
CDIO 3

INDLEDENDE PROGRAMMERING (02312), UDVIKLINGSMETODER FOR IT-SYSTEMER (62531) OG VERSIONSSTYRING OG TESTMETODER (62532)

Troels Christoffersen
s152780



Tobias Maneschijn
s205422

Marcus Kjærsgaard
s205436



Hildbjørg Didriksen
s164539

Frederik Lundsbjerg
s205472



Rasmus Nylander
s205418

Gülsen Tasçi
s194612



GRUPPE 21

Afleveringsfrist : 27. November - 2020

Abstract

IOOuteractive was tasked with developing a Monopoly Junior game in which the customer had given a specific set of requirements. In essence the customer wanted to build upon a previous board game project, and furthermore add some functionalities. These added functionalities expands on the complexity of the previous board game. And as it turns out, almost exponentially increases both the effort the group has to deliver, and the difficulty in creating and implementing the monopoly jr. game the customer wants.

Given this project is that of a greater size, the customer so kindly gave us a more elaborate set of requirements, which with some minor modifying, in coorporation with the customer, turned out to more or less be a recipe for cooking up this monopoly jr. game.

Of course we wanted to deliver on every single requirement, but as previously stated the difficulty of this project, and the time frame given to deliver, exceeded what the group could handle. We ended up with a somewhat working game, and delivered on most of the requirements. The requirements which were not fulfilled were described, and solutions to these were proposed.

Overall the group has delivered a somewhat satisfying result, and even more importantly, learned a big lesson and gained some useful experiences.

Timeregnskab og Ansvarsfordeling

Timeregnskab

Timeregnskabet kan findes på følgende hjemmeside:

[https://docs.google.com/spreadsheets/d/
1ywkS7sP8V0ca2PdL3H3YJop3QzlPSc65H7xAewZvKgQ/edit?usp=drive_web&
ouid=103225916642794909878](https://docs.google.com/spreadsheets/d/1ywkS7sP8V0ca2PdL3H3YJop3QzlPSc65H7xAewZvKgQ/edit?usp=drive_web&ouid=103225916642794909878)

Ansvarsfordeling

Abstract	Troels
Indledning	Hildibjørg & Marcus
Krav	Troels, Hildibjørg & Rasmus
Analyse	Alle
Design	Alle
Implementering	Alle
Test	Marcus, Hildibjørg & Tobias
Projektplanlægning	Marcus
Konfiguration og Versionstyring	Troels
Konklusion	Frederik

Indhold

1	Indledning	5
1.1	Spillet	5
1.2	Spillets felter	5
1.3	Udarbejdelse af projekt	7
2	Krav	8
3	Analyse	12
3.1	Aktør	12
3.2	Use Case Diagram	12
3.3	Use Case tabeloversigt	13
3.4	Use Case beskrivelser	14
3.5	Domænemodel	19
3.6	Systemsekvensdiagram	20
4	Design	21
4.1	GRASP	21
4.2	Designklassediagram	21
4.2.1	GRASP principper for designklassediagram	23
4.2.2	Abstrakt, nedarvning og polyformi	24
4.3	Sekvensdiagram	24
5	Implementering	26
5.1	Singletons	26
5.2	OurArrayList	27
5.3	XML	27
5.4	Oversættelse med <i>Localisation</i>	29
5.5	Board	29
5.6	BoardLoader	29
5.7	Square	29
5.8	BankBalance	30
5.9	Die	31
5.10	Game	31
5.11	GUIWrapper	32
5.12	GUIManager	32
5.13	ChanceCard	32
5.14	Player	35
5.15	Deck	36
6	Test	37
6.1	DTU maskine og responstid test	37
6.2	Bugtest	39
6.3	JUnit tests	39

6.4	Test af terning	40
6.5	Formelle Testcases	40
6.6	Code coverage	43
7	Projektplanlægning	45
7.1	Udviklingsproces for forløb	45
7.2	Planlagte forløb	45
7.3	Faktiske forløb	46
8	Konfiguration og Versionstyring	49
9	Import af Git Rep og Start Spil	51
9.1	A Beginners guide to opening Monoploy Jr.	51
10	Konklusion	53
11	Bilag	54
11.1	Link til GitHub	54

1 Indledning

Programmørerne i IOOuteractive er blevet bedt om at lave et Monopoly Junior spil. Spillet er en nemmere udgave af Monopoly, som er egnet til børn 5 år og ældre. Kunden har stillet specifikke krav til hvordan spillet skal foregå og hvor vi bl.a. har fået tildelt en GUI, som der skal gøres brug af i programmet.

1.1 Spillet

Spillet og projektets formål er, at kunne opstille et brætspil, som kan bestilles af 2-4 spillere, som hver har én spillebrik af henholdsvis figurerne: Hund, Kat, Skib og Bil. Spillerne slår med 1 terning og rykker rundt på spillepladen. Spillerne skal så købe ejendomme på spillebrættet og tjene Å penge. Dette fortsætter indtil, at der er én spiller der går fallit. Derefter er der i spillets simple form, så spilleren der har den største pengebeholdning der vinder spillet. I spillets avancerede version, vil der derudover være mulighed for, at kunne betale tilbage i ejendomme, så man ikke nødvendigvis går fallit. Fælles for begge versioner er, at der i det tilfælde, hvor der er uafgjort i pengebeholdningen, mulighed for at tælle ejendommene med i den samlede værdi.

1.2 Spillets felter

Spillet består af forskellige felter der har forskellige funktioner.

Startfelt:

Spillebrættet har et startfelt, hvor alle spillere starter fra når spillet påbegyndes. Efterfølgende vil startfeltets funktion være at hver gang en spiller lander eller passerer fletet modtager spilleren 2 Å.

Fængselsfelter:

Der er 2 felter i spillet der har med fængslet at gøre: *Gå-I-Fængsel-feltet* og *På-Besøg-I-Fængslet-feltet*. Hvis man lander på *Gå-I-Fængsel-feltet* skal spilleren på fletet sættes i fængsel. Selve fængslet ligger på fletet *På-Besøg-I-Fængslet-feltet*, spilleren flyttes hertil og hvor der derudover er tilhørende regler for, hvordan man kommer ud igen. Modsat når en spiller lander på *På-Besøg-I-Fængslet-feltet* sker der ingenting.

Chance-felter

Der er 4 chancefelter i spillet og hvis en spiller lander på et af disse, trækkes der et chancekort fra chancekort bunken og så skal spilleren følge de instrukser der står på kortet.

Gratis Parkering

Hvis en spiller lander på Gratis Parkeringsfeltet så sker der ikke noget og spillerens tur afsluttes.

Ejendomsfelter:

Ejendomsfelter kan købes af spillerne hvis de lander på det og det ikke har en ejer i forvejen. Hvis en spiller lander på et ejet felt, så skal den spiller betale husleje til ejeren af feltet. Hvert felt har en farve og hvis en spiller ejer alle ejendomme i samme farve, så skal de andre spillere der lander på feltet betale ejeren det dobbelte i husleje. Her ses en liste over alle ejendomsfelter samt deres tilhørende farve og pris:

- Burgerbaren (Brun) 1 ÅÅ
- Pizzeriaet (Brun) 1 ÅÅ
- Slikbutikken (Lyseblå) 1 ÅÅ
- Iskiosken (Lyseblå) 1 ÅÅ
- Museet (Lyserød) 2 ÅÅ
- Biblioteket (Lyserød) 2 ÅÅ
- Skaterparken (Orange) 2 ÅÅ
- Swimmingpoolen (Orange) 2 ÅÅ
- Spillehallen (Rød) 3 ÅÅ
- Biografen (Rød) 3 ÅÅ
- Legetøjsbutikken (Gul) 3 ÅÅ
- Dyrehandlen (Gul) 3 ÅÅ
- Bowlinghallen (Grøn) 4 ÅÅ
- Zoo (Grøn) 4 ÅÅ
- Vandlandet (Mørkeblå) 5 ÅÅ
- Strandpromenaden (Mørkeblå) 5 ÅÅ

1.3 Udarbejdelse af projekt

Til udarbejdelse af vores projekt vil vi gøre brug af Unified Process som vores udviklingsproces. Her vil der tages udgangspunkt i at arbejde henover en 3-ugers periode, hvor projektet opdeles i mindre delelementer. Vha. Unfied Process gør vi brug af Unified Modelling Language, hvor der vil udarbejdes diverse artifacts til at kunne udvikle og dokumentere vores softwareprojekt. Derudover vil der gøres brug af diverse JUnit-, sandsyndigheds- og brugertest til at kunne sikre os en tilfredsstillende funktionalitet af programmet. Til sidst vil der blive beskrevet hvordan projektet har forløbet sig både ift. planlægning men også strategi for versionsstyring i Github og konfigurationsstyring. Her vil der blive beskrevet hvilke udfordringer der har været undervejs, og hvad gruppen har taget ved læре fra forløbet. Derudover vil der afrundningsvis vurderes om hvorvidt kravspecifikationen er afdækket på tilfredsstillende vis.

2 Krav

Regler

1. Spillet spilles af 2-4 spillere
2. Spillet skal indeholde følgende
 - 1 Spillebræt
 - 4 Brikker
 - 20 Chancekort
 - 1 terning
3. Hver spiller skal kunne vælge deres egen figur i starten af spillet
4. Spillet agerer selv som bank

Spillets gang

5. Den yngste spiller starter
6. ÅA penge bliver i starten af spillet fordelt efter antal spillere:
 - Ved 2 spillere: 20 ÅA penge hver
 - Ved 3 spillere: 18 ÅA penge hver
 - Ved 4 spillere: 16 ÅA penge hver
7. En runde starter ved at kaste terningen og derefter rykker spillet ens figur med uret på brættet. Herefter går turen videre i urets retning.
8. Hver gang start passeres modtages der med det samme 2 ÅA penge
9. Landes der på et ledigt felt. Skal spilleren købe det. (beløb står på feltet)
 - Pengene går til banken
 - Feltet bliver spillerens farve
10. Landes der på et ejet felt (beløb står på feltet)
 - Skal husleje betales
 - Hvis du selv ejer grunden sker der ikke noget

11. Hvis en spiller ejer begge ejendomme i samme farve skal der betales dobbelt husleje
12. Spillets skal indeholde følgende typer af felter
 - Start
 - Chance
 - Gå i fængsel
 - På besøg i fængsel
 - Gratis Parkering
13. I starten af spillet skal der kunne vælges mellem følgende 2 spilleformer
 - Simpel
 - Hvis spilleren ikke har råd til at betale for husleje, eller betale afgift fra et chancekort, går spilleren fallit. Og så er spillet slut
 - De andre spillere tæller deres penge, og den, der har flest har vundet!
 - Uafgjort? Spillerne med lige mange penge tæller deres ejendomsværdi, højeste ejendomsværdi vinder.
 - Avanceret
 - Hvis du ikke har penge nok til at betale husleje eller en afgift fra et chancekort, skal du betale med dine ejendomme.
 - Hvis du skylder en anden spiller penge, får den spiller dine ejendomme. Hvis du skylder banken penge, bliver dine ejendomme sat til salg igen.
 - I tilfælde af at spilleren skal betale med ejendomme, skal spilleren selv vælge hvilke ejendomme der bruges som betaling
 - Hvis du stadig ikke kan betale, er du gået fallit, og spillet slutter efter al den fallittes spillers værdi er overdraget. Alle tæller deres penge for at se hvem der har vundet.
 - Uafgjort? Spillerne med lige mange penge tæller deres ejendomsværdi, højeste ejendomsværdi vinder.

Analyse- og designdokumentation

14. Kravliste
15. Use case diagram
16. Eksempler på use case beskrivelser - vælg mindst én, der beskrives fully dressed
17. Domænemodel
18. Et eksempel på systemsekvensdiagram

19. Et eksempel på sekvensdiagram

20. Designklassediagram

Implementering

21. Der skal gøres brug af UTF-8

22. Klasser der skal initialiseres skal have konstruktører

23. Lav get og set metoder hvor det giver mening

24. Lav `toString` metoder hvor det er relevant

25. Lav en klasse `gameboard` der kan indeholde alle felterne i et array

26. Tilføj en `toString` metode der udstriber alle felterne i arrayet

27. Udvikl videre på de klasser i har fra CDIO 2

28. Benyt GUI'en. Gui' skal importeres fra Maven: Maven repository

29. Tekst strenge skal så vidt muligt importeres fra filer

Dokumentation

30. Forklar hvad `arv` er

31. Forklar hvad `abstract` betyder

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

33. Dokumentation for test med screenshots

34. Dokumentation for overholdelse af GRASP

Test

35. Lav mindst tre testcases med tilhørende fremgangsmåde/testprocedurer og testrapporter

36. Lav mindst én Junit test til centrale metoder. Inkludér code coverage dokumentation

37. Lav mindst én brugertest. Husk at brugeren skal være en der ikke kan kode

38. Spillet skal kunne spilles på en maskine på en af DTU's databarer

39. Der må max være forsinkelser på 1 sekund

Versionstyring

40. Lav et lokal Git-repository som en del af IntelliJ-projektet. Alternativt kan afleveres et link til et repository på nettet eks.
41. Rapporten skal indeholde en vejledning i hvordan man importerer Git-repository i IntelliJ.

Konfigurationsstyring

42. Dokumenter platformens dele med versionsnummer så de kan genskabes til senere brug
43. Definer hvordan der sikres at der på ethvert tidspunkt vil kunne finde den sidste nyeste version af samtlige artefakter.
44. Definer hvordan der sikres at dokumentation altid er opdateret for systemet
45. Beskrive hvor filerne er
46. Hvordan sikres at versionen er opdateret?
47. Hvordan findes versioner som passer sammen?
48. Beskriv hvordan projektet importeres i IntelliJ fra git.
49. Beskriv hvordan man kører programmet
50. Der skal benyttes Maven til at hente junit-

3 Analyse

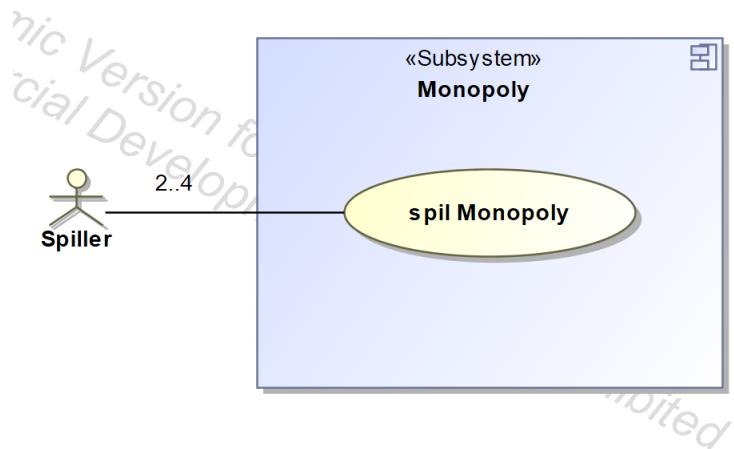
I analysefasen af projektet vil der tages brug af de funktionelle og ikke-funktionelle krav til brætspillet. Ud fra dette vil vi udarbejde et use case diagram, samt en tabeloversigt over use-cases med ID-numre, funktion og casual beskrivelse. Der vil derudover også blive udarbejdet use case beskrivelser udfra use case tabellen, hvor der henholdsvis vil gøres brug af detaljeret(fully-dressed) format og hovedsageligt brief form. Til sidst vil der udarbejdes en domænemodel og et sekvensdiagram. Domænediagrammet indeholder koncepter af klasser og systemsekvensdiagrammet er nødvendigt for at få et dynamisk overblik over systemet for, at kunne designe koden for brætspillet.

3.1 Aktør

Brætspillet indeholder én primær aktør:

- Spiller

3.2 Use Case Diagram



Figur 1: Use-Case Diagram

3.3 Use Case tabeloversigt

ID	Funktion	Casual beskrivelse
UC01	Spil Monopoly	2-4 deltagere spiller på skift indtil der er fundet en vinder
UC02	Kast Terning	En spiller kaster en terning og får givet en værdi i antal øjne på terningen
UC03	Ryk brik	En spiller har kastet en terning og rykker til et nyt felt i urets retning
UC04	Passeret start	En spiller lander eller har passeret start-feltet og modtager 2 ÅA penge
UC05	Lander på et ejendomsfelt	En spiller køber et felt og placerer et <i>solgt</i> -skilt
UC06	Lander på Chance feltet	En spiller tager et chancekort fra toppen af bunken, udfører en instruks og kortet lægges nedest i bunken igen.
UC07	Fængsel feltet	En spiller lander på <i>gå i fængsel</i> -feltet og rykker til <i>fængsel</i> -feltet og kommer ud igen
UC08	Vind spil	En spiller findes når en af spillerne er gået fallit
UC09	Gæld	En spiller betaler med et af sine ejendomme, hvis spilleren ikke har råd til at købe et felt

Tabel 1: Use case tabeloversigt

3.4 Use Case beskrivelser

Detaljeret beskrivelse:

Use Case	Lander på et ejendomsfelt
ID	UC05
Scope	Monopoly junior
Level	N/A
Primary actor	Spiller
Stakeholders and Interests	N/A
Preconditions	At spilleren har påbegyndt brætspillet, det er spillerens tur og spiller skal have rykket til et ejendomsfelt
Post condition	Spiller ejer enten feltet, har betalt husleje eller er gået fallit
Main Success Scenario	<ol style="list-style-type: none"> 1. Spiller rykker til ejendomsfelt 2. Spiller køber feltet og betaler banken 3. Spiller placerer et <i>solt-skilt</i> på feltet 4. Spiller ejer nu feltet
Extensions (alternative flows)	<ol style="list-style-type: none"> a. Hvis ingen ejer feltet <ol style="list-style-type: none"> 2. Spiller køber felt og placerer <i>solt-skilt</i> b. Hvis spiller selv ejer feltet <ol style="list-style-type: none"> 1. Turen afsluttes c. Hvis en anden spiller ejer feltet <ol style="list-style-type: none"> 1. Hvis en anden spiller ejer begge felter i samme farve <ol style="list-style-type: none"> 1.1 Spiller betaler dobbelt i husleje, til den spiller som ejer ejendommen, eller går fallit 2. Spiller betaler husleje, til den spiller som ejer ejendommen, eller går fallit d. Runden afsluttes
Special Requirements	N/A
Technology and Data Variations List	N/A
Frequency of Occurrence	Der er høj frekvens, da brætspillet hovedsagligt består i at købe og eje ejendomme

Tabel 2: UC05 Lander på et ejendomsfelt

Brief beskrivelse:

Use Case	Spil Monopoly
ID	UC01
Brief Description	2 til 4 spillere påbegynder brætspillet. Spillet spilles og der findes en vinder
Primary actors	Spiller
Secondary actors	N/A
Preconditions	At man åbner Monopoly Junior spillet på sin computer
Main flow	<ol style="list-style-type: none"> 1. Hver spiller indtaster sit navn og alder 2. Hver spiller vælger én figur 3. Spillet spilles 4. Der findes en vinder
Post conditions	Der er fundet en vinder af spillet
Alternative flows	N/A

Tabel 3: UC01 Spil Monopoly

Use Case	Kast terning
ID	UC02
Brief Description	En spiller kaster en terning og får givet en værdi i antal øjne på terningen
Primary actors	Spiller
Secondary actors	N/A
Preconditions	At spilleren har påbegyndt brætspillet og det er spillers tur
Main flow	<ol style="list-style-type: none"> 1. Når det er spillerens tur kastes en terning 2. Spilleren kan aflæse antal øjne på terningen
Post conditions	Spilleren rykkes det antal felter frem, som terningens værdi viser
Alternative flows	N/A

Tabel 4: UC02 Kast terning

Use Case	Ryk brik
ID	UC03
Brief Description	En spiller har kastet en terning og rykker til et nyt felt i urets retning
Primary actors	Spiller.
Secondary actors	N/A
Preconditions	At spilleren har påbegyndt brætspillet og det er spillers tur
Main flow	<ol style="list-style-type: none"> 1. Use case starter med spiller har slået med terningen. 2. Spiller aflæser antal øjne på terningen 3. Spiller rykker ternings værdi frem i felter (i urets retning)
Post conditions	Spilleren er rykket til et nyt felt
Alternative flows	<ol style="list-style-type: none"> 1. Hvis spiller rykker felt pga. konsekvens fra chancekort 2. Hvis spiller rykker til fængsel pga. <i>Gå-I-Fængsel-feltet</i>

Tabel 5: UC03 Ryk brik

Use Case	Passeret start
ID	UC04
Brief Description	En spiller lander eller har passeret <i>Start-feltet</i> og modtager 2 Å
Primary actors	Spiller
Secondary actors	N/A
Preconditions	At det er spillers tur og spiller har rykket til nyt felt
Main flow	<ol style="list-style-type: none"> 1. Spiller rykker til nyt felt 2. Spiller lander på eller passerer <i>Start-feltet</i> 3. Spiller modtager 2 Å
Post conditions	Spiller har modtaget 2 Å, hvis ikke spiller er i fængsel
Alternative flows	<ol style="list-style-type: none"> 1. Hvis spiller rykker til fængsel pga. <i>Gå-I-Fængsel-feltet</i> og modtager spiller ikke 2 Å

Tabel 6: UC04 Passeret start

Use Case	Lander på <i>Chance</i> -felt
ID	UC06
Brief Description	En spiller tager et chancekort fra toppen af bunken, udfører en instruks og kortet lægges nederst i bunken igen.
Primary actors	Spiller
Secondary actors	N/A
Preconditions	At spilleren er landet på et <i>chance</i> -felt
Main flow	<ol style="list-style-type: none"> 1. Spiller er landet på et <i>chance</i>-felt 2. Spiller tager det øverste kort i chancekortbunken 3. Spiller udfører instruks på kortet 4. Spiller lægger det brugte kort nederst i bunken.
Post conditions	Spiller har trukket et chancekort, udført handling og lagt kort tilbage i bunken.
Alternative flows	<ol style="list-style-type: none"> 1. Hvis spiller har modtaget et <i>Du løslades uden omkostninger</i>-kort <ol style="list-style-type: none"> 1.1 Spiller gemmer kort til vedkommende er i fængsel 2. Hvis spiller har givet et <i>"Giv dette kort til.."</i>-kort til anden spiller <ol style="list-style-type: none"> 2.1 Den pågældende spiller, som har modtaget kortet, gemmer kortet til det er vedkommendes tur og udfører aktiviteten i deres tur

Tabel 7: UC06 Lander på *chance*-felt

Use Case	Fængsel feltet
ID	UC07
Brief Description	Spilleren skal gå i fængsel og kommer ud igen
Primary actors	Spiller
Secondary actors	N/A
Preconditions	At spilleren er landet på <i>Gå-I-Fængsel</i> -feltet
Main flow	<ol style="list-style-type: none"> 1. Use case starter med spiller har slået terningen 2. Spilleren har slået det antal øjne hvor spilleren lander på <i>Gå-I-Fængsel</i>-feltet 3. Spilleren skal nu flyttes til <i>I-Fængsel</i>-feltet 4. Spilleren skal nu vente én runde 5. Spilleren betaler 1 Å for at komme ud af fængslet
Post conditions	Spilleren kommer ud af fængslet igen
Alternative flows	<ol style="list-style-type: none"> 1. Hvis spilleren har chancekortet <i>Du løslades uden omkostninger</i> så må spilleren godt komme ud uden at skulle betale 1 Å 2. Hvis spilleren ikke har nok penge til at betale for at komme ud går spilleren fallit

Tabel 8: UC07 Fængsel feltet

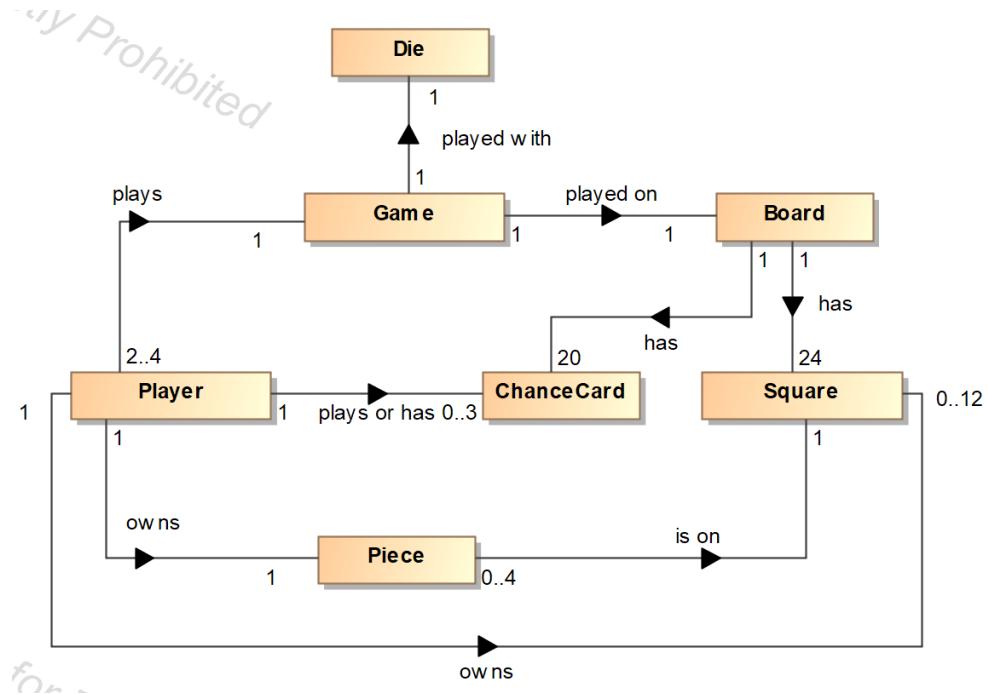
Use Case	Vind spil
ID	UC08
Brief Description	Spilleren skal gå i fængsel og kommer ud igen
Primary actor	Spiller
Secondary actor	N/A
Preconditions	Der er en spiller, som er gået fallit.
Main flow	<ol style="list-style-type: none"> 1. En spiller går fallit 2. Den spiller med flest penge vinder 3. Spillet er afsluttet
Post condition	Der er fundet en vinder af spillet
Extensions (alternative flows)	<p>a. Hvis optælling er uafgjort</p> <ol style="list-style-type: none"> 1. Læg værdi af ejendomme til optællingen b. Den spiller med højst samlet værdi vinder

Tabel 9: UC08 Vind spil

Use Case	Gæld
ID	UC09
Brief Description	Hvis spiller ikke har penge nok til at betale hvad de skylder så skal spilleren betale gælden med sine ejendomme
Primary actors	Spiller
Secondary actors	N/A
Preconditions	At spiller ingen penge har
Main flow	<ol style="list-style-type: none"> 1. Spiller rykker hen på et ejendomsfelt ejet af en anden spiller 2. Spiller har ikke nok penge til at betale huslejen 3. Spiller skal nu betale den anden spiller tilbage med sine egne ejendomme 4. Spilleren der skal betale vælger hvilke ejendomme sælges" 5. Spiller slutter sin tur
Post conditions	Gælden er betalt eller spiller er gået fallit
Alternative flows	<ol style="list-style-type: none"> 1. Spiller lander på et ejendomsfelt som er ejet af en anden spiller og spilleren har ingen penge eller ejendomme. Spilleren går fallit 2. Spiller lander på et ejendomsfelt som ikke er ejet af nogen og spilleren har ingen penge. Spilleren betaler banken med sine ejendomme og de bliver sat til salg igen 3. Spiller lander på et ejendomsfelt som ikke er ejet af nogen og spilleren har ingen penge eller ejendomme. Spilleren går fallit

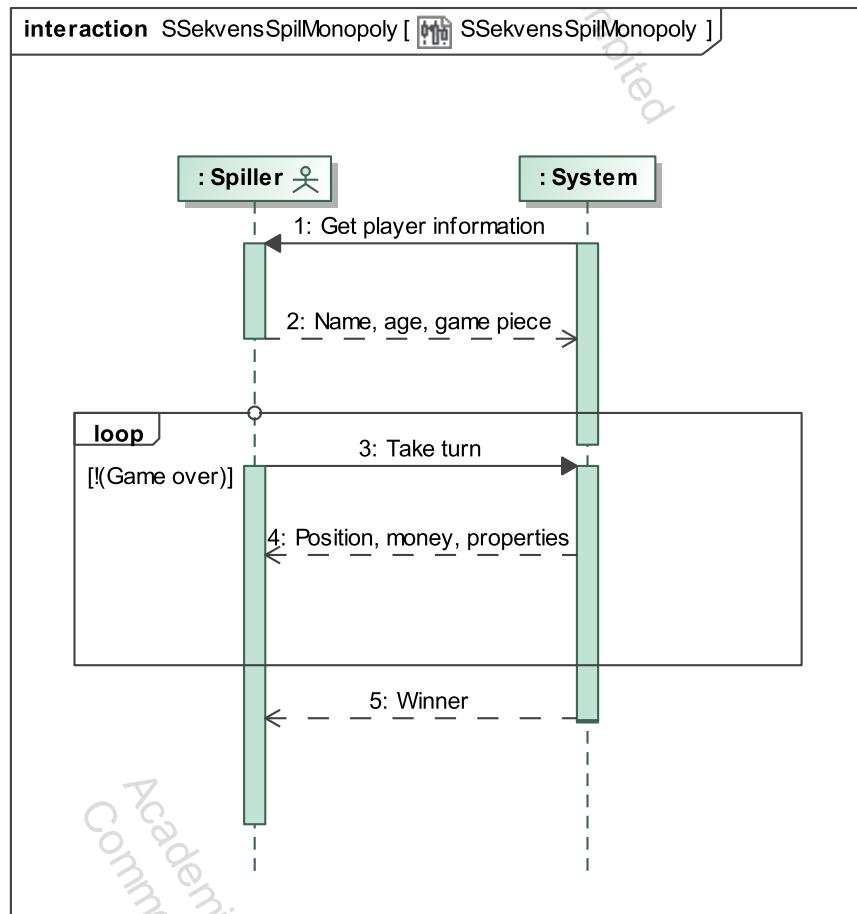
Tabel 10: UC09 Gæld

3.5 Domænemodel



Figur 2: Domænemodel til spillet

3.6 Systemsekvensdiagram



Figur 3: Systemsekvensdiagram

4 Design

I designfasen af projektet vil der tages udgangspunkt i de stillede designmæssige krav til brætspillet. Dette dækker over, at få udarbejdet et designklassediagram udfra GRASP og lavet et sekvensdiagram.

4.1 GRASP

Vi vil benytte GRASP principperne til at designe vores diagrammer, således at software-klasserne har et veldefineret og afgrænset ansvar som understøtter reusability.

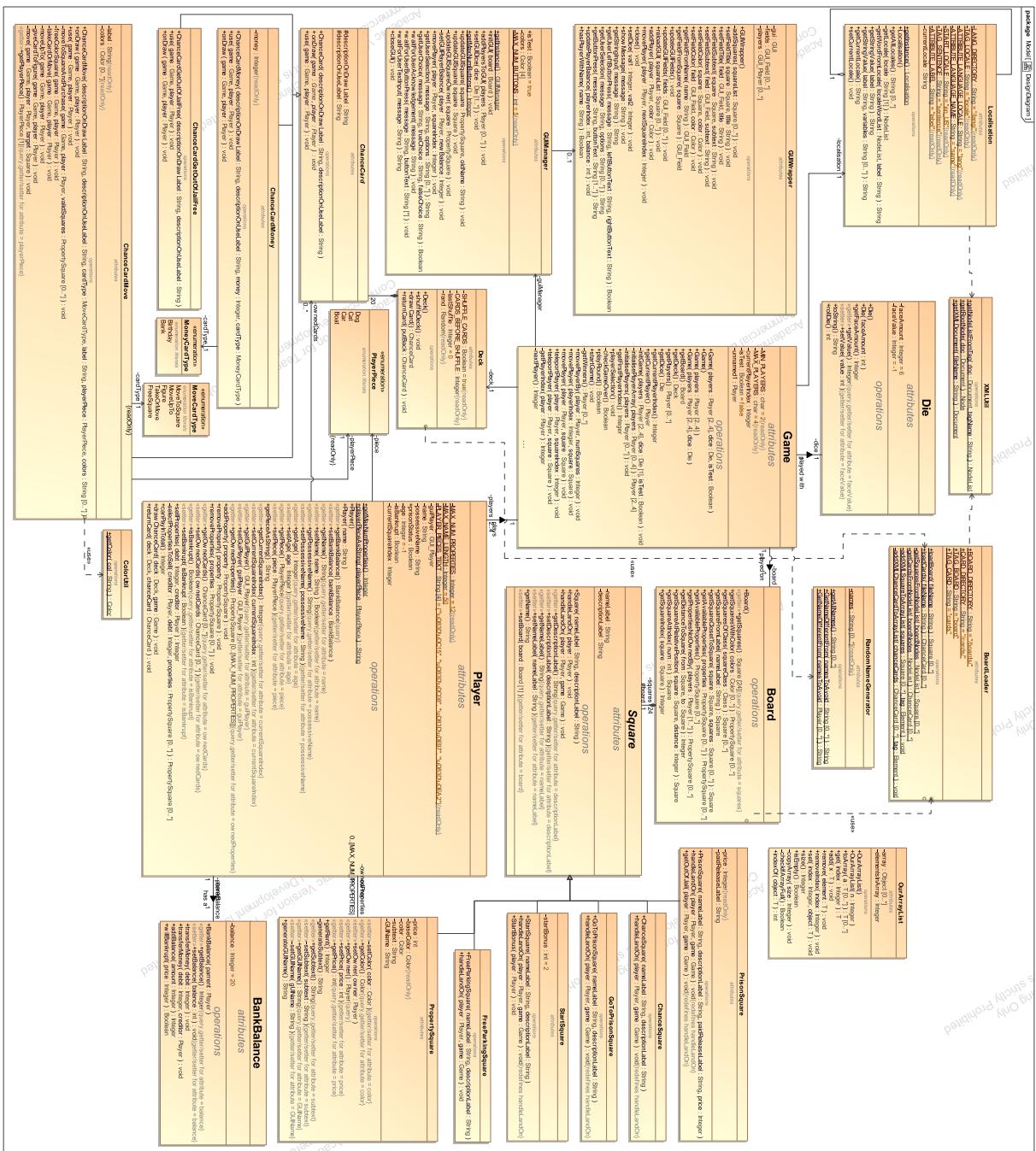
De følgende GRASP principper forklares og argumenteres for under designklassediagrammet:

- Creator: Ansvar for objektskabelse
- Information Expert: Ansvar for tildeling af ansvar til objekter
- Low Coupling: Ansvar for lav afhængighed/associationer mellem klasserne
- High Cohesion: Ansvar for høj samhørighed mellem klasserne
- Controller: Ansvar for input til systemet

[?][afsnit 4.1]

4.2 Designklassediagram

Et designklassediagram laves på baggrund af GRASP-principperne samt tidligere analytiske arbejde med særligt udgangspunkt i vores domænemodel (Figur 2), som indeholder concepter af klasser. I designklassediagrammet arbejder vi med software-klasser, som skal implementeres i systemet. Her har software-klasserne mere konkrete beskrivelser end domænemodellen. Derfor er der bl.a. beskrevet attributter, tilhørende metoder og associationer mellem klasserne. Designklassediagrammet arbejdes dermed også videre på i implementationsfasen, da der vil opstå ændringer undervejs. Understående diagram er vores endelige dokumenterede designklassediagram for systemet: [?][afsnit 4.2]



Figur 4: Designklassediagram

4.2.1 GRASP principper for designklassediagram

Nogle af principperne er genanvendt fra [?][afsnit 4.2.1]

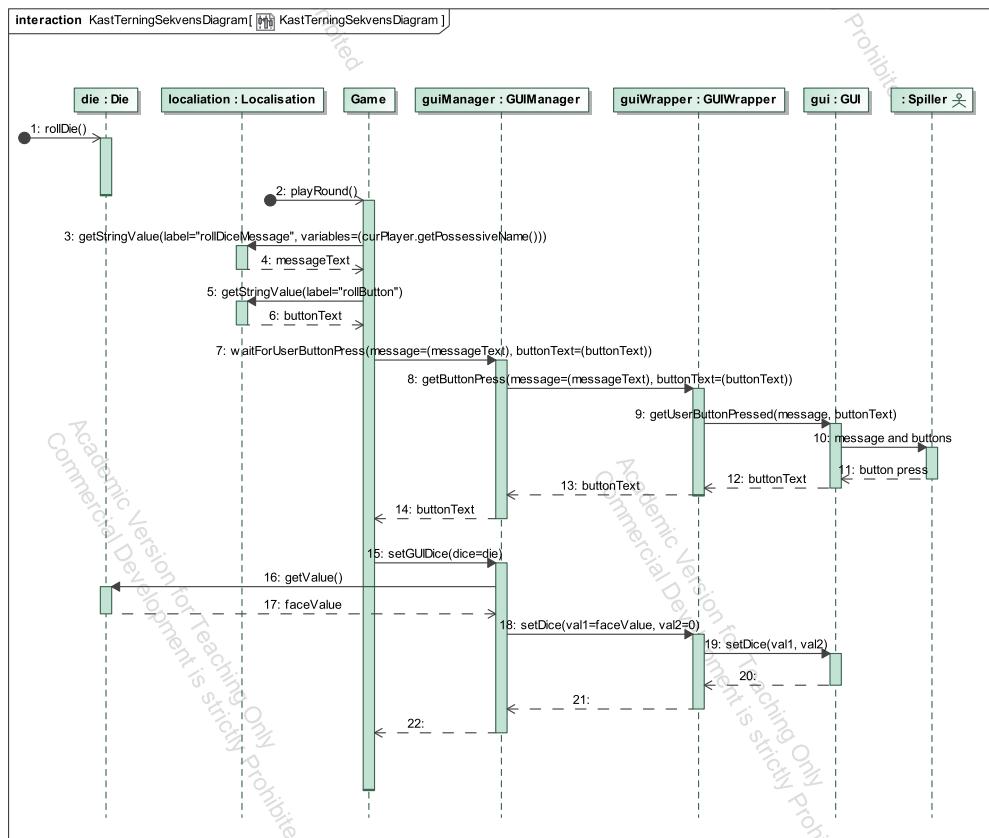
- Creator:
 - *Board* er ansvarlig for at indeholde objekter af *Square*
 - *Player* har til ansvar at benytte objekter af *BankBalance* og kan indeholde *PropertySquare* objekter samt *PlayerPiece*- og *ChanceCard* objekter.
 - *Squares* ansvarsområde omhandler indholdet af objekter fra *PropertySquare*, *StartSquare*, *PrisonSquare*, *FreeParkingSquare*, *GoToPrisonSquare* og *ChanceSquare*
 - *ChanceCard* har et ansvar om at indeholde objekter af *ChanceCardMoney*, *ChanceCardMove* og *ChanceCardGetOutOfJailFree*
 - *Deck* holder ansvaret om at indeholde *ChanceCard* objekter
- Information Expert:
 - Klassen *Game* er en information expert, der står for at håndtere spiller, dækket med *ChanceCard* og at spillerne rykkes rundt på spillepladen
 - *BankBalance*-klassen har informationerne til at ændre spillerens pengebeholdning
 - *Board*-klassen er en information expert, som håndterer felter i *Square*, der er givet ved navn og beskrivelse
 - *Square*-klassen har informationer til at skabe et felt med navn og beskrivelse
 - *PropertySquare* indeholder informationer til at generere et felt bestående af pris, farve og ejerskab
 - *Deck* har de nødvendige informationer for at blande *ChanceCard* objekter
- Low Coupling:
 - Der er hovedsageligt få koblinger mellem klasserne, hvor den største mængde koblinger er 7 på klassen *Game* efterfulgt af *Player* med 5 koblinger
 - Da *Game* og *Player* har mange koblinger, støtter de ikke low coupling, men klasserne har stadig high cohesion
- High Cohesion:
 - Generelt er der tale om high cohesion mellem klasserne, da de har en passende mængde ansvar for kodegenerering som opfyldes ved samarbejde med andre klasser
- Controller:
 - Klassen *GUIManager* vælges som facade controller, som står for inputs til systemet.

4.2.2 Abstrakt, nedarvning og polyformi

En vigtig pointe at have med fra vores design af softwareklasserne, er at vi bl.a. gør brug af abstrakte klasser. Her ses det i design klassediagrammet, at både *Square* og *ChanceCard* fungerer som abstrakte klasser. Dette betyder at man ikke kan instantiere objekter af disse, men man i stedet gør brug af nedarvning. Nedarvning dækker over at man eksempelvis har en superklasse, *Square*, med definerede attributter og metoder. Udfra superklassen kan man danne subklasser, f.eks *StartSquare* eller *PropertySquare*. Disse subklasser vil derved nedarve superklassens attributter og metoder. Det skal nævnes at subklasserne også kan have yderligere attributter og på den måde "specialisere" sig mere end superklassen, og danne forskellige typer af *Square*. En vigtig pointe at have med ift. nedarvning er, at hvis alle subklasserne har en nedarvet metode ved samme navn, eksempelvis *handleLandOn*. Her vil hver subklasse have deres egen effekt af denne metode. Dette begreb kaldes for override, inden for polymorfi i programmering, da subklasserne overskriver superklassens metode. Fordelen ved at gøre brug af abstrakte klasser og metoder, samt nedarvning og override er, at det bliver mere læseligt og mindre komplekst at implementere senere hen.

4.3 Sekvensdiagram

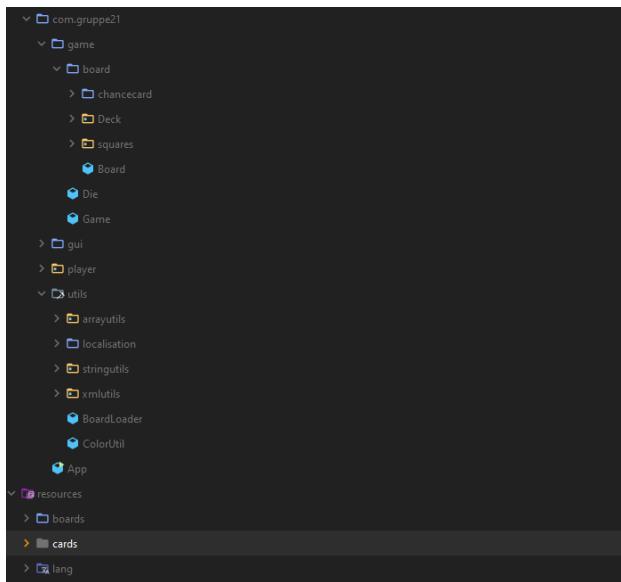
For at vise interaktionen mellem klasserne, benytter vi os af det dynamiske sekvensdiagram. I modsætning til det statiske designklassediagram, viser sekvensdiagrammet eksempelvis bedre sammenspillet mellem *Game*, *GUIwrapper*, *GUI* og *Spiller* når *rollDice* metoden bliver kaldt:



Figur 5: Sekvensdiagram over kast af terning(UC02)

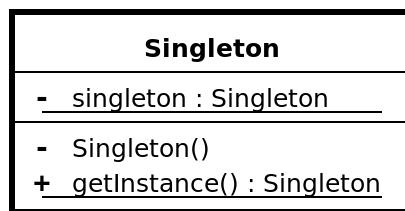
5 Implementering

Spillet er udviklet i version 1.8 af programmeringssproget Java ved brug at Jetbrains IntelliJ som udviklingsmiljø. Spillet blev ved start sat op som et Maven projekt og det har hjulpet os mærkbart med at eksempelvis at få JUnit samt GUI til at virke uden nogen gnidninger. I denne version af CDIO har vi valgt at lave mere struktur i vores projekt og vi har derfor lavet såkaldte packages, som deler vores kode op i forskellige ansvarsområder i vores folderstruktur.



Figur 6: Vores folderstruktur i projektet

5.1 Singletons



Figur 7: Singleton UML[?]

Ved at bruge Singleton mønstret bestemmer man at der kun kan være én instans af en klasse på samme tid. Ved siden af dette åbner det muligheden for at man kan tilgå denne instans i hele programmet. Dette er både rigtigt smart, men kan også være farligt for

programmet. Det er vigtigt at man er sikker på at der kun kan og skal være én enkelt instans af en klasse. I vores projekt er der to klasser som tager brug af Singleton mønstret, dette er hhv. *Localisation* og *GUIManager*. Der skal nemlig kun være en instans af *Localisation* og helt sikkert også kun en GUI (*instans af GUIManager*).

Et eksempel på hvordan singleton mønstret kan implementeres i Java kan se ud er vist herunder fra vores Localisation klasse.

```

1  private static Localisation instance;
2
3  public static Localisation getInstance() {
4      if (instance == null)
5          instance = new Localisation();
6      return instance;
7  }
8 }
```

5.2 OurArrayList

Vi har forsøgt at lave vores egen udgave af *ArrayList* til dette projekt. Den har ikke nær så mange metoder som en *ArrayList*, men den har det vi havde brug for. Vi fik implementeret *toArray*, *get*, *set*, *add*, *remove*, *removeIndex*, *size*, *isEmpty* og *indexOf*. *OurArrayList* holder egentlig bare på et array af en given type og har metoder til at manipulere med og hente data fra dette array. Vi har taget brug af generic types til klassen, som betyder at man kan give selve klassen en type parameter når den instantieres. *ArrayList* gør det samme i form af *ArrayList<T>*. Vores klasse er altså defineret som *class OurArrayList<T>*. Et eksempel på brugen af generic types ses herunder:

```

1  public T get(int index) {
2      T element = null;
3      element = (T) array[index];
4      return element;
5  }
6 }
```

5.3 XML

En vigtig ting i spil og andre applikationer nu til dags er konfiguration. Man skal kunne ændre nogle ting i sit program uden at ændre i sin kildekode. Dette gælder i vores tilfælde eksempelvis oversættelse, spillebrættet og chancekort. Vi har valgt at tage brug af XML (Extensible Markup Language) til at læse fra.

Hvad er XML?

XML er et markdown-sprog, der bruger tags meget lig HTML. Hvert åbningstag kan indeholde attributter, og mellemrummet mellem åbnings- og lukningstags kan indeholde flere tags eller almindelig tekst. Ethvert XML-dokument har brug for et såkaldt rodtag, der indeholder alle andre tags. I vores projekt har vi lavet en klasse, *XMLUtil*, til at læse XML og parse til *Nodes*. Det er de øvrige klasser *Localisation* og *BoardLoader* der står for at

bruge den data til noget.

```
1 <?xml version="1.0" encoding="utf-8" ?>
2 <root-tag>
3   <tag attribute="some value"> content </tag>
4 </root-tag>
```

Hvordan bruger vi XML?

Vi har taget brug af *DocumentBuilder* og *DocumentBuilderFactory* fra pakken *javax.xml.parsers* til at læse XML fra filer. En XML fil læses fra Maven's ressource mappe og det parses til en instans af *Document* klassen. Dette objekt indeholder en liste af *Node* objekter, som er superklasser til f.eks *Tag* klassen. Klassen *Tag* er den der indeholder informationer om et XML Tag, såsom attributter og indre tekst.

```
1 public static Document getXMLDocument(String fileName) throws ParserConfigurationException,
2   SAXException, IOException {
3   DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
4   DocumentBuilder builder = factory.newDocumentBuilder();
5   InputStream inputStream = XMLUtil.class.getResourceAsStream(fileName + ".xml");
6   Document doc = builder.parse(inputStream);
7   doc.getDocumentElement().normalize();
8   return doc;
}
```

5.4 Oversættelse med *Localisation*

Vi har lavet en oversætter, *Localisation*, som kan læse sprog fra en XML fil ved brug af *XMLUtil*. Hvert sprog har sin egen XML fil navngivet som f. eks en _US ved brug af ISO-639 og ISO-3166 koder. Det første er en kode for sproget, mens det andet er for landet.

Listing 1: Eksempel af sprog i XML

```
1 <locale lang="en_US" name="English">
2   <sentence label="word1">This is a sentence</sentence>
3 </locale>
```

Sproget er som standard sat til engelsk, men kan ændres til et andet ved at kalde *setCurrentLocale(String currentLocale)*. Derefter er det muligt at få et ord/en sætning i form at en streng i det nuværende sprog ved at kalde *getStringValue(String label)*. Der er også mulighed for at bruge variabler i en sætning *getStringValue(String label, String... variables)*. Det ser ud som vist herunder:

Listing 2: Eksempel af sprog i XML

```
1 <sentence label="requestPlayerAge">Please write your age, [PLAYER_NAME]</sentence>
```

5.5 Board

For at gøre det nemmere at oversætte spillet til andre sprog, vurderes det fra gruppens side, at den bedste løsning ville være at samle alle tekst strenge, som f.eks.

"SquareName" og "EventText" i en xml fil, for at gøre det nemmere at oversætte gruppens monopoly junior spil til andre sprog.

Selve squarene i spillet er gemt i et array, hvert element i dette array har et navn, samt tilhørende attributter. *Boardloader* indlæser navne fra xml fil, *addXMLSquareOurToArrayList*, tilføjer attributter til den hver af felterne.

I al sin enkelhed er boardet bare et objekt der er initialiseret ved at læse boardet fra en XML fil.

5.6 BoardLoader

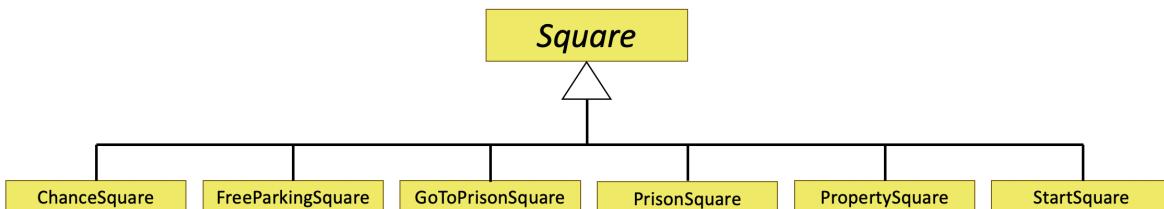
Boardloader indeholder metoder der fortolker og opretter ChanceCards og Squares objekter fra en XML fil. Hvilket gør det nemmere at fjerne, tilføje eller modifcere felter på spillebrættet eller chancekortene, da ændringer af disse bare skal laves i en XML fil.

Ændringer til *Squares*, skal laves i main_board.xml, og ændringer til *ChanceCard* skal laves i referencen til *ChanceCard.xml*.

5.7 Square

Square klasserne er gør brug af nedarv. *Square* klassen er fungerer som parent klassen. Klassen har metoder, så som get og set metoder, men den indeholder også metoden *handleLandOn*, som har ansvaret for hvad der sker når en spiller lander på et specifikt felt.

De forskellige felter har forskellige funktioner og gør forskellige ting. Derfor er der lavet en felt klasse til hver unik handling. Ved at bruge nedarv kan vi nemt bruge metoder af samme navn og slipper for besværet der kan opstå, hvis der var flere forskellige metoder der skulle bruges afhængigt af hvor man er på spillepladen. Her kan ses en oversigt over alle felterne og hvordan de nedarver fra *Square*:



Figur 8: Oversigt over nedarven i *Square* klasserne

Fælles for alle child klasserne er, at de har sin egen konstruktør og de overrider også *handleLandOn* metoden fra . Der er nogle unikke metoder for nogle af klasserne som er nødvendige for feltets udførelse. F.eks., har *StartSquare* en *startBonus()* metode, som håndterer at spilleren får en startbonus når man passerer start. Metoden kan ses her:

```

1 public class StartSquare extends Square {
2     private int startBonus = 2;
3 .
4 .
5 .
6     public void StartBonus(Player player) {
7         player.getBankBalance().addBalance(startBonus);
8     }
  
```

Der gøres derudover også brug af XML-filer til at håndtere, hvilken tekst der skal vises for brugeren når de lander på et specifikt felt. Dette gælder både navn på feltet og dens beskrivelse.

5.8 BankBalance

Bankbalance er en klasse, hvis ansvar er at holde styr på spillernes pengebeholdning. Klassen har en konstruktør, som sætter variablen *parent* til at referere til den tilhørende spiller og startpengebeholdningen til 20 ÅA. Derudover har klassen en *addBalance* funktion, hvis opgave er at tilføje eller trække penge fra en spillers pengebeholdning. I forlængelse af dette, har *Bankbalance* fået implementeret to *transferMoney* metoder, der gør brug af overload-princippet. Dette betyder begge metoder har samme navn, men forskellige parametre og derfor forskellige funktionaliteter. Disse metoder er implementeret på følgende måde i programmet:

```

1 public void transferMoney(int debit){
2     transferMoney(debit, null);
3 }
  
```

```

5  public void transferMoney(int debit, Player creditor){
6      if (creditor == parent) return;
7      if (getBalance() < debit)
8      {
9          debit -= getBalance();
10         transferMoney(getBalance(), creditor);
11         debit -= parent.sellProperties(debit, creditor);
12     }
13     addBalance(-debit);
14     if (creditor != null){ //creditor == null -> creditor is the bank
15         creditor.getBankBalance().addBalance(debit);
16     }
17 }
```

Den første *transferMoney* metode bruges til, når der skal betales til banken. Dette gøres ved at kalde den anden *transferMoney* metode, ved at indsætte null på *creditor*'s plads i parametrene. Ønsker man derimod, at der skal betales til en anden spiller, indsætter man derimod en reference til den ønskede spiller på *creditor*'s plads i parametrene.

Metoden *transferMoney* tjekker først om den indsatte *creditor* er samme spiller, som pengebeholdningens *parent* - hvis dette er tilfældet returneres metoden. Er pengebeholdningen under nul, vil en spiller ikke have penge nok til, at kunne betale sin *creditor*. Derfor vil *transferMoney* kalde dens tilhørende *parent*'s *sellProperties* metode, som sælger en af spillernes ejendomme. Hvis pengebeholdningen derimod ikke er under nul, vil værdien (*debit*) trækkes fra den givne spillers pengebeholdning via *addBalance* og evt. overføres til *creditors* pengebeholdning, hvis der er tale om en anden spiller.

5.9 Die

Vi har benyttet samme *Die* klasse som vi havde lavet i vores tidligere projekter CDIO1[?] og CDIO2[?]. *Die* klassen er ansvarlig for terningen der bliver brugt i spillet. *Die* klassen skal virke som en terning hvor at den bliver kastet og giver en værdi mellem 1 og 6. Dette gøres ved at benytte Math klassens random metode til at generere et tilfældigt tal i intervallet [0,1] og selve implementeringen kan ses her:

```

1  public int rollDie() {
2      faceValue = (int) (Math.random() * faceAmount + 1);
3      return faceValue;
```

5.10 Game

Game er den største af klasserne i spillet og er også den med det største ansvar. Den står nemlig for at styre selve spillet og hvordan runderne køres. *Game* klassen fungerer omtrent på samme måde som i vores CDIO 2 projekt, men er denne gang udvidet til at bruge Monopoly Juniors regler. Den er derudover også blevet opdelt i forskellige metoder efter ansvar for at øge læsbarheden af koden. I det tidligere projekt havde vi en stor del af koden i en metode, hvilket nu er opdelt i flere mindre delelementer. *Game* holder eksempelvis på spillerne og spilbrættet.

5.11 GUIWrapper

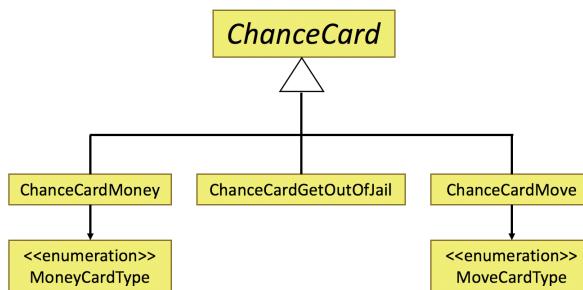
GUIWrapper er ligesom i CDIO 2 en wrapper klasse til det GUI bibliotek som kunden har stillet til rådighed. Den er lavet så vi kan tilpasse det og gøre det nemmere at implementere i programmet. Den store ændring vi har lavet i CDIO 3 til denne klasse er at den nu bliver instantieret og styret af *GUIManager*

5.12 GUIManager

GUIManager klassen står for kommunikationen mellem spilleren/GUI og resten af programmet. Den er implementeret med Singleton mønstret, som vi forklarede tidligere i kapitlet. Hovedårsagen til at vi lavede denne klasse var at gøre det nemmere at arbejde med GUI'en for hele udviklerholdet.

5.13 ChanceCard

ChanceCard er opdelt i en abstrakt superklasse, nedarvende subklasser og eventuelt tilhørende typer af kort. Hver subklasse repræsenterer en instans af ChanceCard, som hver har deres unikke funktionaliteter. Disse associationer kan opstilles i et klassediagram på følgende måde, ligesom der er vist i designklassediagrammet[Figur 4]:



Figur 9: Oversigt over *ChanceCard* klasserne

ChanceCard-superklassen

Superklassen *ChanceCard* er en abstrakt klasse, som har to String attributter: *descriptionOnDrawLabel* og *descriptionOnUseLabel*. Derudover har *ChanceCard* to metoder, hhv. *onDraw* og *use*. De nedarvende klassers metode *onDraw* bliver kaldt, så snart et pågældende kort bliver trukket fra *Deck* og metoden *use* aktiveres når kortet gøres i brug. Både *onDraw* og *use* bliver da overridet, for hver subklasse af *ChanceCard* superklassen. En af de fordele der er ved, at gøre brug af nedarvning er at f.eks. *returnCard* bliver kaldt i metoden *use* for alle subklasserne. Derved er man sikker på, at alle kort bliver lagt tilbage i *Deck* efter brug. Derudover gøres det også nemmere, at implementere subklasserne, da dette ikke skal tilføjes for hver enkelte.

```

1 public abstract void onDraw(Game game, Player player);
2
3     public void use(Game game, Player player) {
4         player.returnCard(game.getDeck(), this);
5     }

```

ChanceCardGetOutOfJailFree

Subklassen, *ChanceCardGetOutOfJailFree*, nedarver fra superklassen *ChanceCard* og repræsenterer det kort, som omfatter, at en spiller kan komme ud af fængslet uden omkostninger. Dette kort er meget simpelt, da den består af to nedarvede metoder. Metoden *onDraw* overrider superklassens metode og henter en beskrivelse af kortets funktionalitet fra XML-filen og viser dette via GUI'en. Klassen har derudover også metoden *use*, som viser en beskrivelse i GUI'en af, at kortet gøres i brug. Metoden *use* bliver her aktiveret i *Game*, hvis en spillers status er i fængsel og den givne spiller ejer kortet.

ChanceCardMoney

Subklassen *ChanceCardMoney* håndterer de ChanceCard som har indflydelse på spillernes *BankBalance*.

Den indeholder attributterne *money* og *cardType*.

Attributten *money* er et heltal, som viser beløbet der enten kan tildeles eller trækkes fra spillere.

Attributten *cardType* er af typen *MoneyCardType*, som repræsenterer de mulige ChanceCards der er relevante for klassen.

Klassen repræsenterer følgende ChanceCards via *MoneyCardType*, som forklares efterfølgende:

- *Birthday*-ChanceCard, hvor den spiller der får trukket kortet, modtager 1 Å fra de resterende spillere.
- *Bank*-ChanceCard, som dækker over to ChanceCard, som spilleren kan trække. Den ene ChanceCard har beskrivelsen candyCard, hvor spilleren skal betale 2Å, mens den anden er homeworkCard, hvor spilleren modtager 2Å. Dette kan ses inde i GUI'en og fremgår ikke tydeligt i selve *ChanceCardMoney*.

Følgende stykke kode illustrerer henholdsvis *Bank*-ChanceCard og *Birthday*-ChanceCard:

```

1 public void use(Game game, Player player) {
2
3     switch (cardType) {
4         case Bank:
5             player.getBankBalance().addBalance(money);
6             break;
7         case Birthday:
8             for (Player debtor : game.getPlayers()) {
9                 //If player == debtor nothing changes.
10                debtor.getBankBalance().transferMoney(money, player);

```

```

11     }
12     break;
13   }
14 }
```

MoneyCardType-subklassen repræsenterer de ChanceCard der benyttes til *ChanceCardMoney*. Konstruktøren til *ChanceCardMoney* tager en *MoneyCardType* som parameter, hvor *MoneyCardType* kun kan have værdierne *Birthday* og *Bank*.

Fra *MoneyCardType*:

```

1 public enum MoneyCardType {
2   Birthday,
3   Bank
4 }
```

ChanceCardMove

Subklassen *ChanceCardMove* håndtere den type ChanceCard som omhandler spillerens felt på brættet.

Følgende typer ChanceCards forekommer i subklassen via *MoveCardType*:

- *MoveToSquare*-ChanceCard flytter spilleren til et nyt felt, hvilket er nødvendigt for det ChanceCard, som flytter spilleren til startfeltet, Strandpromenaden eller Skaterparken.
- *MoveUpTo*-ChanceCard gør det muligt for spilleren at vælge, hvor mange felter man vil rykke frem (fra 0-5).
- *Figure*-ChanceCard er et figurkort som refererer til et af figurerne (Hund, Kat, Skib og Bil), hvor spilleren med en bestemt figur har mulighed for at købe et selvvalgt felt (med det antagelse at feltet er ledig).
- *TakeOrMove*-ChanceCard, hvor spilleren kan enten vælge at tage endnu en ChanceCard eller rykke et enkelt felt frem.
- *FreeSquare*-ChanceCard som giver spilleren mulighed for at rykke frem til et farvet felt og få feltet gratis (under den antagelse at feltet er ledig).

Nedenstående kode viser cases med de forskellige værdier som *MoveCardType* repræsenterer:

```

1 public void use(Game game, Player player) {
2   switch (cardType) {
3     case MoveToSquare:
4       move(game, player, game.getBoard().getSquareFromLabel(label));
5       break;
6     case MoveUpTo:
7       moveUpTo(game, player);
```

```

8     break;
9 case Figure:
10    PropertySquare[] validSquares = game.getBoard().getAvailableProperties();
11    if (validSquares == null) {
12      validSquares = game.getBoard().getPropertiesNotOwnedBy(player);
13    }
14    moveToSquareAndPurchase(game, player, validSquares);
15    break;
16 case TakeOrMove:
17    takeCardOrMove(game, player);
18    break;
19 case FreeSquare:
20    freeColorSquare(game, player);
21    break;
22  }
23 }
```

Parameteren cardType har typen MoveCardType, som kan have værdier, der er illustreret i følgende stykke kode:

```

1 public enum MoveCardType {
2   MoveToSquare,
3   MoveUpTo,
4   Figure,
5   TakeOrMove,
6   FreeSquare
7 }
```

5.14 Player

Der er taget udgangspunkt i samme *Player*-klasse som anvendt i tidligere CDIO projekter, CDIO1[?] og CDIO2[?]. Klassen er i dette tilfælde videreført til at kunne indeholde relevante værdier angivet ud fra kravspecifikationen. Dette er værdier såsom spillernavn og spilleralder, ligesom at klassen også instantierer objekter fra henholdsvis *PlayerPiece* og *BankBalance* klassen, nemlig henholdsvis spillerbrikkerne fra *PlayerPiece* og spillerens pengebeholdning i ÅA fra *BankBalance* klassen.

Player klassen står endvidere for håndtering af en række øvrige informationer om Monopolyspillets spillere. Det er desuden i *Player* klassen at der tilskrives en oversigt over aktive chancekort, samt de ejendomme som en given spiller ejer. Der er endvidere boolske værdier der angiver, hvorvidt spilleren er konkurs eller i fængsel. Klassen gør brug af eksterne tekster til brug af spilbrikkerne.

Foruden at indeholde værdier for spillernavn, spilleralder, aktive chancekort og ejede ejendomme, så foregår selve reguleringen af de omtalte værdier ligeledes gennem *Player* klassen. Det vil altså sige at det er *Player* klassen der sætter en alder, navn, trækker chancekort, returnerer chancekort, køber- og sælger ejendomme, samt vurderer hvorvidt spilleren er konkurs, og dermed har tabt spillet. Det er ligeså *Player* klassen der tilskriver den aktuelle placering på spilbrættet.

Player klassen indeholder endvidere skærpede regler, under de angivne avancerede spilleregler. Når spillet vælges at spilles under de avancerede regler, så tilføjes muligheden for at sælge sine ejendomme, hvilket håndteres af *Player* klassen. Til brug herfor findes metoder for køb og salg af ejendomme.

5.15 Deck

For at matador junior spillet kan indeholde 20 chancekort som er beskrevet i reglerne, vælges der at implementere en Deck klasse. Klassen initialisere de 20 kort, samt indeholder metoderne drawCard, shuffleDeck og returnCard.

Kortene bliver initialiseret, ud fra en XML filen, samt klassen BoardLoader. I XML filen er alle kortene i spillet beskrevet, dvs. XML filen indeholder kortenes individuelle attributter. Som bliver læst af BoardLoader, og instantieret rigtigt.

Det fysiske monopoly jr. indeholder som tidligere nævnt 20 kort, hvor spilleren skal trække det øverste kort, hvis denne lander på et chance felt i spillet, hvorefter spilleren gør som beskrevet på kortet.

Det blev besluttet at Deck klassen, repræsenterer de kort som i det fysiske spil vil ligge på bordet. Dvs. når en spiller trækker et kort, får spilleren et kort af Deck, og når spilleren lægger sit kort tilbage, modtager Deck et kort af spilleren. Så derfor skal deck kunne holde styr på hvor mange kort der er i bunken, da spillere ifølge reglerne kan have kort på sig.

Der er fra gruppens side valgt at kortene skal ligge på hver sin plads i et array. Arrayet der indeholder kort vil fremover blive refereret til som cards. I al sin enkelhed var ideen at spilleren trækker kortet på den første plads i cards, og når spilleren er færdig med kortet bliver det lagt i bunden i cards.

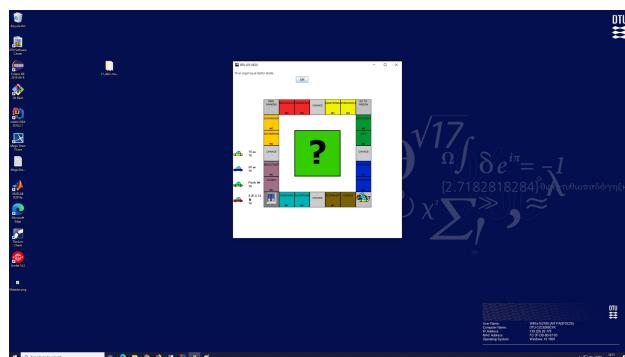
For at have muligheden for at blande kortene i spillet bliver metoden shuffleDeck indført. Metoden shuffleDeck er en meget simpel funktion, som starter på første plads i cards, og bytter med et andet tilfældigt kort i cards. Og herefter tager næste plads i cards og bytter med et andet tilfældigt kort i cards, og så videre indtil den når til sidste plads i cards.

Det viser sig her at det er belejligt at når spilleren trækker et kort bliver det "fjernet" fra cards. Da der så i ShuffleDeck metoden ikke behøver at blive taget højde for de kort som spillene sidder med.

6 Test

6.1 DTU maskine og responstid test

Kunden har ønsket, at programmet skal kunne køres på en af DTU's databars maskiner. Dette testes ved netop at benytte en af disse maskiner, som kunden har bedt om. Resultatet af testen har vist, at spillet kan køre uden problemer. Her kan der ses dokumentation, i form af et screenshot, som påviser, at programmet kører og testes:



Figur 10: Skærmbillede fra DTU databarmaskine

En af de ting der er valgt at teste, ift. at kunne sikre brugervenlighed af programmet, har været ved at måle på responstiderne. Her har vi taget udgangspunkt i, at forsinkelsen maksimalt må være på 1s efter brugeren har trykket på enter knappen. Måden dette er implementeret er ved, at indsætte en timer i koden vha. *currentTimeMillis* metoden, der mäter hvor lang tid det tager for metoderne at eksekvere. Koden fungerer på den vis, at der en klasse, *RespsonsTime*, med en statisk variabel for hver maksimale forsinkelse pr. testede metode. Eksempelvis er der en statisk variabel *MAX_roll*, som holder styr på den højest testede forsinkelse for metoden *rollDie* i *Die* klassen. Outputtet for tidtagningen vil således blive vist i konsollen løbende, og vil til sidst i programmet vise de endelige målinger. Dette er gjort, så man løbende kan identificere og analysere hvor og hvornår, der evt. kan forekomme større forsinkelser. En ting man skal være opmærksom på ift. denne måde at teste er, at tiden bliver taget inde i selvemetoden. Der kan derfor godt være en afvigelse ift. tiden det tager at returnere. Derudover er oversigten i konsollen til sidst lavet for, at give et overblik over målingernes maksima. Nedenfor ses et eksempel på, hvordan testen kan implementeres med *Die*'s metode *rollDie*:

```

1 public int rollDie() {
2     long timerStart = System.currentTimeMillis();
3
4     faceValue = (int) (Math.random() * faceAmount + 1);
5
6     long timerEnd = System.currentTimeMillis();
7     long totalTime = timerEnd - timerStart;
8     if(totalTime > RespsonsTime.getMAX_roll()) RespsonsTime.setMAX_roll(totalTime);
9     System.out.println("Respsons time: " + totalTime + "ms for rollDie() | Current max: " +
    RespsonsTime.getMAX_roll() + "ms");

```

```

10
11     return faceValue;;
12 }
13

```

Vi har i dette projekt været i lettere tidsnød, og har derfor ikke fået, at måle på alle dele i programmet. Det har derfor været besluttet i gruppen, at prioritere måling på de syv metoder, som er mest anvendte. Dette er gjort da gruppen har vurderet, at disse metoder vil kunne have størst indvirkning på brugeroplevelsen og responstiden af systemet.

Nedenfor ses et screenshot af responstiderne for udvalgte metoder:

```

Respons time: 0ms for rollDie() | Current max: 3ms
Respons time: 0ms for teleportPlayer() | Current max: 3ms
Respons time: 0ms for transferMoney() | Current max: 2278ms
Respons time: 57ms for setOwner() | Current max: 58ms
Respons time: 154ms for transferMoney() | Current max: 2278ms
-----
Max respons time for setOwner: 58ms
Max respons time for teleportPlayer: 3ms
Max respons time for transferMoney: 2278ms
Max respons time for rollDie: 3ms
Max respons time for shuffleDeck: 0ms
Max respons time for returnCard: 0ms
Max respons time for drawCard: 1ms
-----
```

Figur 11: Eksempel på output i konsol

Undervejs i det spillet har kørt, har der overordnet set ikke opstået nogle påfaldende forsinkelser for det blotte øje. Der er derudover også foretaget diverse målbare hastighedstest vha. IntelliJ. Disse test viser at programmet har været ca. 10s om at åbne op og har haft maks. 2278ms (ca. 2.3s) forsinkelse. Hastigheden for opstart af programmet er over de 1s - dog har kunden fortalt, at vi kan se bort fra denne afgivelse. Forsinkelsen på 2.3s virker ved første øjekast til at være meget højt, ift. kravsetningen på 1s - hvilket det også er.

Dog viser dette ikke det generelle billede, for de syv testede metoder. Som det kan aflæses på ovenstående screenshot, ligger alle metodernes maksimum langt under kravet på 1s, på nær *transferMoney* metoden i *BankBalance* klassen. Det har her kommet til gode, at testen viser responstid for hver udførte metode undervejs, og ikke kun til sidst i spillet. Dette har nemlig gjort, at vi har kunne få svar på forsinkelsens oprindelse. Det har vist sig, at forsinkelsen på 2.3s opstår, når en spiller skal vælge hvilke ejendomme vedkommende ønsker at sælge, for at kunne betale gæld til en anden spiller. Årsagen er, at selve metoden bliver kørt i det den pågældende spiller lander på feltet, og kører indtil spilleren har indtastet, hvilken ejendom der ønskes solgt. Denne tid afhænger derfor af, hvor længe spilleren er om at vælge, og viser derfor ikke et retmæssigt billede af responstiden for selve metoden. Metoden *transferMoney* ligger typisk et sted mellem 0-200ms for størstedelen af spillets løbetid.

Selvom det umiddelbart virker voldsomt at testen viser en maksimum responstid på ca. 2.3s, vil vi i gruppen dog stadig vurdere, at kravet er opfyldt. Dette bygges på, at de

resterende seks ud af syv metoder kører langt under kravsetningen og den metode der kører længst tid, skyldes en udefrakommende faktor. Det skal dog nævnes, at testen ikke er 100% sigende ift. hele programmet, men vi vurderer, at det er nok til at dække målsætningen. En vigtig pointe at have med fra denne test er derudover også, at det er vigtigt at være kritisk overfor hvad man mäter på og hvordan man gør det.

6.2 Brugertest

Som nævnt tidligere i afsnittet, er der til dette projekt også blevet anvendt brugertest. En brugertest dækker over en kvalitativ form for feedback, som kommer fra en ekstern person, med mindre til ingen kendskab til indholdet. I dette tilfælde har vores målgruppe for Monopoly junior spillet været børn i alderen 5+. Der er derfor valgt en anonym testperson, i alderen 14 år og med kendskab til familiespil, som vil komme med feedback til programmet. Løbende i testen vil testpersonen kommentere på sin oplevelse, som vil noteres og evalueres på til videre udvikling.

Fordelen ved, at få feedback fra en udefrakommende er, at personen typisk vil have et andet fokus på fejl og særligt brugervenlighed - dette er i de fleste tilfælde ting, som et udviklingshold ikke nødvendigvis altid vil opdage. Der kan derudover både være fordele og ulemper ved, at indrage brugertest tidligt og sent i et projektforsløb. Der er i dette projekt valgt, at brugertesten er foretaget til sidst i projektet. Dette har efter hensigt været planlagt for, at kunne finjustere det endelige produkt til kunden. Dog skal det nævnes, at vi først fik gennemført brugertesten, på selve dagen kunden ønskede projektet leveret. Brugertesten kunne dog også være lavet tidligt i projektet, hvilket vil give anledning til hurtigere, at kunne få identificeret og rettet fejl og mangler. En tilføjelse til dette er, at man i nogle tilfælde kan opdage planlagte aktiviteter, som kan vise sig unødvendig efter en brugertest. I større IT-projekter (eller projekter generelt) kan dette eksempelvis være en fornuftig idé, da det kan vise sig, at f.eks. en kostbar implementering kan blive sparet. Det kræver forholdsvis få ressourcer, at opstille en brugertest, men kan have et stort afkast i sidste ende.

Denne brugertest har påvist at vores spil ikke har et moderne design og at det kan være rigtig svært at forstå hvordan det spilles. Spillet blev testet af en 14 årig testperson, som mente at det så gammeldags ud og allerede der mistede lysten til at spille det. Testpersonen prøvede sig alligevel igennem spillet, men brugerfladen gjorde det svært for personen at vide hvad der skete. Efter at få forklaret hvad der skete, gik det bedre. Dog er det slet ikke optimalt og viser at det er meget vigtigt også at tænke på udséendet. Vi har forsøgt at få spillet til at være sjovt ved at bruge nogle underholdende tekster til chancekort og på felter i spillet, men det viste sig ikke at være nok til at interessere testpersonen.

6.3 JUnit tests

I dette projekt har det været planlagt, at køre diverse JUnit test for størstedelen af programmet. En JUnit test dækker over, at metoder og klasser testes separat fra resten af

programmet. Dette er med til at sikre, at koden kører efter hensigten. Der er sågar udviklingsmetoder, Test Driven Development (TDD) eksempelvis, som anvender tests til, at skrive kode ud fra. Dette kan man desværre ikke sige er gældene for vores projekt. Der er blevet genbrugt enkelte JUnit tests fra tidligere projekter, men har kun fået én yderligere siden CDIO2[?]. Her er der blevet udarbejdet en test, *canLoadFile*, som tester om *Board* kan hentes fra XML fil - hvilket er en væsentlig test at have med. Dette er dog ikke optimalt ift. dels *coverage*, som vil uddybes senere i afsnittet, men også for sikkerheden i, at funktionaliteten af programmet kører efter hensigten. Årsagen til at det har været svært, at udvikle flere JUnit test har primært været grundet GUI'en og for høj kobling. Eksempelvis har der været klassen *ChanceCard*, som har skulle bruge information for størstedelen af koden - hvilket ikke er optimalt.

Det har været et krav fra kundens side af, at der mindst har skulle være én JUnit test pr. centrale metode. Dette har desværre ikke været tilfældet for det endelige projekt til kunden. Vi har dog efterfølgende fået implementeret en *isTest* boolean i *GUIManager*, som helt klart kunne være en løsning til fremadrettede projekter. Denne løsning nåede ikke at blive implementeret denne gang, hvilket skyldes tidspress for det samlede projekt. Problemet vedrørende høj kobling, har vi dog på nuværende tidspunkt ikke en løsning på endnu. Vi har dog taget ved lære, og vil rette op på dette til fremtidige projekter.

6.4 Test af terning

Til at teste om terningen kast er tilfældige og at slagene er ligefordelt vil der henvises til tidlige tests vi allerede har lavet i en tidligere opgave. Vi har udført disse tests i afsnittet *Test af tilfældighedsgenerering af enkelt terning*[?][s.15-16] samt *Bilag 3*[?][s. 26] der viser at fordelingen af slagene er ligefordelt. Der er blevet brugt samme *Die* klasse i den tidlige opgave og i denne.

Denne fremgangsmåde for test, kunne også være anvendt på *shuffleDeck* i *Deck*, vis vi havde haft mere tid. Der kunne f.eks. måles på om der generes en tilfældig rækkefølge af kortene.

6.5 Formelle Testcases

I projektet har vi valgt, at udarbejde tre forskellige testcases, da dette har været et krav for kunden. Disse testcases dækker over en kort oversigt over, hvordan gruppen (og evt. kunde) kan genskabe testen, hvis dette ønskes. Derudover viser testcase også om, hvorvidt vores tests har været fuldendt eller ej.

TC01 - DTU maskine

Test Case ID	TC01
Summary	Test der viser at spillet kan køres på DTU Databar maskine
Requirements	DTU computer der eksekverer Java 1.8
Preconditions	Personen tænder og logger ind på en DTU databar maskine
Postconditions	Programmet kører på DTU maskinen
Test procedure	<ol style="list-style-type: none"> 1. Personen henter spillet ned på maskinen (eventuelt vha. guide) 2. Personen kører programmet succesfuldt på maskinen
Test data	
Expected result	At spillet kan spilles på maskinen
Actual result	Spillet kan spilles på DTU's maskiner
Status	Bestået
Tested by	Troels Toy Christoffersen
Date	27/11 - 2020
Test environment	<ol style="list-style-type: none"> 1. IntelliJ IDEA 2.1.2019 2. On Windows 10

TC02 - Brugertest

Test Case ID	TC02
Summary	Brugertest hvor testperson(er) giver feedback på programmet, imens det køres
Requirements	<ol style="list-style-type: none"> 1. En (eller flere) testpersoner 2. En computer der kan køre Monopoly Junior
Preconditions	At programmet er hentet og igangsat
Postconditions	Der er kommet feedback fra testpersonen
Test procedure	<ol style="list-style-type: none"> 1. Personen følger en givet guide til at starte spillet 2. Testperson giver løbende feedback på sin brugeroplevelse 3. Der noteres løbende med observationer og feedback. 4. Testperson spiller spillet til ende
Test data	Anonym pige på 15 år
Expected result	At testperson giver konstruktiv feedback
Actual result	Testperson havde svært ved at se hvad man skulle samt hvad der skete og klikkede sig igennem et spil
Status	(Bestået) forhåbentlig
Tested by	Tobias Maneschijn
Date	27/11 - 2020
Test environment	<ol style="list-style-type: none"> 1. IntelliJ IDEA 2020.2.2 (Community Edition) 2. Windows 10 build 1909

TC03 - Responstest

Test Case ID	TC03
Summary	Denne test viser eksekveringstiden for syv udvalgte metoder
Requirements	<ol style="list-style-type: none"> 1. En computer der kører Java 1.8 2. Adgang til GitHub repository og filer
Preconditions	Person har tændt computer og startet IntelliJ
Postconditions	Test opfylder krav 27 og 38
Test procedure	<ol style="list-style-type: none"> 1. Personen tilgår gruppens Github projektet 2. Personen vælger MonopolyJuniorTest branch 3. Personen kører spillet via. IntelliJ 4. Personen aflæser resultater løbende fra konsolen 5. Personen analyserer resultatet fra konsolen
Test data	
Expected result	Ingen forsinkelser på over 1s
Actual result	6 ud af 7 metoder testet til under 1s
Status	Bestået
Tested by	Marcus R. Kjærsgaard
Date	26/11 - 2020
Test environment	<ol style="list-style-type: none"> 1. IntelliJ IDEA 2020.2.1 2. macOS High Sierra 10.13.6

6.6 Code coverage

IntelliJ har den funktion at kunne teste et program med *coverage*. Dvs, at man kører programmet og så registrerer IntelliJ hvor meget af ens kode der bliver anvendt. Det er selvfølgelig målet, at det meste af koden bliver brugt når programmet testes, men man skal også være opmærksom på at noget af koden også er til fejlhåndtering og vil nødvendigvis ikke blive kørt medmindre, at der sker noget til at køre de stykker af kode. Der besträbes, at code coverage'en er på 80% når der testes for, at køre programmet, som når der spilles.

Code coverage for spillet:

Her ses resultatet for coverage:



Figur 12: Billede af resultatet af spillet testet med coverage

Element	Class %	Method %	Line %
Game	100 %	79 %	75
GUI	100 %	95 %	92 %
Player	100 %	86	87
Utils	83 %	83 %	79 %

Tabel 11: Opsummering af coverage

Det kan aflæses på ovenstående screenshot og tabel, at der har været en coverage på over 80% for det samlede program. Dette viser et fornuftigt billede af, at størstedelen af den implementerede koder, bliver brugt i løbet af en gennemspilning. Det skal derudover også nævnes, at noget af koden også bliver anvendt til fejlhåndtering og vil derfor ikke, som udgangspunkt, blive brugt for alle spil.

Code coverage for tests:

Udover at man kan teste coverage for når spillet kører, så kan man også vælge at teste coverage for sine tests. Dette er blevet gjort for de få JUnit test, som vi har haft med i vores program. Vi har sat os en målsætning for at opnå 15% coverage for denne test. Dette ser ud på følgende måde i et screenshot:

58% classes, 23% lines covered in 'all classes in scope'			
Element	Class, %	Method, %	Line, %
com.gruppe21	0% (0/1)	0% (0/1)	0% (0/3)
com.gruppe21.game	50% (1/2)	8% (3/35)	3% (7/186)
com.gruppe21.game.board	100% (1/1)	15% (2/13)	14% (8/56)
com.gruppe21.game.board.chancecard	0% (0/8)	0% (0/22)	0% (0/113)
com.gruppe21.game.board.Deck	0% (0/1)	0% (0/4)	0% (0/25)
com.gruppe21.game.board.squares	100% (7/7)	34% (15/43)	27% (38/140)
com.gruppe21.gui	50% (1/2)	5% (2/40)	3% (7/180)
com.gruppe21.player	66% (2/3)	30% (13/42)	34% (45/132)
com.gruppe21.utils	100% (2/2)	62% (5/8)	55% (44/80)
com.gruppe21.utils.arrayutils	100% (1/1)	76% (10/13)	67% (41/61)
com.gruppe21.utils.localisation	100% (1/1)	66% (6/9)	55% (30/54)
com.gruppe21.utils.stringutils	100% (1/1)	100% (4/4)	100% (21/21)
com.gruppe21.utils.xmlutils	100% (1/1)	100% (3/3)	100% (8/8)

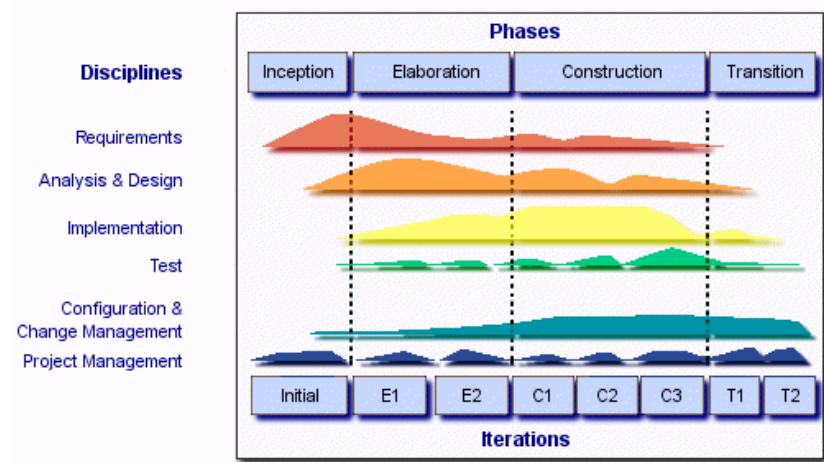
Figur 13: Billede af resultatet af tests med coverage

Som der kan ses på ovenstående screenshot, viser det sig at vi har nået en coverage af vores samlede kode, på 53% af klasserne. Dette er delvist tilfredsstillende, da vores målsætning har været at opnå 15% for vores samlede kode. Dog vil man generelt gerne, ligesom nævnt tidligere, opnå en coverage på 80% for JUnit test af koden også. Årsagen til dette ikke er opnået, har været afdækket i forrige afsnit omkring JUnit test.

7 Projektplanlægning

7.1 Udviklingsproces for forløb

For dette projekt har vi primært haft den samme agile udviklingsproces for udarbejdelse af projektet, som valgt i CDIO-1 og CDIO-2. Her har vi taget udgangspunkt i Unified Process (UP). Som nævnt i indledningen dækker UP over, at opdele hele projektet i mindre delprojekter, kaldet iterationer. Disse iterationer bliver samlet sammen løbende, hvilket vil sige at projektet vil vokse iterativt - altså at projektet vokser lidt for lidt, efter hvert fuldførte delelement. UP kan stilles op i en model, som viser de forskellige faser, discipliner og iterationer for et givent UP-projekt:



Figur 14: UP forløb. [?]

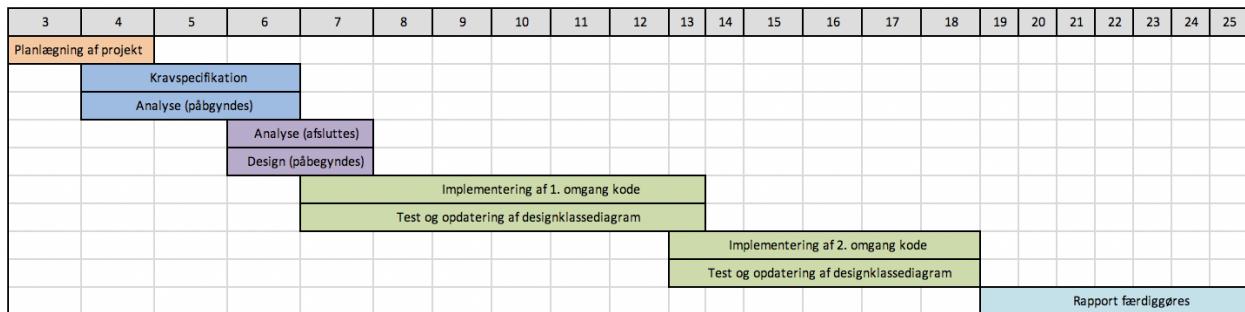
7.2 Planlagte forløb

Vi har efter gode erfaringer i CDIO-2 planlagt, at sætte os ned samlet i gruppen, for at få fastsat datoer for, hvornår hver fase af projektet ønskes færdigt. Her blev vi enige om at inception-fasen for projektforløbet, skulle være færdigt d. 6. november, hvor kravspecifikationer og vigtigste use-case udarbejdes.

Elaboration-fasen for projektet valgte vi, at have deadline d. 7.november. Denne fase dækker over, analyse af systemet afsluttes, design påbegyndes og kodeansvar uddelegeres mellem gruppemedlemmer.

Deadline for konstruktionsfasen af projektet blev vi enige om, at opdele i to mindre iterationer. Den første valgte vi at sætte til d. 13. november, som dækker over nogle væsentlige klasser i systemet, der gør det muligt at arbejde videre på *Game*. Den anden del af fasen blev sat til d. 18. november, som primært dækker over *Game* færdiggøres. I begge

iterationer for konstruktionsfasen planlagde vi at implementering og test af kode samt opdatering af design artifaktor udarbejdes og bliver færdiggjort. Alt dette skulle derudover løbende dokumenteres i en rapport, hvis deadline blev sat til d. 25. november. Dette projektforløb kan derfor opstilles i et Gantt-kort på følgende måde:

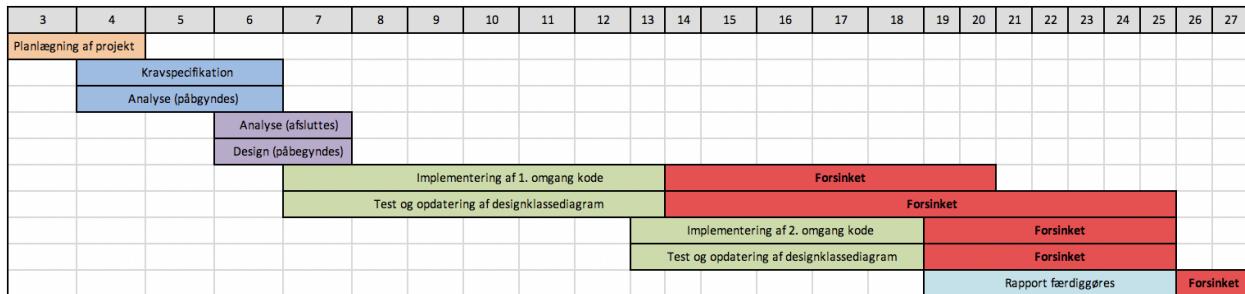


Figur 15: Gantt-kort for planlagte forløb

7.3 Faktiske forløb

I vores faktiske forløb af projektet, havde vi som sagt valgt at have samme fremgangsmåde, som i vores tidligere CDIO-projekter. Det vil sige at alle har været med inde over arbejdet i inception-fasen. Her er opgavebeskrivelsen blevet læst grundigt igennem, der er opstillet kravspecifikationer og påbegyndt vigtigste analyse-elementer. Dette har bl.a. været med til vi tidligt i processen har kunne definere spørgsmål til kunden, omkring kravspecifikationen og opgavebeskrivelsen. Derudover har det også givet anledning til, at skabe en mere intuitiv overgang til design-fasen af systemet. Årsagen til dette skyldes, at alle har været på lige fod i forbindelse med, hvad der forventes af kunden. Der har ikke været nogle betydelige forsinkelser ift. det der blev planlagt i denne fase. En vigtig pointe at have med fra denne del af projektet er, at kunden har været længere om, at vende tilbage på enkelte spørgsmål end forventet. Dette har gjort vi først har kunne afslutte dele af kravspecifikationen til sidst i projektforløbet, hvilket normalt ikke er optimalt og kan have store konsekvenser. Dog har disse krav ikke haft den største påvirkning på vores videre arbejde, men kunne have haft væsentlig betydning, hvis der var tale om mere essentielle krav.

I elaboration-fasen blev analysen af projektet færdiggjort og design blev påbegyndt i fælleskab. Ud fra vores udarbejdede designklassediagram valgte vi, at uddelegerede kodeansvar blandt gruppemedlemmerne. Dette gjorde vi ved, at opstille diverse klasser i en tabeloversigt og for nogle klasser, også korte underpunkter ift. hvad der skulle udarbejdes. Igen kan det ses på Gantt-kortet at tidsestimeringen af fasen holder stik.



Figur 16: Gantt-kort for faktiske forløb

I konstruktion-fasen af forløbet, begyndte der dog at opstå flere komplikationer, end vi havde forventet os. Som nævnt i det planlagte projekt, har vi taget ved lære fra CDIO-2, hvor vi fandt ud af det er en god idé at tage højde for, om der er elementer af kode, der skal være færdiggjort før andet kan påbegyndes. Vi oplevede dog i første del af konstruktion-fasen, den klart største forsinkelse for projektet. Her blev det planlagt at være færdig med diverse implementeringer d. 13. november, men disse endte med at være ufuldendte omkring d. 25. november - altså den dag det faktisk var planlagt, at hele projektet ville være færdigt og afleveret til kunden. Denne forsinkelse har derfor også medført at *Game* har måtte blive udskudt. Heldigvis var det planlagt fra vores side af, at deadline ville være tidligere end afleveringsfristen, så vi har haft tid nok til, at rette op på en håndfuld af de uforudsete problematikker.

Hvad har så været skyld i forsinkelserne?

Der har været flere årsager til forsinkelse af projektet. Et af de første vi støtte på var, at vi i gruppen valgte at arbejde med elementer, som ikke har været på pensum i undervisningen og ikke alle har haft erfaring med før - her er der særligt tale om XML og ArrayList. Denne inddragelse har gjort, at vi i gruppen løbende har skulle forsøge, at sætte os ind i hvordan XML-implementeringen fungerer og finde ud af hvordan ArrayList virker. I forbindelse med ArrayList valgte vi dog senere i projektet, at finde en metode hvorpå dette kunne undgås. Ovenstående har vi skulle gøre parallelt med, at vi har implementeret koden, der forudsætter kendskab til ovenstående. Derudover har vi også samtidigt skulle arbejde på andre projekter i samme kurser og andre kurser. Dette havde vi glemt at tage højde for i projektplanlægningen.

En af de helt store faktorer for forsinkelsen har også været, at der ikke er blevet lagt nok arbejde i præcisionen af, hvad der faktisk skal udarbejdes. Et godt eksempel på dette var, at vi først d. 16. november, blev enige om at definere en *Deck*-klasse til systemet. Dette burde vi allerede i elaboration-fasen havde tænkt over i design-delen af projektet. Dette er dog et rigtig godt eksempel på vigtigheden af, at have en agil udviklingsproces, som har plads til denne sene tilføjelse. Generelt set har der været flere mindre ting, som kunne være

undgået, ved et bedre og mere fyldestgørende design.

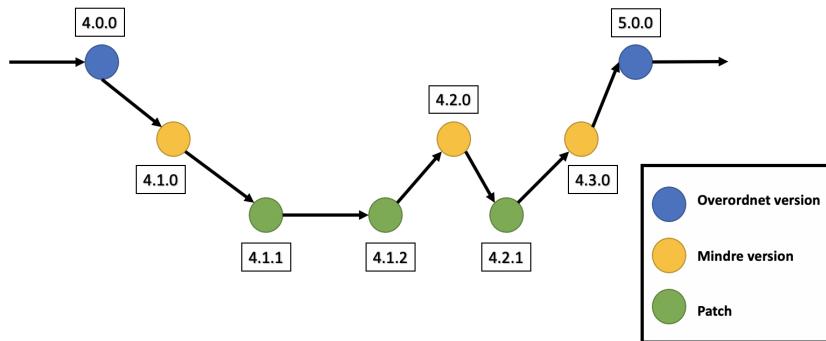
Set på den positive side, har vi dog været gode til at opstille møder, når der har opstået større problemer. Til disse møder har vi bl.a. talt om, hvad der kan gøres for at vi kan nærmest muligt af vores projekt - her valgte vi ofte at udskyde deadline, da vi har haft tid nok i sidste ende. Generelt set har vi rådgivet hinanden undervejs og i de værste tilfælde prioteret, hvilke funktioner der skulle fravælges. I forlængelse af dette kan det ses, at arbejdsindsatsen for gruppe, har været noget højere end tidligere projekter[?]. Her har det vist sig, at vi i gennemsnittet har arbejdet ca. 30timer pr. gruppemedlem i CDIO2 og 47timer pr. gruppemedlem i nuværende CDIO3. Det vil sige, at den gennemsnitlige arbejdsindsatsen i timer, er steget med lidt over 50% ift. CDIO2. Dette er en forhøjelse, som tydeligt viser, at der har været mere pres og mere tid lagt i projektet. På den anden side kan dette også bruges til, at vise vi måske ikke altid har været lige effektive med den tid, som er blevet brugt. Denne erfaring kan f.eks. perspektiveres over på Søren Lauesens artikel[?], som berører emner som dette. I artiklen bliver der bl.a. nævnt under cause F3 og G2, at projekter generelt ofte kan mislykkedes, grundet mangel på kompetencer og planlægning. I dette projekt har det eksempelvis været metoder, som vi ikke har haft erfaring indenfor og nogle gange udskudt opgaver til for sent hen i projektet. En vigtig pointe at have med er, at vi generelt set har været gode til, at fordele vores timer uddover hele projektet, og ikke kun til sidst i en slutsprint.

Vi har derfor fremadrettet lært, at bruge mere tid i udarbejdelse af design, så disse tanker bliver så vidt muligt udtaenkt inden de bliver forsøgt kodet. Derudover vil en grundigere udarbejdelse af designklassediagrammet og sekvensdiagrammer også gøre, at alle i gruppen har en bedre idé om associationerne mellem klasserne og sammenspillet mellem metoderne.

8 Konfiguration og Versionstyring

For nemt at kunne holde styr på platformens dele, ville det være smart at de forskellige platforme vi bruger bliver versioneret. Dette dækker over Java, operativt system, IntelliJ samt GUI biblioteket, der hentes fra Maven.

Generelt er en god måde at versionere på, at give sine forskellige iterationer versionsnumre, og så tælle op i versionsnummeret hver gang der kommer opdateringer af programmet. Hvis vi nu har et produkt med versionsnummer 4.2.1, hvor 4 tallet indikerer hvilken overordnede version der er tale om. 2 tallet er hvilken mindre version det er og 1 kunne være hvilket patch(bugfixes eller små rettelser) der er tale om. Normalt vil version 0.1 være en eller anden form for beta, og version 1.0 vil så være første version af produktet.



Figur 17: Version nummer eksempel

Normalt når der laves en stor ændring og der går fra version 1.0 til 2.0 vil versionen ikke være baglæns kompatibel. Som regel hvis den overordnede version er den samme, vil mindre ændringer være bagud kompatible. Iblandt de store ændringer kan der laves en masse mindre ændringer eller patch, hvilket er forsøgt illustreret i figur x.x (Den længere oppe)

Der er desværre ikke i dette projekt blevet gjort brug af versionering. Gruppen som helhed har desværre været alt for fokuseret på at få lave et godt design der lagde fundament for til at udvikle solid kode der virkede, hvor fokus helt klart primært var på koden.

Dokumentation for projektet, foregår vha. Overleaf, der bliver synkroniseret med GitHub, nye og gamle versioner kan findes via. GitHub link i bilag.

I GitHub, er det muligt at lave et tag der indikerer versionsnummer. Det ville have været smart at versionsnummere passer sammen for IntelliJ samt dokumentationen i Overleaf. Det blev dog erfaret at det umiddelbart var svært at finde tidlige versions numre hvis de laves som et tag. Så bedste løsning til at versionere i IntelliJ, så ud til at være at lave et commit med det nuværende versionsnummer, og så have en liste over versions numre, som der så kan søges efter i IntelliJ Git menuen. Udover det kunne det være en ide, i listen over versions numre, også lige at lave en kort beskrivelse når der blev lavet store ændringer, så det er hurtigt og nemt at se hvilke versioner bestemte features er blevet added.

Artifakter såsom systemsekvensdiagrammet, Use-cases, kravliste, domænemodel osv. findes alle sammen i nyeste version i dokumentationen på Overleaf eller GitHub, samt er det muligt at finde ældre versioner i tidlige commits. Det er desværre ikke nemt for dette projekt, da der skal rødes mange commits igennem for at finde et bestemt artifikat.

En måde hvorpå man kan sikre sig at man ikke har forskellige versioner af artifakter, samt dokumentation og kode, er at undgå at give versionsnumre separat, til de forskellige plattorme. Det virker ikke som den bedste ide, at man har et artifikat med versionsnummer 2.0.22 som passer til kode version 5.0.11, i så fald skal man have udarbejdet en eller anden oversigt over hvilke versioner der passer sammen.

I stedet kunne det være en ide at sørge for at hele projektet overordnet har versionnumre. Hvor version 5.2.10 for hele projektet indeholder artifakter kode osv. med samme versions nummer. Der er ikke umiddelbart nogen soleklar måde at imødekommne dette problem. Det er dog et essentielt problem at finde en god overskuelig løsning på, da de vil gøre det meget nemmere at finde versioner af artifakter, kode samt dokumentation der passer sammen.

Gruppen ser tilbage på konfigurations delen af opgaven. Hvor der vurderes at vi har haft forståelse og forudsætninger for at opfylde -alle krav til denne del. Det er desværre grundet mangel på overblik, højest sandsynligt pga. tidspres, ikke blevet gjort til en tilfredsstillende grad. Gruppen noterer sine fejl, og skriver det bag øret til næste gang.

9 Import af Git Rep og Start Spil

For at kunne spille spillet "Monopoly Jr.". Anbefales det at benytte en pc med Windows 10 installeret, som har installeret og kan køre IntelliJ Idea. Programmet skal åbnes vha. IntelliJ

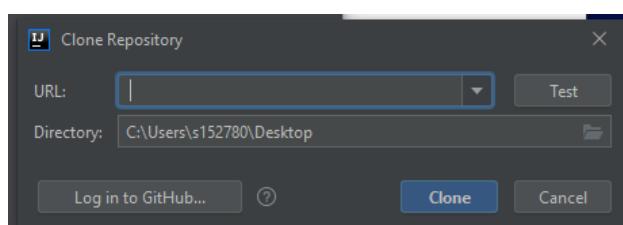
Nedenfor er en beskrivelse af hvordan programmet køres med IntelliJ:

9.1 A Beginners guide to opening Monoploy Jr.

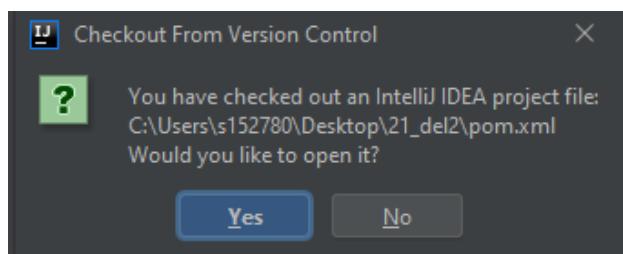
- 1) Åben IntelliJ og tryk på Check out from Version Control



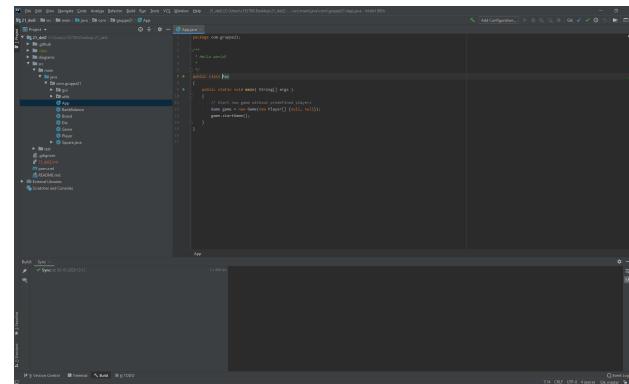
- 2) Indtast https://github.com/Gruppe-21/21_3.git i URL og tryk clone



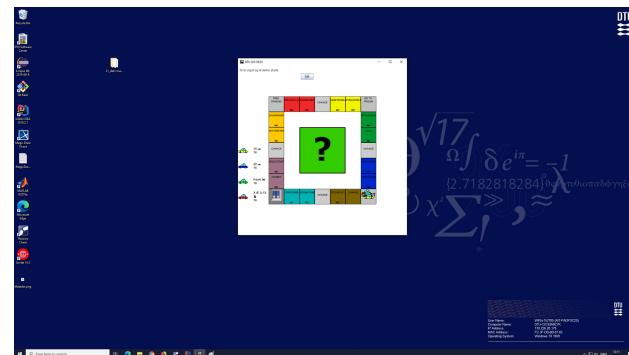
- 3) Tryk på yes



4) Naviger til App mappen som vist på billedet



5) Tryk på alt + shift + f10 og tryk enter og spillet burde virke



10 Konklusion

Der er for projektet opstillet en kravspecifikation, som dækker kravene fra kundens ønsker til Monopoly Junior spillet. Kravene er anvendt som rettesnor gennem projektets arbejde.

Krav om at spillet skal spilles af 2-4 spillere, en brik til hver spiller, 20 chancekort, en enkelt terning samt en spilleplade i et array, har vi implementeret. Spillerne kan vælge deres egen figur ved spilstart, hvorefter yngste spiller starter. Ved start af spillet, tildeles en vis mængde Å afhængig af antallet af spillere, og selve spillet fungerer som bank. Vi har været i stand til at anvende dele af tidligere CDIO-projekters produkter, hvilket har givet os et skelet at arbejde udfra.

Produktets funktionelle krav er afdækket vha. formelle testcases, herunder hastighedstest, JUnit tests (gennem Maven) til test af både klasser og deres metoder, og dokumenteret disse med bl.a. screenshots. Der er foretaget brugertest af en udefrakommende person fra målgruppen, ubekendt med programmering. Derudover er der blevet testet på en maskine i DTU's databar. Til brug for versionsstyring tages højde for forskellige versioner gennem Git og branches. Der er til projektet udarbejdet en vejledning i import af Git repo til IntelliJ. Klassen *Game* afvikler selve spillet fra start af en runde, hvor spiller kaster terningen og rykker sin figur frem, og afhængigt af om det er en ejendom, chancefelt eller startfeltet der passerer på, skal spilleren enten købe ejendommen, trække et chancekort el. modtage Å. Lander spilleren på et felt ejet af en anden spiller, skal der betales til denne, og ejer personen begge ejendomme i farven, betales dobbelt. Der findes endvidere et fængselsfelt, som spillere kan ende på, enten fordi de er i fængsel eller på besøg. Ved start af spillet vælges ml. simple og avancerede regler, som er to forskellige versioner i kompleksitet. Ifm. implementeringen har vi valgt at anvende IntelliJ som udviklingsværktøj med tegnsætning UTF-8. Gennem projektet har vi arbejdet med forskellige analytiske modeller, bl.a. use case diagram, domænemodel, systemsekvensdiagram, sekvensdiagram og designklassediagram. Vi har igennem disse modeller klarlagt bl.a. ansvarsområder og associationer. Opbygningen af spillet er lavet således, at tekster hentes fra XML-filer, og derved er det enkelt at oversætte spillet. Spillet er oversat til flere sprog; herunder dansk og engelsk. Nedarv er anvendt i flere af klasserne, hvor en parent klasse har subklasser under sig. Koden til spillet er designet efter GRASP principperne, således at hver klasse har high cohesion. Klasserne anvender konstruktører til deres attributter, og bl.a. metoderne get, set, og toString de relevante steder. Vi har endvidere anvendt den ønskede GUI fra Maven repository. Koden er delvist blevet tildelt løbende versionsnumre.

Vi er gennem projektet stødt på udfordringer, som har medført ikke alt blev lavet som ønsket. Vi har oplevet tidspres, og længere implementeringstider. Endvidere har vi oplevet, at kravspecifikationen blev justeret undervejs, hvilket har medført det blev svære at opfylde alle krav. Vi har gennem projektet lært, at vi må arbejde på at få enkelte elementer i projektet færdige.

Afrundningsvis kan det konkluderes trods mangler, at kravspecifikationen overordnet set er blevet afdækket, således at kundens produktønske og udviklingen er endt tilfredsstillende.

11 Bilag

11.1 Link til GitHub

https://github.com/Gruppe-21/21_del3.git