næste spiller
Alternative flows: Ingen.

Analyse

Vores diagrammer og modeller findes også under bilag, hvor man bliver ført hen til et google drive link til billedet med bedre opløsning og zoom funktion.

Risikoanalyse[3]

- Undervurderet projekt størrelse
- Planlægning af tid
- Stor design omstrukturering
- Fejlet system integration
- Ikke leveret system dele
- Misforstået krav
- Sygdom blandt holdmedlemmer
- Manglende erfaring

Risici	Sandsynlighed	Skade	Risiko
Undervurderet projekt størrelse	1	5	5
Planlægning af tid	4	5	20
Stor design omstrukturering	1	2	2
Fejlet system integration	4	2	8
Ikke leveret system dele	2	5	10
Misforstået krav	1	1	1
Sygdom blandt holdmedlemmer	5	2	10
Manglende erfaring	4	5	20

[3] CDIO 1 - https://docs.google.com/document/d/1-XGd8bKliHMJeK0lLX8EHQ7FTbhvrboE7ldwqu_zQbQ/edit

Lykkekort prioriteringsliste[4]

Høj prioritet

- Lykkekort 3-19
- Lykkekort 24-25
- Lykkekort 27
- Lykkekort 29-31
- Lykkekort 33-34
- Lykkekort 35 og 36

Mellem prioritet

- Lykkekort 21-23
- Lykkekort 26
- Lykkekort 32

Lav prioritet

- Lykkekort 20
- Lykkekort 1 og 2
- Lykkekort 28

Vi har her prioriteret lykkekortene, som indgår i et matador spil ud fra listen, der er blevet givet. Funktionaliteten af spillet bliver prioriteret højest. Det vil sige at alle de kort, hvor en spiller skal rykke plads, gøre noget med fængslet, får penge, eller skal give penge har fået den højeste prioritet, dog med nogle enkelte undtagelser.

[4] https://drive.google.com/file/d/1KEOv1JM8HPDRva1MNPQn-mOV5ZKs39ke/view?usp=sharing

Klasser

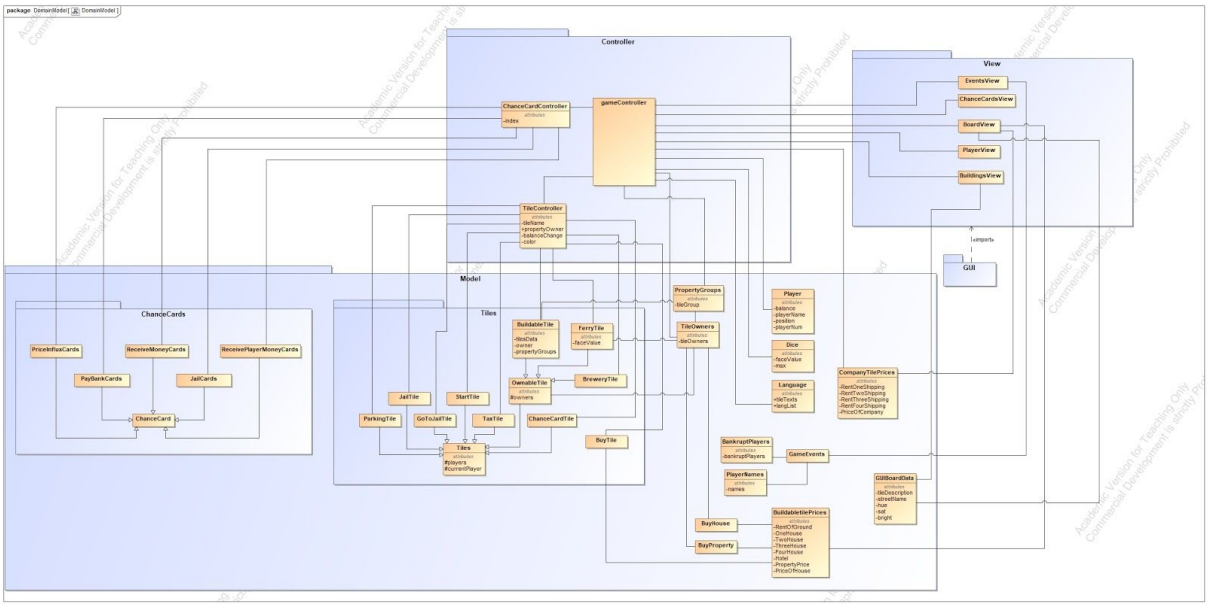
Inden vi begyndte på koden, fik vi dannet os et overblik over, hvilke klasser der ville være relevante at have med. Vi har omstruktureret vores arkitektur fra sidste projekt til at følge MVC (Model View Controller) princippet i stedet, da det giver en "High cohesion" og "Low coupling". Vi har defineret følgende klasser i disse forskellige packages:

- Controller
    - ChanceCardController
    - GameController
    - TileController
      Test
- Model
    - ChanceCards package
        - *ChanceCard*
        - GoToJailCards
        - JailCards
        - PayBackCards
        - PriceinfluxCards
        - ReceiveMoneyCards
        - ReceivePlayerMoneyCards
    - Tiles package
        - BreweryTile
        - BuildableTile
        - ChanceCardTile
        - FerryTile
        - GoToJailTile
        - JailTile
        - *OwnableTile*
        - ParkingTile
        - StartTile
        - TaxTile
        - *Tile*
    - Player
        - BankruptPlayers
        - Player
        - PlayerNames
        - PlayerColor
    - BuildableTilePrices
    - BuyHouse
    - BuyProperty

    - CompanyTilePrices
    - Dice
    - GameEvents
    - GUIBoardData
    - Language
    - PropertyGroups
    - SellHouse
    - TileOwners
    - Dice
    - Language
    - OwnablestilesData
    - Displaymanager
    - Language
    - Player
    - TileDice
    - Displaymanager
    - Player
    - Tile
- View
    - BoardView
    - BuildingsView
    - ChanceCardsView
    - EventsView
    - PlayerView
    - PropertyOwnerView

Herefter valgte vi at koble de enkelte klasse sammen i vores domænemodel for at give et bedre overblik over, hvordan klasserne ville kommunikere indbyrdes.

Domænemodel[5]

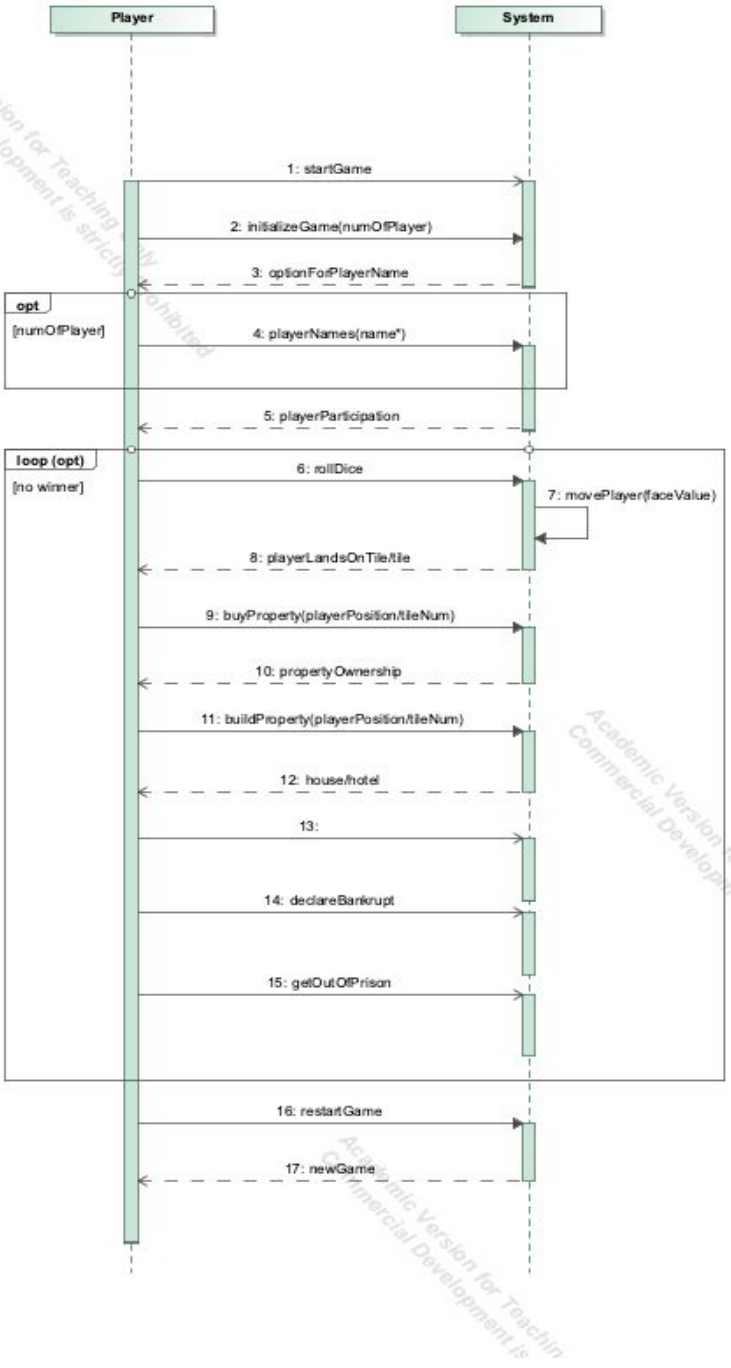


Figur 2 : Domænemodel.

På figur 2 har vi vores domænemodel, som indeholder vores klasser i forskellige pakker. Ud fra domænemodel kan vi aflæse de vigtige abstraktioner, statiske informationer, begreber og sammenhænge mellem klasserne. Hvilket gør det nærmere for at lave et design klassediagram.

[5] https://drive.google.com/file/d/1iWz8Mm1G11E6ELEWtW-yI06ty57pnX4a/view?usp=sharing

System Sekvensdiagram

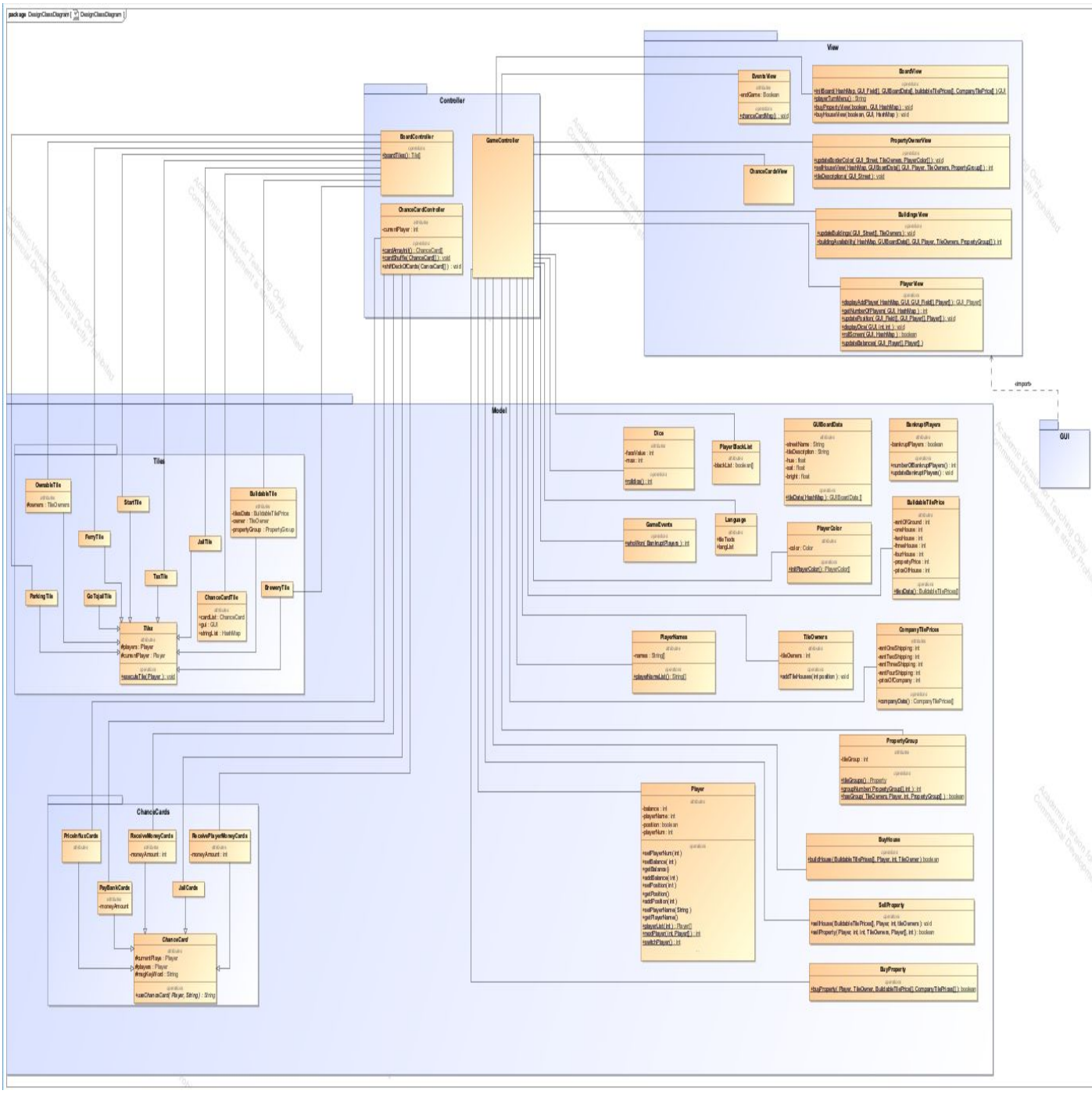


Figur 3 : System sekvensdiagram.

I system sekvensdiagrammet kan man se hvordan en normal tur forløber i spillet, der bliver brugt loops for at spilleren kan få flere muligheder i en tur.

Design

Design klassediagram



Figur 4 : Design klassediagram.

Vi delte klasserne op i forskellige pakker, i forhold til deres typer. Tiles pakken indeholder alle felterne, og ChangeCards indeholder alle lykke kortene osv. I Klassediagrammet (figur 4)

kan man tydeligt se alle klassernes attributter, metoder og deres sammenhænge. Hvilket gør det meget nemmere at implementere. Klasser der har samme attributter og metoder er blev nedarvet fra en basis superklasse såsom i pakken ChanceCards og Tiles pakken.

MVC:

Model:
Model klasser er de klasser, der bliver tildelt data behandling. I dette projekt havde det for eksempel været Dice klassen der indeholder rollDice() som laver de random terningekast.

View:
View klasser er de klasser, der bliver alt hvad en potentiel bruger af programmet ville se. I dette projekt, er det altså alle de klasser der behandler GUI'en. De bør ikke have data behandling, men kun sørge for at vise beskeder og sende bruger inputs videre som skal bruges i model klasser.

Control.
Control klasser er de klasser, som sammensætter model og view klasserne så de kan kommunikere med hinanden. Det er altså de klasser som kontrollerer model og view klasserne. I control klasser gælder det så vidt som muligt om at kunne nøjes med at oprette variabler, og bruge metoder i andre klasser.

I projektet er alle klasserne delt op i enten Model, View eller Control pakken. Så vidt som muligt er MVC blevet brugt i projektet med enkelte undtagelser som i ChanceCardTile, som er en model klasse hvor GUI'en også bliver brugt. Dette kunne nok have været undgået, men det ville have krævet noget omstrukturering. Brug af MVC har bestemt gjort, at det har været nemmere at gennemskue hvordan opsætningen skulle være, hvilket også gør programmet nemmere at læse.

Sekvensdiagram

Grundet at vores sekvensdiagram er så stort, har vi lagt det som et bilag:
https://drive.google.com/file/d/1XT_F72Z2ToUkWj92Q9RE9RBsi_xcNAyp/view?usp=sharing

I sekvensdiagrammet, kan man se, at der i starten bliver initialiseret et spil, derefter i loopet kan man se timeline over en almindelig tur for en spiller, som bliver gentaget for hver tur for alle spillerne. loopet bliver kun brudt, når der er fundet en vinder til spillet. Derfor har vi kaldt dette loop for game loopet.

GRASP Patterns

Vi forsøgte at anvende GRASP Patterns i projektet, især de vigtigste patterns, såsom høj samhørighed, lav kobling og information ekspert. Men, på grund af manglende programmering erfaring kunne vi desværre ikke opnå det helt, som vi forventede.

Vi har opnået en lidt høj samhørighed og information ekspert. Det kan vi se i vores klassediagram. F.eks. klassen BoardController og ChanceCardController indeholder informationer om chance kortene og tilesene. Det vil sige, pattern information ekspert blevet anvendt i de to klasse.

Vi synes at alle klasserne i projektet er ansvarlige indenfor deres område. Dvs. klasserne indeholder de nødvendige metoder og attributter for at beskrive klassen, og hvad klassen bør indeholde.
I forhold til lav coupling kan vi ikke understøtte og sige at vi opnåede det. Men, vi prøvede at undgå at kalde en klasses metode eller variable for mange steder for at opnå lav kobling.
F.eks. alle klasserne i View pakken blev kun kaldt i GameController klassen.
Det er gør så skal vi ikke skal gå ind i hver klasse og rette dem, hvis der sker nogen ændring.

Implementering

Udviklingsprocessen af koden er lavet i iterationer baseret på Unified Process metoden. Features bliver udviklet i små ryk som projektet skrider fremad. Det gør at vi altid har et program der fungerer og fører til en nemmere at implementering af ny funktioner. Vores implementering er bygget op sådan at det følger vores model så tæt som muligt, og dermed også benyttet GRASP til såvidt muligt at kunne gøre koden nemmere at forstå, læse og senere brug i et senere projekt. Vi har så vidt muligt sørget for at holde lav kobling og høj samhørighed, og nedarvning for at undgå at skrive unødvendigt meget kode. Dog har vi været nødt til at skrotte nogle features for at kunne nå at få et færdig produkt uden de store fejl. Disse features er her i blandt: salg af grund, auktion, pantsætning og de lavt prioriterede lykke kort. Disse features er blevet valgt fra pga. manglende erfaring og tid.
Under udviklingen har vi gjort brug af GitHub for at kunne arbejde fleksibelt på koden og have muligheden for at teste og implementere på andre branches.

Kildekoden er blevet beskrevet i dette afsnit, vi har valgt at tage udgangspunkt i nedarvning, da det er en fundamental byggesten i hvordan vores spil fungerer. Hvordan vi håndterer oversættelsen af sprog i programmet, er også vigtig, da det gør teksthåndtering nemt og simpelt at redigere, da det er bundet op på nøgleord. En anden funktion der er vigtig er filtrere fallitte spillere ud af spiller puljen, så de ikke får en tur uden at have penge at gøre godt med.

Nedarvning

I projektet har gruppen valgt at bruge nedarvning et par steder i koden. Et af disse steder er i håndteringen af spillebrættet, hvor vi har lavet en abstrakt klasse med en abstrakt metode som er blevet kaldt executeTile. Denne metode er lavet til at blive brugt af felterne på spillebrættet, som et standard eksekverings kald. executeTile metoden er så til at blive udfyldt i de subklasser den bliver nedarvet til ved brug af @Override.

@Override er en feature i java som tillader subklasser at lave en specifik implementering af en metode som er givet via nedarvning fra dens superklasse. Dette vil sige at subklassen nedarver en metode, som har samme navn, parameter, signature og returnerings type fra dens superklasse, derefter vil metoden i subklassen overskrive metoden i superklassen ved brug af @Override.
Overskrivning af metoder via nedarvning er en måde at opnå polymorfi, så den version af metoden der vil blive eksekveret er bestemt af det objekt som kalder på klassen.

```
1   package Model.Tiles;
2
3   import Model.Player;
4
5   public abstract class Tile {
6       protected Player[] players;
7       protected Player currentPlayer;
8
9       public Tile(Player[] players) { this.players = players; }
12
13      public abstract void executeTile(Player currentPlayer);
14  }
15
```

Figur 5: Tile klasse i Tiles package

Der er også et par forskellige felter, som man kan købe i spillet. Her har gruppen lavet en ny klasse, som nedarver fra Tile klassen. Denne nye metode vil tillade andre klasser, der nedarver fra denne klasse at blive ejet af en player.

```java
package Model.Tiles;

import ...

public abstract class OwnableTile extends Tile {

    protected TileOwners owners;

    public OwnableTile(Player[] players, TileOwners owners) {
        super( players);
        this.owners = owners;
    }

    @Override
    public void executeTile(Player currentPlayer) {

    }
}
```

Figur 6 : OwnableTile klasse i Tiles package

En af disse klasser, hvor vi bruger begge nedarvede klasser er breweryTile, som både skal bruge executeTile, men også skal kunne blive købt af en spiller. Klassen i dette tilfælde skal kunne opkræve leje fra andre spillere og tjekke for, om spilleren der ejer feltet også ejer det andet bryggeri, og udregner hvad den endelige leje skal være.

```java
public class BreweryTile extends OwnableTile {
    public BreweryTile(Player[] players, TileOwners owners) { super(players, owners); }


    /**
     * Checks the ownership of tile 12 and 28, and applies a multiplier if both or one of them
     * is owned by the same player. A dice roll is also taken into account in the rent.
     * The player pays the final result of the rent to the owning player.
     * @param currentPlayer The player landing on the tile
     * @param sumOfDice Sum of 2 dice rolls
     */
    @Override
    public void executeTile(Player currentPlayer, int sumOfDice) {
        int playerPosition = currentPlayer.getPosition();
        int currentPlayerNum = currentPlayer.getPlayerNum();
        int rent = 100;

        // Checks of there are any players owning the property
        if(owners.getTileOwner(playerPosition) != -1){
            // Checks that the player landing on the tile does not own the tile
            if (owners.getTileOwner(playerPosition) != currentPlayerNum) {
                int priceMultiply = 0;
                // Assigns the owner of the tile to the variable breweryOwner
                int breweryOwner = owners.getTileOwner(playerPosition);

                // One ups the multiplier if the owner owns tile 12 (Tuborg)
                if(owners.getTileOwner( position: 12) == breweryOwner) {
                    priceMultiply++;
                }

                // One ups the multiplier if the owner owns tile 28 (Coca Cola)
                if(owners.getTileOwner( position: 28) == breweryOwner) {
                    priceMultiply++;
                }

                // Calculates the final rent
                rent = rent * priceMultiply * sumOfDice;

                //Pays the owner money from the currentPlayer
                currentPlayer.addBalance(-rent);
                players[breweryOwner].addBalance(rent);

            }
```

Figur 7 : executeTile() Metode fra BreweryTile klassen i Tiles package

Dette tillader os at lave en klasse, som vi kan bruge som en information expert, hvor vi initialisere dem i et objekt array. Vi kan dermed kalde på metoden executeTile på det felt, som spilleren lander på, hvilket så udfører handlingen for den korresponderende felt plads.

```
15 @    public static Tile[] boardTiles(BuildableTilePrices[] prices, Player currentPlayer,
16          Tile[] boardTiles = new Tile[40];
17
18          boardTiles[0] = new StartTile(players);
19          boardTiles[1] = new BuildableTile(players, owners, prices,propertyGroups);
20          boardTiles[2] = new ChanceCardTile(players, cardList, gui, stringList);
21          boardTiles[3] = new BuildableTile(players, owners, prices,propertyGroups);
22          boardTiles[4] = new TaxTile(players);
23          boardTiles[5] = new FerryTile(players,owners);
24          boardTiles[6] = new BuildableTile(players, owners, prices,propertyGroups);
25          boardTiles[7] = new ChanceCardTile(players, cardList, gui, stringList);
26          boardTiles[8] = new BuildableTile(players, owners, prices,propertyGroups);
27          boardTiles[9] = new BuildableTile(players, owners, prices,propertyGroups);
28          boardTiles[10] = new JailTile(players);
29          boardTiles[11] = new BuildableTile(players, owners, prices,propertyGroups);
30          boardTiles[12] = new BreweryTile(players,owners);
31          boardTiles[13] = new BuildableTile(players, owners, prices,propertyGroups);
32          boardTiles[14] = new BuildableTile(players, owners, prices,propertyGroups);
33          boardTiles[15] = new FerryTile(players,owners);
34          boardTiles[16] = new BuildableTile(players, owners, prices,propertyGroups);
35          boardTiles[17] = new ChanceCardTile(players, cardList, gui, stringList);
36          boardTiles[18] = new BuildableTile(players, owners, prices,propertyGroups);
37          boardTiles[19] = new BuildableTile(players, owners, prices,propertyGroups);
38          boardTiles[20] = new ParkingTile(players);
39          boardTiles[21] = new BuildableTile(players, owners, prices,propertyGroups);
40          boardTiles[22] = new ChanceCardTile(players, cardList, gui, stringList);
41          boardTiles[23] = new BuildableTile(players, owners, prices,propertyGroups);
42          boardTiles[24] = new BuildableTile(players, owners, prices,propertyGroups);
43          boardTiles[25] = new FerryTile(players,owners);
44          boardTiles[26] = new BuildableTile(players, owners, prices,propertyGroups);
45          boardTiles[27] = new BuildableTile(players, owners, prices,propertyGroups);
46          boardTiles[28] = new BreweryTile(players,owners);
47          boardTiles[29] = new BuildableTile(players, owners, prices,propertyGroups);
48          boardTiles[30] = new GoToJailTile(players);
49          boardTiles[31] = new BuildableTile(players, owners, prices,propertyGroups);
50          boardTiles[32] = new BuildableTile(players, owners, prices,propertyGroups);
51          boardTiles[33] = new ChanceCardTile(players, cardList, gui, stringList);
52          boardTiles[34] = new BuildableTile(players, owners, prices,propertyGroups);
53          boardTiles[35] = new FerryTile(players,owners);
54          boardTiles[36] = new ChanceCardTile(players, cardList, gui, stringList);
55          boardTiles[37] = new BuildableTile(players, owners, prices,propertyGroups);
56          boardTiles[38] = new TaxTile(players);
57          boardTiles[39] = new BuildableTile(players, owners, prices,propertyGroups);
```

Figur 8 : boardTiles() Metode fra BoardController klassen

Oversættelse af sprog

Vi har i programmet implementeret en funktionalitet, der gør det let at udvide sproget i spillet til andre sprog. Dog lige nu har vi kun fuldt implementeret dansk som et sprog.
Vi gør dette ved brug af et hashmap, som læser en tekst-fil (.txt) ud fra, hvilke sprog der er blevet valgt, som i vores tilfælde er dansk og vil derfor læse Danish.txt filen.
Vi har lavet funktionaliteten sådan, at man skriver et keyword ind for at få den ønskede tekst streng ud. Dette gør det nemt at ændre, eller udvide sproget i spillet, da der ikke er behov for at ændre keywordsene i koden, men blot behøver at tilføje en ny tekst-fil for det ønskede sprog, og oversætte output linjerne, der ligger på lige linje numre, til det ønskede sprog.

```java
public class Language {

    /**
     * This method reads a .txt file with a specific setup and returns a HashMap<String, String> with the content
     * (Use English.txt as reference. Uneven numbered lines are keywords for the even lines under them which contains the respective string texts)
     *
     * @param language String to change the .txt file path you get in the switch case extra cases can be added for more languages.
     * @return Returns the HashMap<String, String> with the content of the file loaded.
     * @throws IOException
     */
    public static HashMap<String, String> languageInit(String language) throws IOException {

        Path path;
        switch (language.toLowerCase()) {
            case "danish":
                path = Paths.get( first: "Danish.txt");
                break;
            default:
                path = Paths.get( first: "English.txt");
        }
        BufferedReader reader = new BufferedReader(new FileReader(String.valueOf(path.toAbsolutePath())));
        BufferedReader lineReader = new BufferedReader(new FileReader(String.valueOf(path.toAbsolutePath())));

        int lines = 0;

        while (lineReader.readLine() != null) lines++;

        HashMap<String, String> langList = new HashMap<>();

        for (int i = 0; i < lines; i++) {
            langList.put(reader.readLine(), reader.readLine());

        }
        return langList;
    }
}
```

Figur 9 : Language klasse i Model package

Alle keywords ligger på de ulige linje numre, og når der bliver søgt på et keyword, ville der blive returneret den string der står på næste linje fra keywordet.

```
endTurnMsg
Afslut tur
chooseBuildingProperty
Vælg en grund du vil bygge hus eller hotel på.
playerTurnChoice
 vælg hvad du vil gøre i din tur
buyPropertyMsg
Køb Grund
buildOnPropertyMsg
Byg på en grund
boughtPropertyMsg
Du har købt grunden.
notBoughtPropertyMsg
Du har ikke købt grunden pga. du mangler penge, eller at der er en der ejer den i forvejen.
```

Figur 10 : Danish.txt udsnit af tekst-fil

Som set på figur 10 har hvert keyword en korresponderende linje tekst.

For at kalde på den ønskede besked i koden, skal man blot kalde på stringList.get() og så skrive keywordet

```
public static String playerTurnMenu(GUI gui, HashMap<String, String> stringList, GUI_Player[] guiPlayer,
    String buttonPress;
    String screenMsg = guiPlayer[currentPlayerNum].getName() + stringList.get("playerTurnChoice");
    String buttonMsg = stringList.get("endTurnMsg");
    String buttonMsg1 = stringList.get("buyPropertyMsg");
    String buttonMsg2 = stringList.get("buildOnPropertyMsg");
    String buttonMsg3 = stringList.get("sellOnPropertyMsg");
    buttonPress = gui.getUserButtonPressed(screenMsg, buttonMsg, buttonMsg1, buttonMsg2, buttonMsg3 );
    return buttonPress;
```

Figur 11 : Eksempel på Brug af hashmap i koden

Håndtering af faillite spillere

Mens spillet bliver spillet, ville der altid være spillere der går fallit. Og derfor er der et behov for en feature, for hvad der skal gøres med disse spillere, da de ikke skal have mulighed for at spille videre, efter de er gået fallit. Det er en vigtig feature, fordi der også skal være en måde at holde styr på, hvem der er den vindende spiller, altså den sidste spiller, som ikke er gået fallit.
Derfor har vi et boolean array og metoden updateBankruptPlayers (Figur 12). Denne metode kigger på om spillernes pengebeholdning er nået nul eller mindre. Når en spiller så er gået fallit, ville de blive markeret fallit i bankruptPlayers arrayet.

Vi har også en numOfBankruptPlayers, som returnerer mængden af fallitte spillere, hvilket vi bruger til at bestemme, om der er fundet en vinder.

```java
public class BankruptPlayers {
    private boolean [] bankruptPlayers;
    public BankruptPlayers(Player[] players){ this.bankruptPlayers = new boolean[players.length]; }

    /**
     * counts how many players went bankrupt
     * @return number of bankrupt players
     */
    public int numOfBankruptPlayers(){
        int numOfBankruptPlayers = 0;
        for (int i = 0; i < bankruptPlayers.length; i++) {
            if(bankruptPlayers[i]){
                numOfBankruptPlayers++;
            }
        }
    return numOfBankruptPlayers;
    }

    public void setBankruptPlayer(boolean isBankrupt, int numOfPlayer) { bankruptPlayers[numOfPlayer] = isBankrupt; }

    public boolean [] getBankruptPlayers() { return bankruptPlayers; }

    /**
     * updates array of bankrupt players, when a player goes bankrupt
     * @param players
     */
    public void updateBankruptPlayers(Player[] players){
        for (int i = 0; i < players.length; i++) {
            if (players[i].getBalance() <= 0){
                bankruptPlayers[i] = true;
            }
        }
    }
}
```

Figur 12 : BankruptPlayers klasse i Model.Player package

switchPlayer metoden (Figur 13) bliver brugt til at skifte til den næste spiller, efter en spiller har afsluttet sin tur. Her bliver der brugt bankruptPlayers til at sørge for, at den næste spiller ikke er gået fallit. Hvis en spiller så er gået fallit, ville deres tur blive "sprunget over'' og gå videre til den næste spiller, som ikke er gået fallit.

```
125 @   public static int switchPlayer(int index, Player[] players, BankruptPlayers bankruptPlayer){
126         int playerNum = 0;
127         while(true) {
128             playerNum = Player.nextPlayer(index, players);
129             if(!bankruptPlayer.getBankruptPlayers()[playerNum]){
130                 break;
131             }
132         }
133         return playerNum;
```

Figur 13 : switchPlayer() Metode i Player klassen

Vi bruger numOfBankruptPlayers() metoden til at bestemme, hvem der har vundet i whoWon() metoden, her kigger vi på, om alle bortset fra en spiller er gået fallit, og så returnerer den resterende spiller, som så er den vindende spiller.

```
11 @    public static int whoWon(BankruptPlayers bankruptPlayers) {
12          int allPlayers = bankruptPlayers.getBankruptPlayers().length;
13          int winningPlayerNum = -1;
14
15          if(bankruptPlayers.numOfBankruptPlayers() == allPlayers - 1){
16              for (int i = 0; i < allPlayers; i++) {
17                  if(!bankruptPlayers.getBankruptPlayers()[i]){
18                      winningPlayerNum = i;
19
20                  }
21              }
22          }
23          return winningPlayerNum;
```

Figur 14 : whoWon() metode i GameEvents klassen

Køb af grunde

Der er også behov for at spillerne kan købe grunde. Her har vi lavet metoden buyProperty (Figur 15). Denne metode gør det muligt for en spiller at købe den grund som spilleren står på, ved at sætte den nuværende spiller som ejer af den grund, og derefter trække pengene for grunden fra spillerens pengebeholdning. Dog ikke før den har tjekket, om den nuværende pris ikke er nul, for hvis den nuværende pris er nul, er det ikke ment for spilleren at have mulighed for at købe den grund. Bagefter tjekker den, om den nuværende spiller har råd til at købe den grund, som spilleren står på. Til sidst tjekker den også, om grunden faktisk er ledig og ikke ejet af en anden spiller, dette gør vi ved at se, om den er lig med -1. Denne værdi bruger vi for grunde, som endnu ikke er ejet af en spiller.

```
16 @    public static boolean buyProperty(Player currentPlayer, TileOwners owners, BuildableTilePrices[] buildTileData, Compa
17          boolean propertyBought= false;
18          int position = currentPlayer.getPosition();
19          int tilePrice = buildTileData[position].getPropertyPrice() + companyTilePrices[position].getPriceOfCompany();
20
21          if((buildTileData[position].getPropertyPrice() != 0) || (companyTilePrices[position].getPriceOfCompany() != 0)) {
22              if (currentPlayer.getBalance() > tilePrice) {
23                  if (owners.getTileOwner(position) == -1) {
24                      currentPlayer.addBalance(-tilePrice);
25                      owners.setTileOwner(position, currentPlayer.getPlayerNum());
26                      propertyBought = true;
27                  }
28              }
29          }
30
31          return propertyBought;
```

Figur 15 : buyProperty() metode i BuyProperty klassen

For at spilleren kan se at grunden er blevet købt eller ej, har vi buyPropertyView() metoden (Figur 16) som giver en besked til spilleren baseret på den boolean værdi der bliver returneret i buyProperty() metoden (Figur 15)

```
93      public static void buyPropertyView(boolean isBought, GUI gui, HashMap<String,String> stringList){
94          if(isBought){
95              gui.showMessage(stringList.get("boughtPropertyMsg"));
96          } else {
97              gui.showMessage(stringList.get("notBoughtPropertyMsg"));
98          }
99      }
```

Figur 16 : buyPropertyView() metode i BoardView klassen

Test

Blanding af chance kort

```java
@Test
public void cardShuffleMechanic(){
    Player[] players = Player.playerList( numOfPlayers: 2);
    ChanceCard[] chanceCards1 = new ChanceCard[40];
    ChanceCard[] chanceCards2 = new ChanceCard[40];

    // Assigns the same integers to both chance card arrays;
    for (int i = 0; i < chanceCards1.length; i++) {
        String a = Integer.toString(i);
        chanceCards1[i] = new JailCards(players,a);
        chanceCards2[i] = new JailCards(players,a);
    }

    for (int i = 0; i < chanceCards1.length; i++) {
        //System.out.println(i);
        assertTrue(chanceCards1[i].useChanceCard(players[0]).equals(chanceCards2[i].useChanceCard(players[0])));
    }

    int numOfEquals=0;

    for (int j = 0; j < 10000; j++) {
        // Shuffles both decks of cards.
        ChanceCardController.cardShuffle(chanceCards1);
        ChanceCardController.cardShuffle(chanceCards2);
        boolean a = false;

        // Checks for if an i position's string is equal. If it is a is true and the for loop breaks.
        // The chance of the forloop not breaking after the shuffle should be nearly impossible
        for (int i = 0; i < chanceCards1.length; i++) {
            if(!chanceCards1[i].useChanceCard(players[0]).equals(chanceCards2[i].useChanceCard(players[0]))){
                a = true;
                break;
            }
            numOfEquals++;
        }
        assertTrue(a);
    }
    System.out.println(numOfEquals);

}
```

Figur 17: Test af kort blandings metoden cardShuffle()

I denne Unit test ser vi på om vores blandingsmetode cardShuffle() fungerer. Vi vælger at lave to arrays, som bliver blandet hver for sig. Derefter bliver de sammenlignet med hinanden, for at se om de to arrays er identiske. Hvis de er identiske fejler testen, og hvis der er så meget som 1 plads der afviger fra sammenligningen består testen. Vi testede om hvor mange gange de første par pladser var ens med numOfEquals integeren. I to arrays med 40 pladser. der er blevet blandet fuldstændig tilfældigt vil der være en 1/40 chance for at to pladserne med det samme index har den samme værdi. Dette vil resultere i at numOfEquals ca. burde være 10000/40 = 250 plus en smule, hvis numOfEquals bliver lagt til, vil der igen være en 1/40 chance for at der bliver lagt til. Da der blev testet, blev numOfEquals mellem 246 og 286 hvilket betyder, at hvis ikke kortene er blandet perfekt er det i hvert fald ret tæt på.

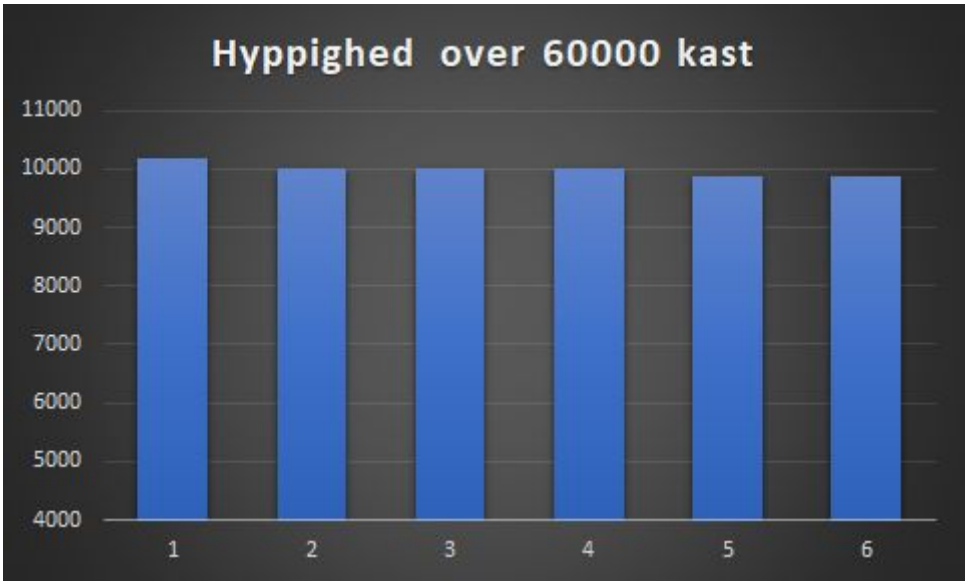
Dice[6]

```java
@Test
void diceRandomness() {
    Dice testdice = new Dice( value: 6);

    int[] dicesidestest = new int[6];
    int outOfRange = 0;

    for (int i = 0; i < 60000; i++) {
        testdice.rollDice();
        int facevalue = testdice.getFaceValue();

        switch (facevalue) {
            case 1: dicesidestest[0]++;
                break;
            case 2: dicesidestest[1]++;
                break;
            case 3: dicesidestest[2]++;
                break;
            case 4: dicesidestest[3]++;
                break;
            case 5: dicesidestest[4]++;
                break;
            case 6: dicesidestest[5]++;
                break;
            default: outOfRange++;
        }
    }

    for (int i = 0; i < 5; i++) {
        assertEquals( expected: 10000, dicesidestest[i],  delta: 400);
        assertEquals( expected: 0,outOfRange);
    }

    System.out.println("Dice facevalue results 1-6: ");
    System.out.println(dicesidestest[0]);
    System.out.println(dicesidestest[1]);
    System.out.println(dicesidestest[2]);
    System.out.println(dicesidestest[3]);
    System.out.println(dicesidestest[4]);
    System.out.println(dicesidestest[5]);
}
```

Figur 18: Test af tilfældigheden af metoden rollDice()

I denne test er der blevet slået med en 6 sidet terning 60000 gange ved hjælp af JUnit. Resultaterne ses nedenunder i figur 19 og giver os en indikation på at det fungerer, da der er lige så stor chance for at slå 1 som der er for at slå 6. Der vil altid være held involveret når man kaster med en terning og derfor har vi slået med terningen 60000 gange. I vores test med Junit har vi også givet testen et delta på 400 for at bestå. Det vil sige at resultatet kan svinge mellem +400 og -400 for at testen består. Vi tester også for om terningen slå højere end 6 eller mindre end 1, og hvis den gør det fejler den.

[6] CDIO del 2
https://docs.google.com/document/d/1npugHuQcEa7tSAv3d7KhBL3mh1j5qmquirQGs2cY3m8/edit#heading=h.65fze1prdg3f



Figur 19 : Søjlediagram med rollDice() testresultater.

Når man kaster en terning, må man gå ud fra at terningens 6 sider alle har en lige stor chance for at blive slået. Derfor ville sandsynlighed for et udfald på et kast med en terning være ⅙ = 0,167% . På Figur 17 kan vi se at teorien passer, dog med nogle enkelte udsvingninger på slag 1, 5 og 6, men det er så småt og er derfor acceptabelt, da det trods alt er sandsynligheder man regner med.

Test Cases

Test Case ID: TC01	
Item: Property	**Description**: A player purchases a property.
Title: Property purchase	Amount of the purchase (property): 4400 kr.
Priority: High	**Pass**: Yes
Initial configuration:	● A player lands on a field that has no owner yet. ● The land is ownable.
Software Configuration	macOS Big Sur (version 11.1) IntelliJ IDEA 2020.2.4 (Ultimate Edition) Project SDK: 1.8 (java version 1.8.0_271) Maven (Version: 3.6.3)
Steps	1. A player starts the game with 30,000 kr. in hand. 2. The player rolls the dice and lands on a field. 3. The player purchases the land and the player now has 25,600 kr. 4. The land now belongs to the player.
Expected behaviour	The player should be able to buy the land by paying the amount that requires.

Test Case ID: TC02	
Item: House	**Description**: A player pays the rent.
Title: Rent of house	**Amount of the rent**: 1 house 600 kr.
Priority: High	**Pass**: Yes
Initial configuration:	● A player lands on a field that has an owner ● The land has one or more houses to rent.
Software Configuration	macOS Big Sur (version 11.1) IntelliJ IDEA 2020.2.4 (Ultimate Edition) Project SDK: 1.8 (java version 1.8.0_271) Maven (Version: 3.6.3)
Steps	1. A player lands on “Valby Langgade” with 18,500 kr. in hand. 2. The rent of 1 house is taken from the player’s balance. 3. The player now has 17,900 kr.
Expected behaviour	Rent of the house should be taken away from the player that lands on the land by the system.

Test Case ID: TC03	
Item: Chance-cards	**Description**: Identification of the effects, when a player lands on a chance-card field
Title: Effects of the chance-card field	
Priority: High	**Pass**: Yes
Initial configuration:	● A player lands on a chance-card field.
Software Configuration	macOS Big Sur (version 11.1) IntelliJ IDEA 2020.2.4 (Ultimate Edition) Project SDK: 1.8 (java version 1.8.0_271) Maven (Version: 3.6.3)
Steps	1. A player lands a chance-card field 2. A message pops up “Ryk frem til Strandvejen”. 3. The OK button is pressed, and the player moves to “Strandvejen”.
Expected behaviour	As one can see that the expected result is obtained.

Vurdering af kvalitet

Spillet som det er nu, har ingen kendte game breaking bugs og fungerer, dog med færre funktionaliteter, end vi først havde forestillet os. Der ville dog kunne blive finpudset lidt på hvordan nogle af beskederne bliver vist. Eksempelvis når man prøver at købe et hus uden at have alle grundene i samme farvegruppe, bliver der vist to beskeder efter hinanden, selvom det ville være godt nok med én. De manglende funktionaliteter gør også at spillet bliver lidt mere kedeligt. Vi opdagede også først til sidst, at muligheden for at kunne sælge sin grund igen, var en ret vigtig del af kerne spillet, da det næsten bliver umuligt købe huse på en grund, hvis en anden spiller ejer en af de grunde med samme farve. Til gengæld er det godt vi fik implementeret chance kortene, da det giver lidt mere tilfældigheder i spillet, hvilket giver en anden dynamik til spillet, og gør at der kommer nye udfald hver gang man spiller spillet.

Projektplanlægning

Vi startede projektet med at at prioritere kundens kravspecifikation. Ud fra kravspecifikationerne fandt vi de vigtige use cases, og lavede use case diagrammet.
Da projektet ikke har ændret sig meget, har vi genbrugt nogle af vores use cases (fra CDIO del 3) og opdateret dem så de passer med vores nye krav.

Derefter påbegynde arbejdet på vores domænemodel, hvilket vi brugte til at opbygge vores klassediagram.

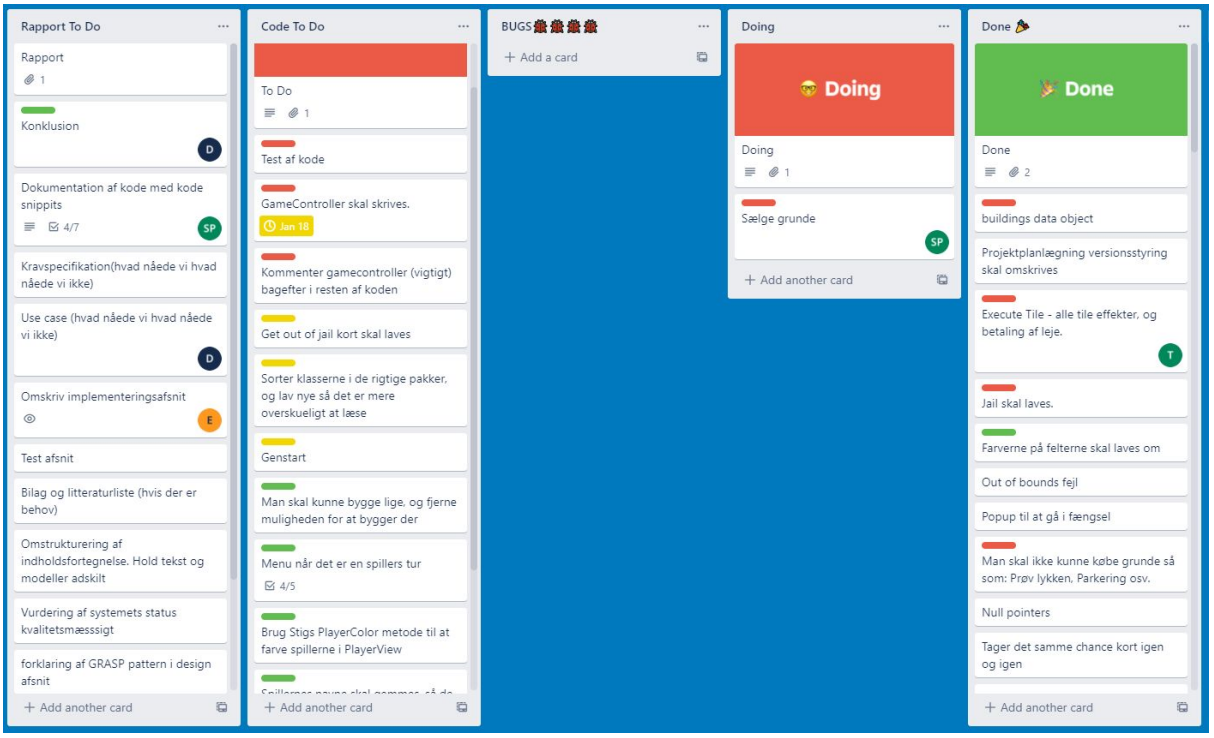
Vi genbrugte vores risici analyse fra vores CDIO del 3 opgave, da vi følte at der ikke var store ændringer i et aspekt. Selvfølgelig tog vi de samme forbehold og forebyggende på de områder vi mente havde højest risici for projektets fuldførelse.

Derefter lavede vi system sekvensdiagrammet for at se, hvordan aktørerne kommer til interagere med systemet.
Da vi nu havde fundet de klasser og metoder vi mente vi skulle bruge, følte vi at vi kunne gå i gang med implementeringen af use caserne.

Vi startede med at implementere de højt prioriteret use cases, så gik vi igange med mellem eller mindre prioriteret use cases. Kravspecifikationerne er også implementeret på det samme måde.
Vi uddelegerede arbejdsopgaverne rigeligt mellem gruppemedlemmerne og påbegynde implementeringen af vores kode.

Trello

Vi har brugt Trello til at holde styr på arbejdsopgaver og for at kunne skabe et overblik over hvem der arbejder på hvad på et givent tidspunkt. Det har gjort det nemmere at kunne fokusere på opgaven, når der har været tvivl om hvilke opgaver der skulle løses.



Figur 20 : Trello

I Trello bruger man lister til at sortere arbejdsopgaverne med. Vi har valgt at have 5 lister med henholdsvis "Rapport To Do", "Code To do", "Bugs", "Doing" og "Done". Dette er gjort, så man nemt kan se hvad der er i gang med at blive lavet, hvem der laver hvad og hvad der er færdigt. De forskellige opgaver er prioriteret med farvel labels. Rød er den mest vigtige og den grønne er den mindst vigtige. Farven på enkelte opgaver kan godt ændre sig afhængig af hvor meget der er blevet udviklet på koden, og om der er tid til at nå den pågældende opgave.

Versionsstyring[7]

I dette projekt har vi brugt Git i form af GitHub til versionsstyring af vores kode, og Google Docs til rapportskrivning og ligeledes holde versionsstyring af den.

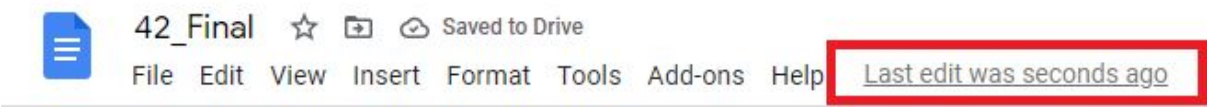
[7] CDIO del 3
https://docs.google.com/document/d/1dC61tPbIbAocAlL080VWjfUlORjD_ZSs9LUETMi-NyA/edit

GitHub

GitHub bruger vi til at holde styr på iterationerne af vores kode. Det tillader at arbejde agilt med koden og sørger for et nemt og forholdsvist problemfrit udviklingsmiljø. Vi har valgt at have en branching strategi, hvor vi har en main, developer og feature branches. I main skal der kun være fungerende kode, og der må ikke blive udviklet i main branchen. I developer er der hvor koden bliver udviklet, så der må gerne være kode, der er en smule ustabilt, men som udgangspunkt skal det fungere uden de store fejl. Feature branches er der hvor nye features bliver udviklet, her må koden gerne være ustabil, da det er ny kode der blive lavet.
Feature branches skal merges ind i developer, når koden er færdig i branchen. Her kan der opstå merge konfliker, de løses og der bliver rettet til i koden i developer branchen, så det fungerer. Når developer branchen er stabil og er klar til brug bliver den merget ind i main branchen, hvor det færdige produkt kommer til at ligge.

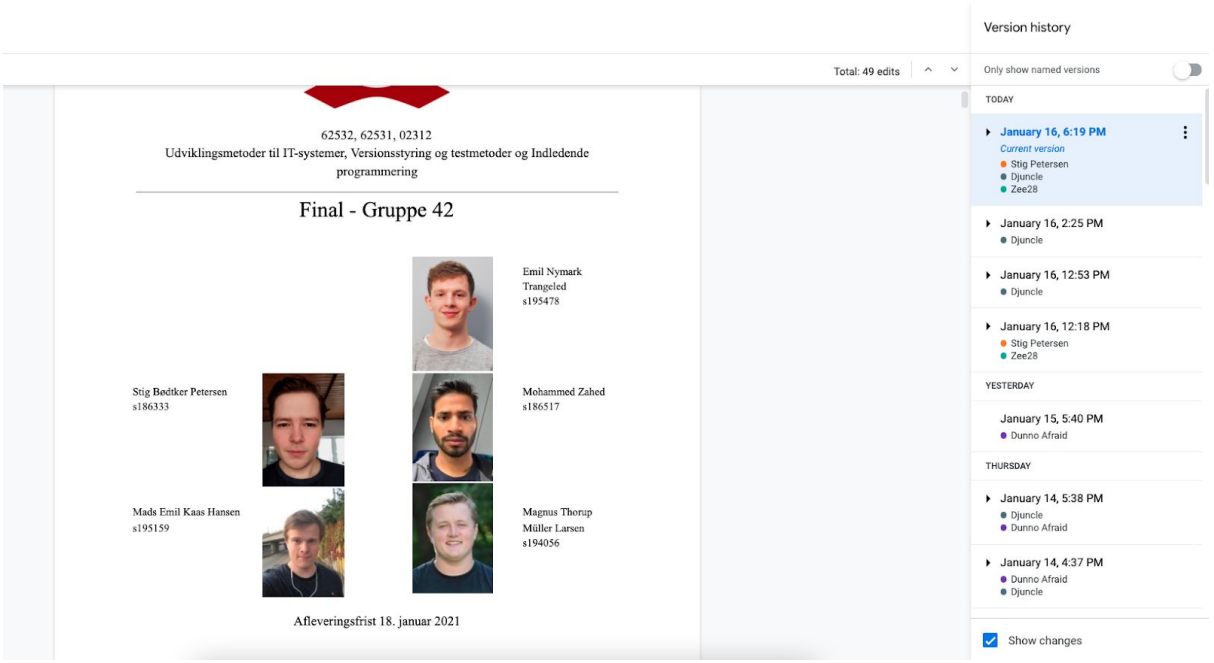
Google Docs

Versionsstyring af Google Docs, her har vi ikke sat nogle specielle regler for håndtering af versionsstyringen da den allerede tilbyder en flexibel arbejdsmetode. Man ville kunne finde Versions Historikken ved at klikke på knappen “Version history”:



Figur 21 : Version history

Dette ville tage en ind til historikken, hvor man kan se ændringerne ude i højre side, og reverte tilbage til en tidligere version, hvis der skulle være brug for det.



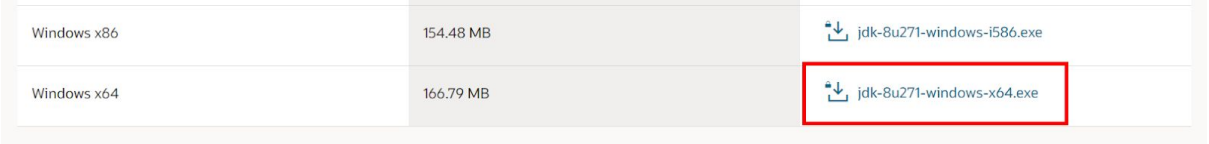
Figur 22 : Google Docs.

Konfigurationsstyring[8]

Udviklingsmiljø: Udviklingsmiljøet vi bruger til at udvikle projektet, er IntelliJ IDEA Ultimate Version 2020.3. Den kan downloades her.

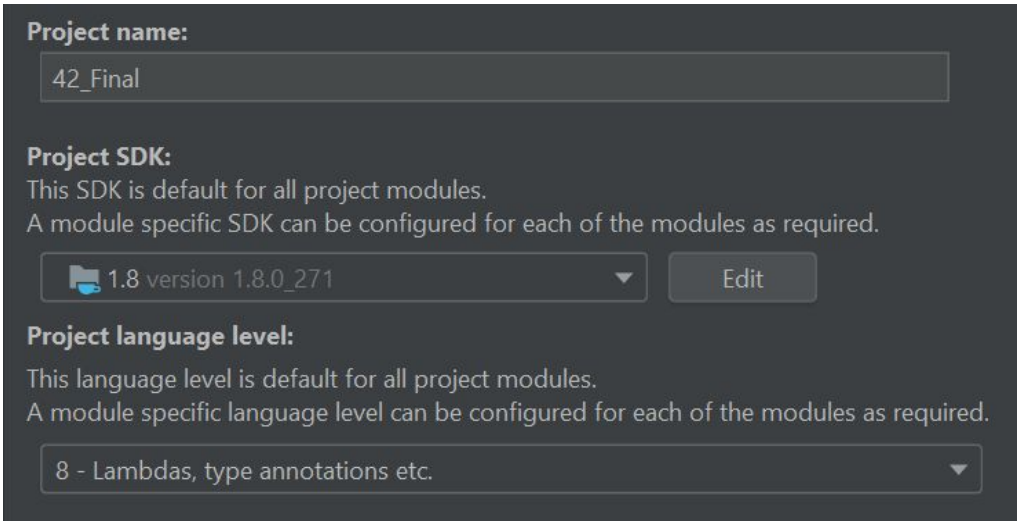
Figur 23 : Downloade Intellij

JDK version: Vi bruger Java JDK version 1.8.0_271 som vores projekt SDK og Language level 8 som vores projekt language level. Java SE Development Kit 8 kan downloades her.



Figur 24 : Java JDK

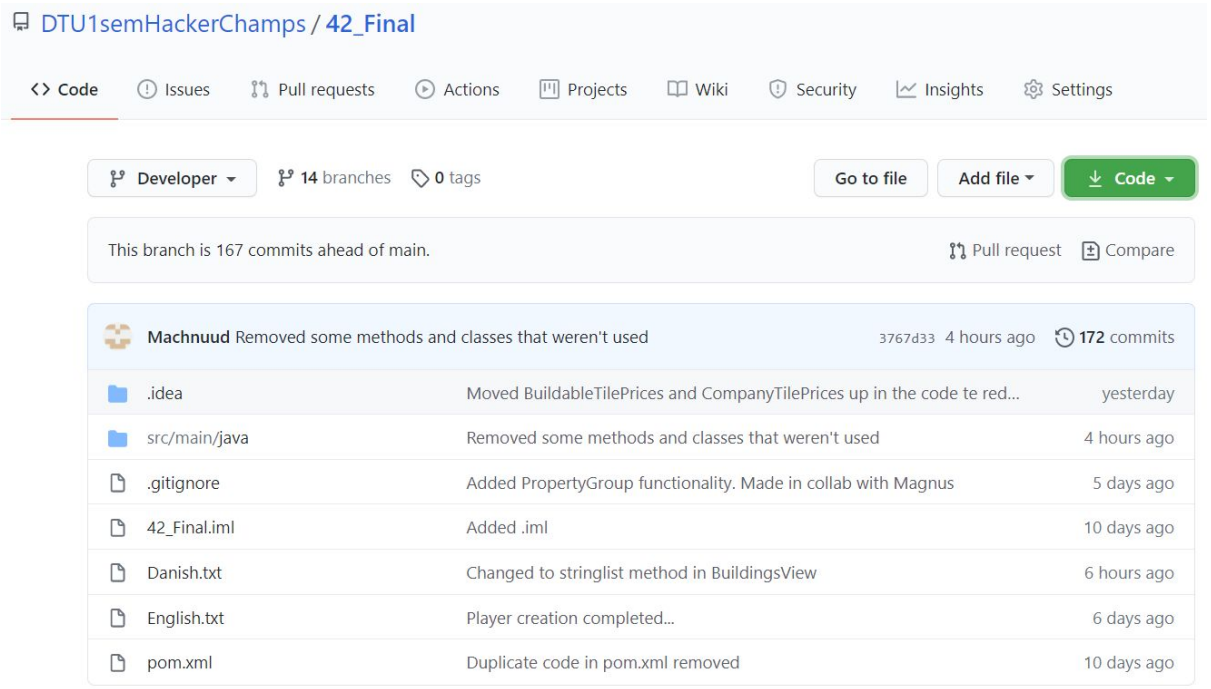
Inde under File > Project Structure i IntelliJ kan man så konfigurere Project SDK'en og Project Language Level hvilket vi har gjort her.



Figur 25 : Konfigurationsstyring.

[8] Gruppen har genbrugt store dele af konfigurationsstyringen fra CDIO 3 https://docs.google.com/document/d/1dC61tPbIbAocAlL080VWjfUlORjD_ZSs9LUETMi-NyA/edit#heading=h.n0k5idfcsjpt

Github: Til versionsstyring af programmet anvender vi Github. Link her: [DTU1semHackerChamps/42_Final (github.com)]



Figur 26 : Hent fra Github

Maven: For at gøre byggeprocessen af vores program nemmere anvender vi Maven, der blandt andet gemmer vores GUI artifacts. Til GUI'en vi har fået udleveret i undervisningen matadorgui.jar bruger vi altså dertil Maven til at gøre det nemmere at køre den.

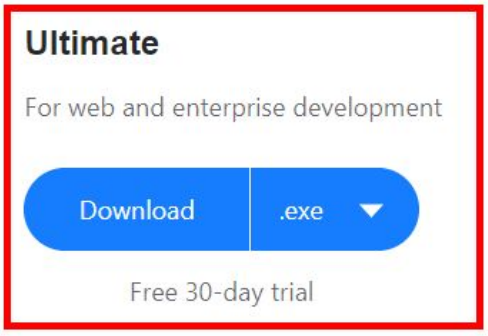
JUnit: Til tests af vores program, anvender vi JUnit 5.4.2. Man kan importere JUnit 5.4.2 direkte inde i IntelliJ ved at gå ind i Test, trykke F2 og Importere til IntelliJ.

Styresystem version: Programmet bliver skrevet på Windows 10 version 2004 build 19041.630.

Rapportskrivning: Til rapportskrivningen anvender gruppen Google Docs, da det er nemt, hurtigt og effektivt.

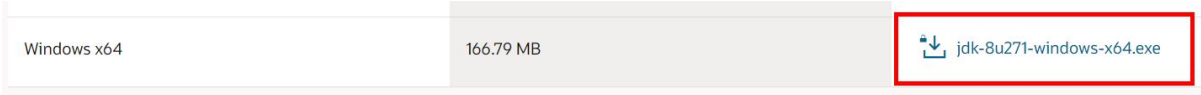
Brugervejledning[9]

1. Brugeren skal først downloade og installere IntelliJ IDE'en. Den kan downloades her.



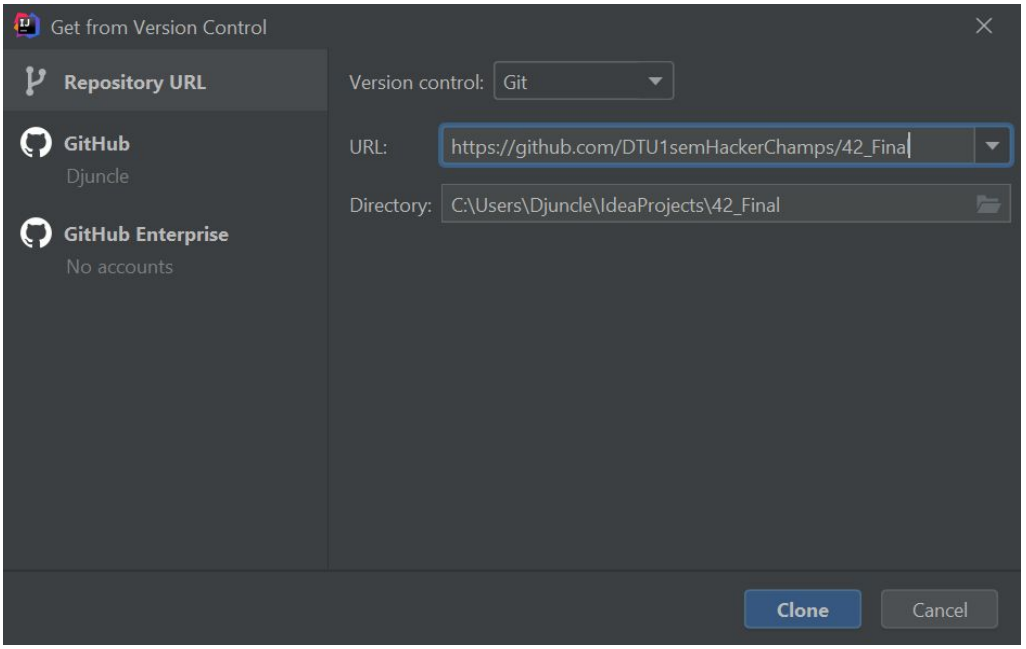
Figur 27 : installere IntelliJ IDE

2. Derefter skal brugeren downloade og installere en JDK. Vi bruger JDK 1.8. Den kan downloades her. Man skal være logget ind på Oracle.



Figur 28 : Installere en JDK

3. Brugeren downloader .zip filen fra Github repositoriet eller importere det direkte ind i IntelliJ ved at bruge File -> New -> Project from Version Control. Inde på URL boksen skal man så paste linket til Github repositoriet og derefter skal man trykke på "Clone".

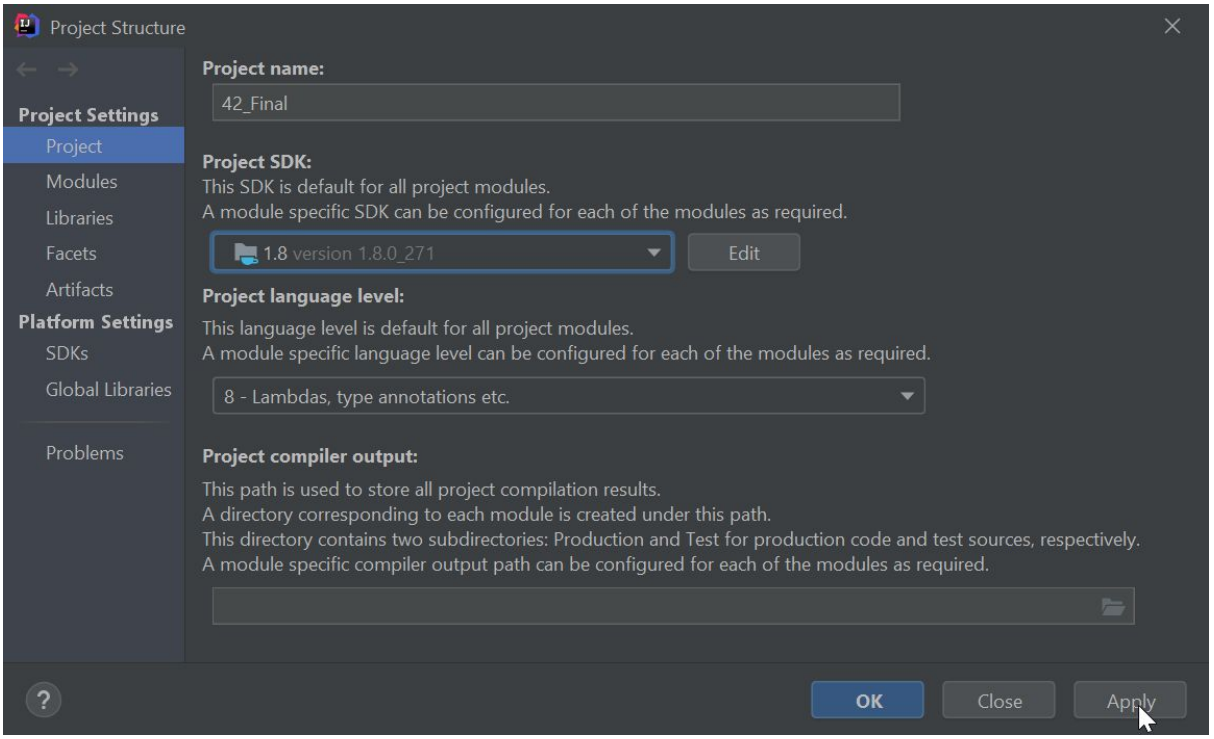


[9] Gruppen har genbrugt store dele af brugervejledningen fra CDIO 3 https://docs.google.com/document/d/1dC61tPbIbAocAIL080VWjfUIORjD_ZSs9LUETMi-NyA/edit#heading=h.n0k5idfcsjpt

Figur 29 : Get from version control

Gem den i et directory som du selv vælger. Her er den gemt i IdeaProjects mappen.

4. For at sikre sig at projektet virker skal man nu under File > Project Structure gå ind og ændre Project SDK'en til JDK 1.8 version 1.8.0_271 og Project language level til 8.

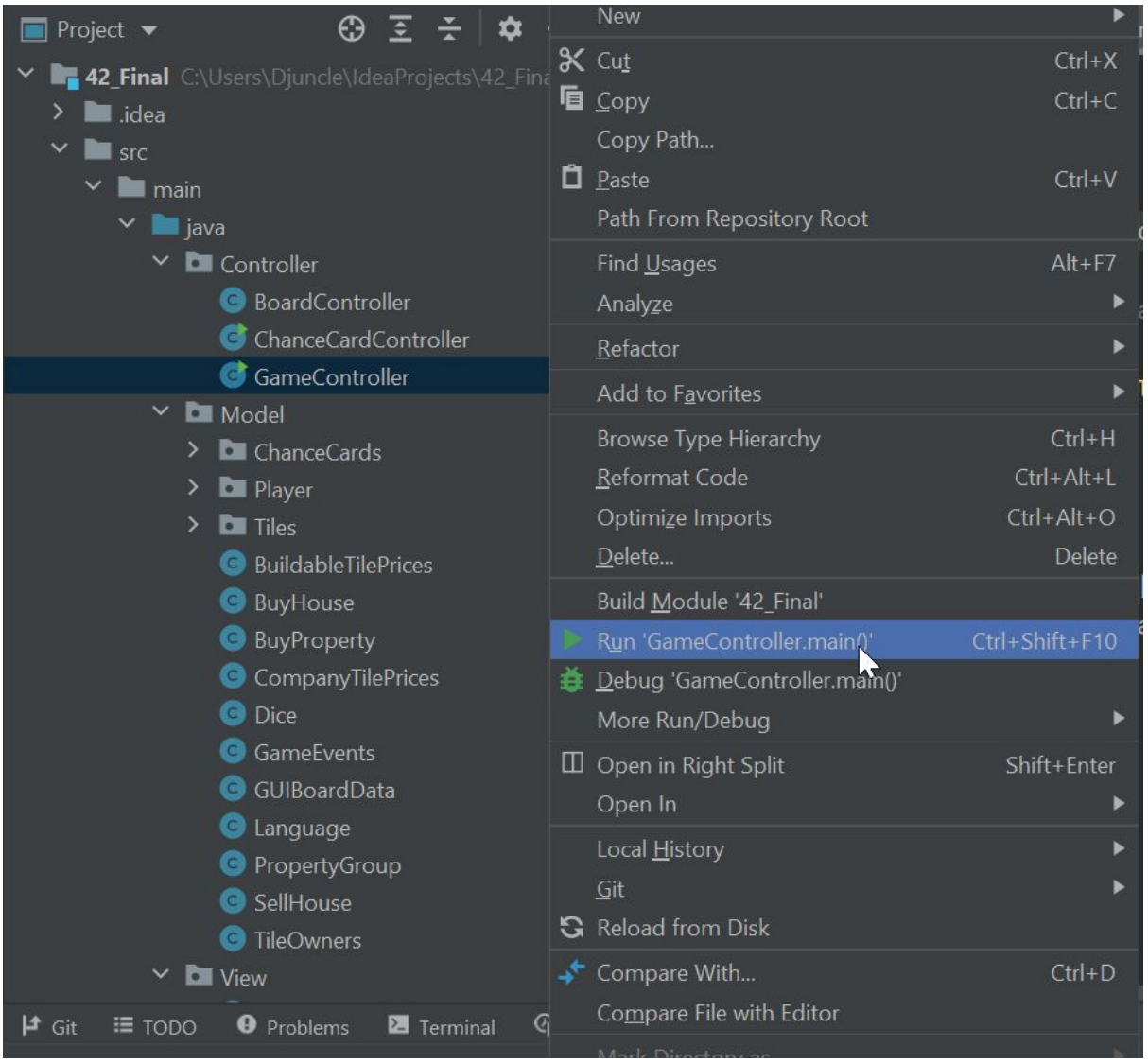


Figur 30 : SDK og language level

5. Hvis Junit ikke er importeret, skal man ind under test klasserne og importerne Junit.
6. Hvis GUI'en ikke virker, skal man vælge "matadorgui-3.1.6.jar" og tilføje den til ens library.
7. Nu skal man gå ind og reloade projektet så man kan få Maven til at fungere. Dette skal kun gøres en gang. Der er en detaljeret tutorial længere nede som forklare hvordan man gør.
8. Nu skal brugeren gå ned og højreklikke på main, trykke run program og køre det. Programmet starter derefter og brugeren kan begynde at spille.

Hvordan kører man programmet

Når man er inde i Intellij, har hentet programmet fra Github repositoriet og nu gerne vil køre programmet, skal man finde klassen der hedder "GameController", højreklikke på den og klikke Run 'GameController.main():

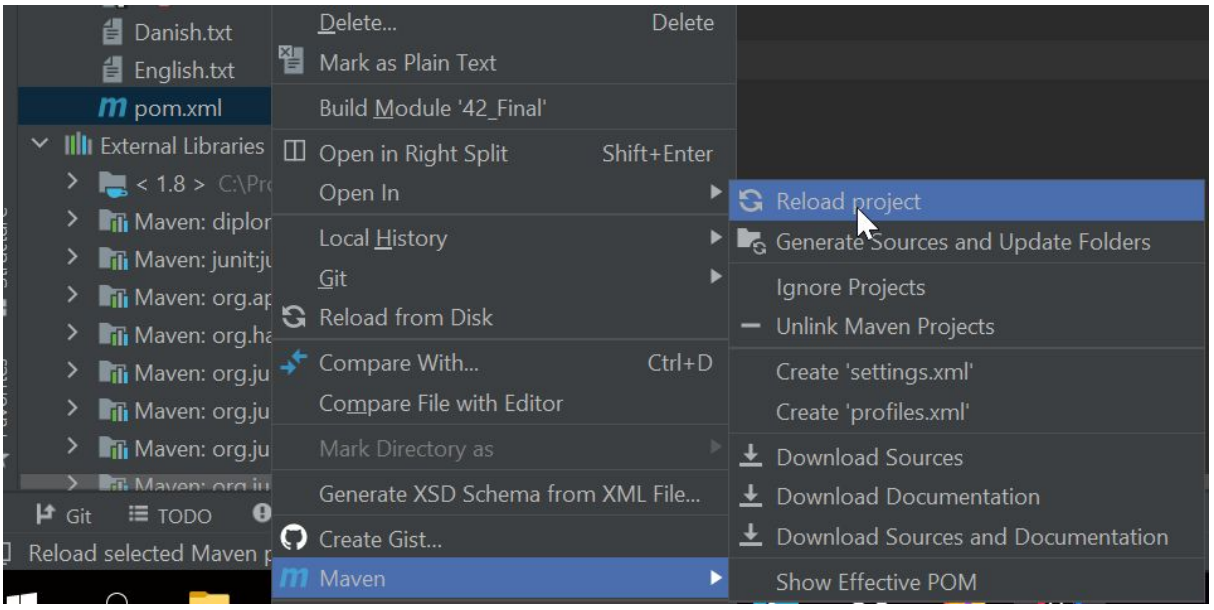


Figur 31 : Kør programmet

Nu skulle programmet køre og man er nu klar til at spille.

Reload af pom.xml fil

Der kan være brug for at reloade pom.xml når man først har loaded projektet ind fra github. For at reloade pom.xml filen skal man finde pom.xml og højreklikke på den, så kommer der en dropdown menu og man navigere ned til Maven > Reload project og klikke på Reload project:



Figur 32 : Maven

Så skulle pom.xml filen være reloaded.

Konklusion

Gruppen kan konkluderet at de har formået at udvikle et Matador spil i Java, der ligner et rigtigt matador spil, dog med nogle centrale dele som er udgået fra spillet. En af disse dele er en auktions feature, som skulle bruges når en spiller vælger ikke at købe en grund, når de lander på en ledig grund. En anden feature der er udgået, er salg af grunde til andre spillere. Muligheden for at spille et hurtigt spil er også en feature som er udgået. Både auktions featuren og hurtig spil featuren var oprindelig ikke med i vores scope af spillet, men ville helt klart blive implementeret, hvis vi fortsatte med at arbejde videre på programmet i fremtiden. Salg af grunde til andre spillere var dog med i det originale scope for vores spil, men featuren endte med at udgå, for at sikre funktionaliteten af andre dele af spillet fortsat ville være fuldt implementeret, da gruppen var under tidspres.

Gruppens tidsplan er under projektet løbende blevet ændret, efter at forholdene ændrede sig. Både når en features blev udviklet til tiden eller blev forsinket. Gruppen mistede også et gruppemedlem midtvejs i processen, hvilket gjorde at gruppen skulle lave en stor omstrukturering af den planlagte tidsplan.

Bilag

Sekvensdiagram:
https://drive.google.com/file/d/1XT_F72Z2ToUkWj92Q9RE9RBsi_xcNAyp/view?usp=sharing

System sekvensdiagram:
https://drive.google.com/file/d/1MuzlN0-dTd3_z5FilQb6QtxXthU5QyeD/view?usp=sharing

Domænemodel:
https://drive.google.com/file/d/1iWz8Mm1G11E6ELEWtW-yI06ty57pnX4a/view?usp=sharing

Design klassediagram:
https://drive.google.com/file/d/1naEGqHgEEIFWSI-D_LQK4E8NsyQpOUVp/view?usp=sharing

Indholdsfortegnelse for code snippets

View→BoardView linje 82-91

Figur 12 : BankruptPlayers klasse i Model.Player package:
Model→Player→BankruptPlayers linje 3-40

Figur 13 : switchPlayer() Metode i Player klassen:
Model→Player→Player linje 125-134

Figur 14 : whoWon() metode i GameEvents klassen:
Model→GameEvents linje 10-24

Figur 15 : buyProperty() metode i BuyProperty klassen:
Model→BuyProperty linje 16-32

Figur 16 : buyPropertyView() metode i BoardView klassen:
View→BoardView linje 93-99

Figur 17: Test af kort blandings metoden cardShuffle()
Test→ ShuffelMechanicTest linje 13-51

Figur 18: Test af tilfældigheden af metoden rollDice()
Test→ DiceTest linje 11-50

Litteraturliste

CDIO 3
https://docs.google.com/document/d/1dC61tPbIbAocAlL080VWjfUlORjD_ZSs9LUETMi-NyA/edit#heading=h.n0k5idfcsjpt
Nyborg, Mads; (2. okt 2020) Tavle noter fra indledende programmering:
https://docs.google.com/document/d/1nhgljX_WCB6znKg1sxO3x5EeXiGf-2HLcwrOqbmebHs/edit#heading=h.xnrqmx5f0nlv

Nyborg, Mads; (2. okt 2020) GUI:
https://drive.google.com/drive/folders/0B1qlt-Xd6kaQZTdVb0hWdEt2eWM