Danmarks Tekniske Universitet

Link til Projekt GitHub

CDIO Final - Gruppe 22


Mads Storgaard-Nielsen
s180076

Sander Eg Albeck Johansen
s195453


Jonas Løvenhardt Henriksen
s195457

Frederik Verne Henriksen
s173394


Mikkel Hillebrandt Thorsager
s195470

Alexander Lambrecht
s195482

1 Intro

1.1 Abstract

As computers and smartphones become more and more part of everyday life, many objects, whether it be calendars, clocks, or letters, have made their transition to electronic devices. This advancement has made it possible to contain many objects in just one, and even traditional board games are making their way onto computers. This paper addresses the process of creating the classic board game "Monopoly", to IOOuteractive, a fictive costumer, from the specification of their requirements to the operation of these. UML Artifacts are used to help analyse and design the code structure and its implementation. The source code is written in Java 8, and the accompanying tests uses the JUnit 4 library and usertest. The code is inspired by the model-view-controller principles in it, so that the classes reach a level of high cohesion and low coupling, and this also ensures it's reusability in future projects. The final product is a board game with a GUI, which fulfills significant requirements from the costumer.

1.2 Indledning

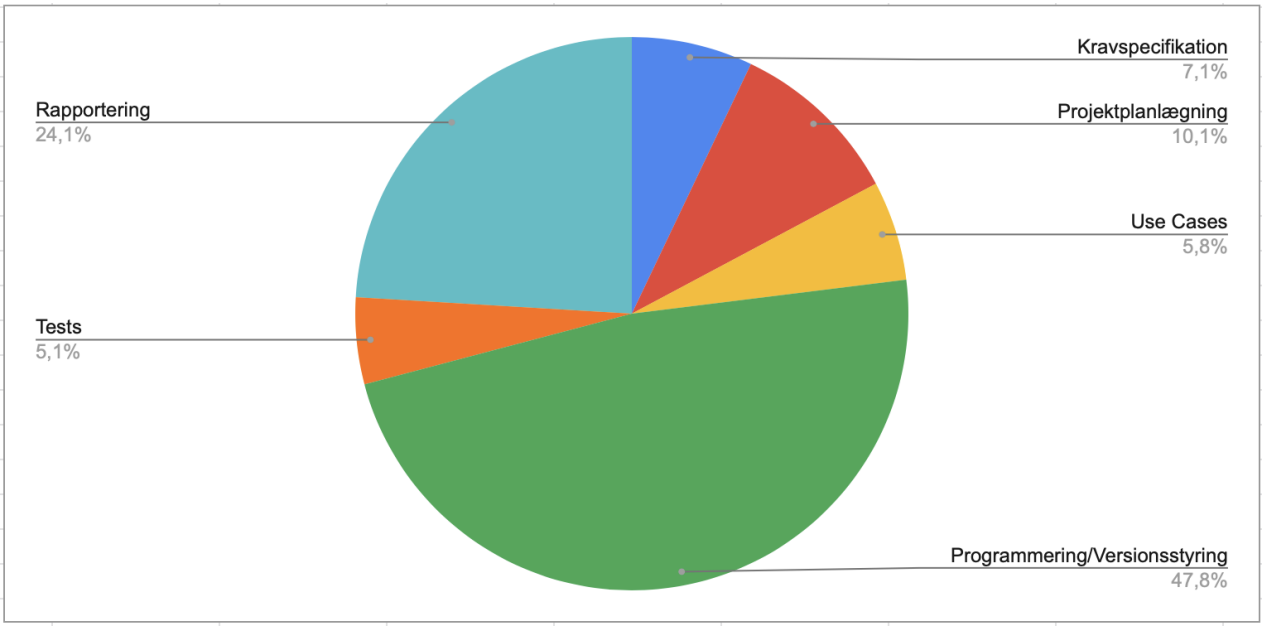
IOOuterActive har givet os en sidste opgave, design et komplet Matador spil. Vores gruppe har brugt Unified Process udviklingsprocessen til at udarbejde spillet, hvilket vi vil uddybe i opgavens afsnit "projektforløb". Vi har bevidst forsøgt at udarbejde projektet efter analyse- og designmønstre, så som GRASP, for at følge universelt anerkendte kodestandarder. Programmet er også illustreret ved brug af UML, samt distribuelt versioneret med Git, da dette er almen praksis ved softwareudvikling.

I denne rapport har vi arbejdet videre på vores tidligere projekter, hvor vi har skabt et terningspil, og et junior matador spil. Dette spil vil adskille sig fra junior matador, ved at spillet ikke længere kan spilles automatisk, da man skal foretage bevidste valg for at kunne spilles. Disse valg kan være om man vil købe en grund eller ej, om man vil betale penge for at slippe ud af fængslet eller blive ved med at slå med terningerne, om man ville opgradere sine grunde med huse og hoteller. Dette gør spillet markant mere komplekst at programmerer, derfor får vi meget mere brug for at holde et overblik end tidligere opgaver.

2 Timeregnskab

2.1 Arbejdsområde

Arbejdsområde	Sander	Jonas	Mads	Mikkel	Frederik	Alexander	**I alt**
Projektplanlægning	8	6	6	6	8	6	**40**
Kravspecifikation	6	4	4	4	6	4	**28**
Use Cases	9	2	3	2	3	4	**23**
Programmering/Versionsstyring	16	36	27	34	35	37	**189**
Tests	4	6	6	6	2	6	**30**
Rapportering	25	10	15	15	15	15	**95**
Total	**68**	**64**	**61**	**67**	**69**	**72**	**405**



2.2 Timeforbrug pr. dag

Dato	Sander	Jonas	Mads	Mikkel	Frederik	Alexander
6/1-2019	2	2	2	2	2	2
7/1-2019	5	5	5	5	5	5
8/1-2019	6	6	6	6	6	6
9/1-2019	2	6	6	2	6	4
10/1-2019	6	5	5	8	6	6
11//1-2019	-	-	-	-	-	-
12/1-2019	-	-	-	-	-	-
13/1-2019	6	6	3	8	6	6
14/1-2019	6	6	5	8	6	6
15/1-2019	6	7	7	7	6	6
16/1-2019	8	7	7	7	8	8
17/1-2019	7	7	-	7	7	7
18/1-2019	-	-	-	-	4	6
19/1-2019	9	2	10	2	2	5
20/1-2019	5	5	5	5	5	5
I alt	68	64	61	67	69	72

Indholdsfortegnelse

3 Kravspecifikation

3.1 Vision

Med udgangspunkt i Matador Junior ønsker kunden IOOuteractive at vi udvikler et fuldt funktionelt matadorspil. Dog ønsker IOOuteractive kvalitet over kvantitet, og forventer at væsentlige features er implementeret og virker. IOOuteractive forventer et spil hvor hver spiller kan træffe valg, og derved påvirke spillets gang.

3.2 Krav

Kravene er inddelt i kategorier efter features, og herefter prioriteret efter MoSCoW principperne.

(M) = Must Have

(S) = Should Have

(C) = Could Have

(W) = Won't Have

3.3 Kravliste efter kategorier

- Basale Brætspil Funktioner:

  R1: (M) Der er en spil plade
  R2: (M) Der skal kunne være 3-6 spillere
  R3: (M) Man skal kunne indtaste sit navn
  R4: (M) Man skal kunne bevæge sig rundt med uret på spilpladen
  R5: (M) Når du passerer start får du 4000 kr.
  R6: (M) Man har to seks øjet terninger man slår, udfaldet flytter dig på brættet
  R7: (S) Hvis man slår to ens, må man slå igen
  R8: (C) Hvis man slår to ens, tre gange i træk ryger du i fængsel
  R9: (S) Man vælger mellem forskellige typer brikker for spillerne som UFO, biler, skibe.

- Køb af Ejendomme:

  R10: (M) køb af grund
  R11: (S) dobbelt husleje når alle felter i farve ejes
  R12: (S) ekstra husleje ved huse eller hotel
  R13: (S) betal ekstra ved eje af flere færgelinjer
  R14: (S) Betal antal af øjne gange 100 ved bryggerier
  R15: (S) ved eje af begge bryggerier skal man i stedet betale 200 gange antal af øjne man slår
  R16: (S) Bygge huse/hoteller
  R17: (S) Når du har 4 huse, og køber endnu et, bliver det til et hotel
  R18: (C) Auktion hvis en spiller vælger ikke at købe grunden
  R19: (W) Pantsætning[1]

[1] Det har ikke lykkedes os at få implementeret vores krav om pantsætning af ejendomme.

- Chancekort:

  R20: (S) Når man lander på et prøv lykken felt, trækkes et lykkekort, og handlingen på kortet udføres.

- Indkomstskat:

  R21: (C) Betale indkomstskat

- Fængsel:

  R22: (S) Man kan ryge i fængsel
  R23: (S) Du kan komme ud ved at slå to ens
  R24: (S) Du kan maks sidde i fængslet i 3 ture i træk
  R25: (C) Kan komme ud af fængsel ved at betale 1000 kr.
  R26: (C) Kan købe grunde i fængslet

- Konklusion af Spil:

  R27: (M) Mangler penge til at betale funktion, gå fallit
  R28: (C) Vælg hvilke grunde/boliger du vil sælge
  R29: (M) Vinder funktion

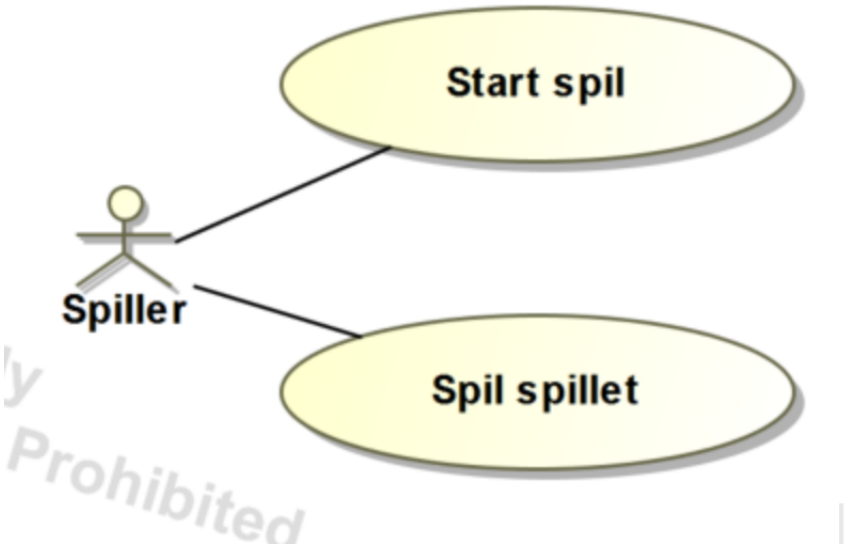
- Non-funktionelle krav
  - Systemet skal have en hurtig responstid.
  - Systemet skal være let at bruge
  - Programmet skal være let at vedligeholde.
  - Programmet skal kunne køre på almindelige computere.
  - Programmet skal nemt kunne oversættes til andre sprog
  - Koden skal kunne genbruges[2]

[2] `https://en.wikipedia.org/wiki/Non-functional_requirement`

4 Analyse

Vi starter ud med at gennemlæse en matador regelbog og analysere et matadorbræt, for at udarbejder en krav liste, denne vil starter ud med ikke at være prioriteret, og senere blive inddelt i prioritering efter MoSCoW metoden med et Bogstav foran sig, samt en kategori til at vide hvor de høre til i spillet. Kategorien hjælper med at huske alle de krav, man skal huske når man designer og implementerer et krav. MoSCoW opdelingen vil give os et godt overblik over hvilke krav vi skal arbejde med i vores iterationer, som vi har smidt ind i en arbejdsplan vi har lagt i bilagene. Vi besluttede at arbejde efter Unified process, da vi er opmærksomme på at vi ikke er i stand til at udarbejde så perfekt en plan at vi kan følge en vandfaldsmodel. Så unified process vil sikre at vi kan ændre i vores design, fx. hvis vi får ideer til hvordan vi kan optimere vores design.

4.1 Use Case Diagram



Her ses spillets to use cases.

4.2 Use Cases

U1 Start Spillet

- Ved start af spillet vælges hvor mange spillere der er, derefter enkeltvis vælger man sin brik og indtaster dit navn. Spillerne indtaster deres navn og vælger brik i rækkefølgen yngst til ældst. Efter dette modtager hver spiller 30000 kr.
  - U1A Spilleren der starter kaster terningerne og udfører sin tur.

U2 Spil Spillet

- Spilleren rykker deres bil frem til et felt baseret på et terningekast, alt efter hvilket felt der landes på, er der forskellige udfald.

4.3 Use Cases - fully dressed

U1 - Start Spil

Use Case: Start Spillet	
ID	U1
Level	Spillet skal begynde og spilleren skal være parat til at starte U2
Primær aktør	En spiller
Stakeholders	IOOuterActive og spilleren
Precondition	Spilleren skal have åbnet programmet og være klar til at spille
Postcondition	Alle spillere indtaster navn.
Basic flow and extensions	1. Programmet starter 2. Spilleren bliver bedt om at vælge antal spillere 3. Spilleren bliver bedt om at indtaste sit navn 4. Spilleren indtaster sit navn 5. Spilleren bliver bedt om at vælge en brik. 6. Spilleren får 30000 kr hver.
Frequency of occurence	Dette er use casen som starter programmet, og vil derfor ske en gang pr. spiller.

U2 - Spil spil

Use Case: Spil Spillet	
ID	U2
Level	For hvert terningekast lander spilleren på et bestemt felt, og ud fra feltet
Primær aktør	Spilleren
Stakeholders	IOOuterActive og spilleren
Precondition	Spilleren skal have udført usecase U1
Postcondition	Når terningerne er slået lander spilleren på et felt, informationen fra dette felt gives til spilleren
Basic flow og extensions	1. Man slår med 2 seks sidede terningerne 2. Spilleren rykker det antal øjne frem ad på spilpladen 3. Spilleren lander på et felt • Spilleren lander på et ikke-ejet købbart felt – Hvis spilleren lander på et ejendomsfelt, kan man vælge at købe det eller sætte det på auktion. – Hvis man lander på en grund-felt, kan man vælge at købe det eller at sætte det på auktion

Use Case: Start Spillet	**Fortsat**
Basic flow and extensions	• Spilleren lander på et ejet købbart felt – Hvis det er et bryggeri, skal man betale med det antal af øjne man slår med en sekssidet terning og ganger ∗ Hvis ejeren af bryggeriet kun ejer et bryggeri, skal man gange antallet af øjne med 100. ∗ Hvis ejeren af bryggeriet kun ejer begge bryggeri, skal man gange antallet af øjne med 200. – Hvis det er et rederi, skal man betale et beløb udfra hvor mange af de 4 andre færgelinjer som den spiller der ejer feltet du landede på har. ∗ Et rederi 500 kr ∗ Et rederi 1000 kr ∗ Et rederi 2000 kr ∗ Et rederi 4000 kr – Hvis spilleren lander på et ikke købbart felt. ∗ Hvis spilleren lander på fængsletfeltet, sker det intet ∗ Hvis spilleren lander på parkeringsfeltet, sker der intet ∗ Hvis spilleren lander på prøv lykken, skal de trække et chancekort og følge det · Bevæg sig hen til et specifikt felt · Få penge · Giv penge til banlken eller de andre spillere · Gå-Ud-af-fængslet kort ∗ Hvis spilleren lander på Start eller passere start får de 4000 kr ∗ Hvis spilleren lander på Gå-i-fængsel feltet, skal de gå i fængsel, hvis de passerer start får de ikke 4000 kr. · Ved starten af spillerens næste tur, kan de betale 1000 kr for at komme ud · Ellers skal spilleren kaste med med 2 seksidet terninger og få to ens for at komme ud. · Når spilleren har kastet 3 gange, uden at få nogle ens må de ved næste tur, komme ud af fængslet. · De kan brug deres slip ud af fængslet kort, hvis de har det og slippe for fængslettid – I løbet af en spillers tur kan spillerens tur kan de altid vælge at pantsætte, købe hus eller opgraderer til hoteller. – En spiller løber tør for penge ∗ Kan vælge et pantsætte sine værdier ∗ Sælge sin Værdier ∗ Kan Ikke pantsætte eller sælg noget, går spilleren fallit og overdrever sin resten værdier til banken og hvis spilleren ikke kunne betale husleje, giver man dem til den spiller.
Frequency of occurrence	Det er programmet primære use case, og derfor vil den optræde altid ved brug af programmet, bortset fra når du starter programmet.

4.4 Risikoanalyse

P/I Matrix

Probability/Impact Matrix						
Omfang(forventede konsekvenser)		Unlikely	Seldom	Occasional	Likely	Frequent
Katastrofale: Uacceptable fejl, Ufungerende spil, Spil ødelæggende bug, kernefunktionalitet virker ikke		M	M	H	EH	EH
Kritiske: Spillet er ikke intuitivt, svært at forstå uden forklaring, Must have krav ej implementeret Krav implementeret, men virker ikke Kernefunktionalitet ej testet		L	M	H	H	EH
Moderat: GUI Bugs(navne ej vist, tager ikke imod input navnene matcher ikke det indtastede, ingen begrænsning på antal af spillere, forkert startbalance)		L	L	M	M	H
Ubetydelige: Logiske fejl, der ikke påvirker spillet		L	L	L	L	M
Legend: EH - Extremely high risk, H - High risk, L - Low risk, M - Medium risk.						

Risiko faktorer

Risikofaktor	Probability	Impact
Terning ej funktionel	Unlikely	Catastrophic
Chancekort ej funktionelle	Seldom	Moderate
GUI bug(unplayable)	Seldom	Catastrophic
GUI bug(playable)	Occasional	Moderate
Kan ikke købe grunde (fejl ved implementation)	Occasional	Severe
Kan ikke købe huse/hoteller (fejl ved implementation)	Occasional	Catastrophic
Spiller kan undlade at bygge "jævnt"(fejl ved implementation)	Occasional	Severe
Fejl ved betaling af Grund/Ejendom/Huse/Hoteller	Seldom	Catastrophic
Specielle felter ej funktionelle (Fængsel, rederi, bryggeri, indkomstskat)	Occasional	Severe
Auktion ej funktionel	Likely	Negligble
Spiller får ikke 4000, ved at passerer start	Unlikely	Severe
Logik omkring dobbeltslag ej funktionel	Unlikely	Severe
2 eller flere spillere får ens biler	Occasional	Catastrophic
Fejl ved huslejeudregning	Occasional	Catastrophic

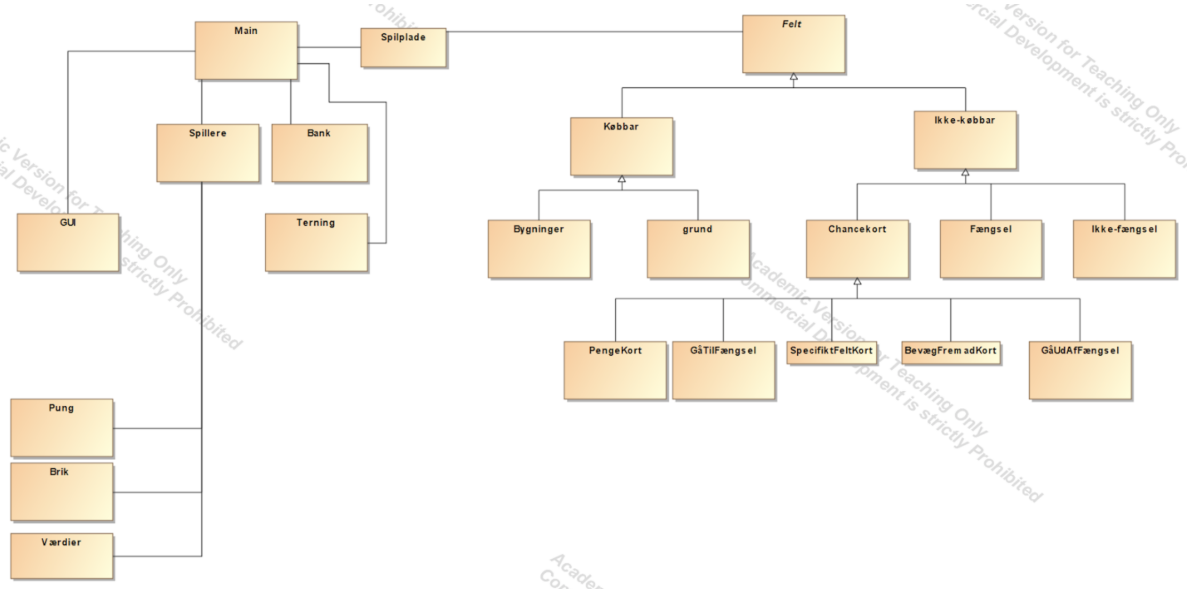
For at forebygge eventuelle risici har vi sørger for at teste vores kode, og så vidt muligt fosøgt at overholde grasp principperne, så ændringer i klasserne ikke vil havde indflydelse på andre klasser end den der ændres i.

4.5 Navneordsanalyse

Ud fra vores use cases og use case-diagrammet, har vi lavet en navneords analyse, og kommet med følgende forslag til klasser i vores program.

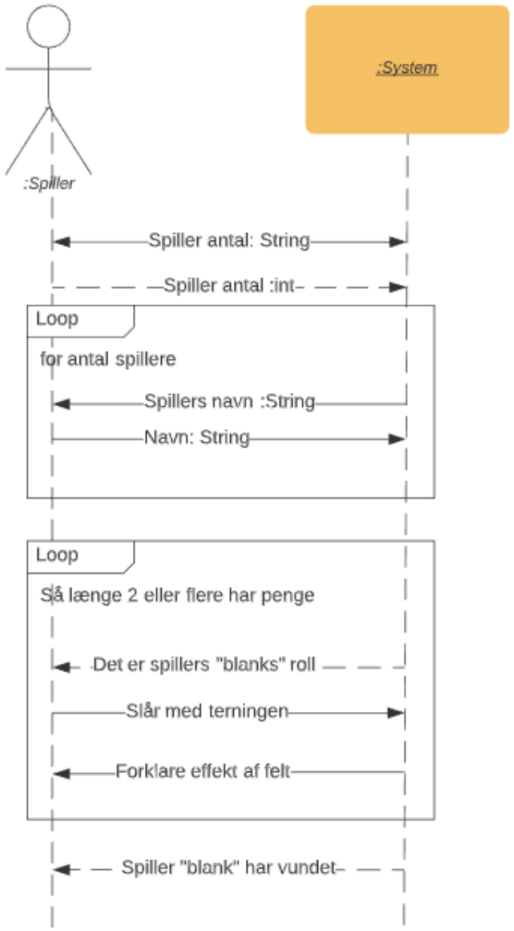
- Spiller
- Spilleplade
- Chancekort
- Felt
- Pung
- Terning
- Brik
- værdier
- fængsel
- Grund
- Ejendom
- Tur

4.6 Domænemodel



Her ses vores domænemodel over matador spillet, dette var vores første udkast til systemtets opbygning, dette diagram er blevet forbedret igennem flere iterationer, inden det er endt som design klasse diagram.

4.7 System sekvensdiagram

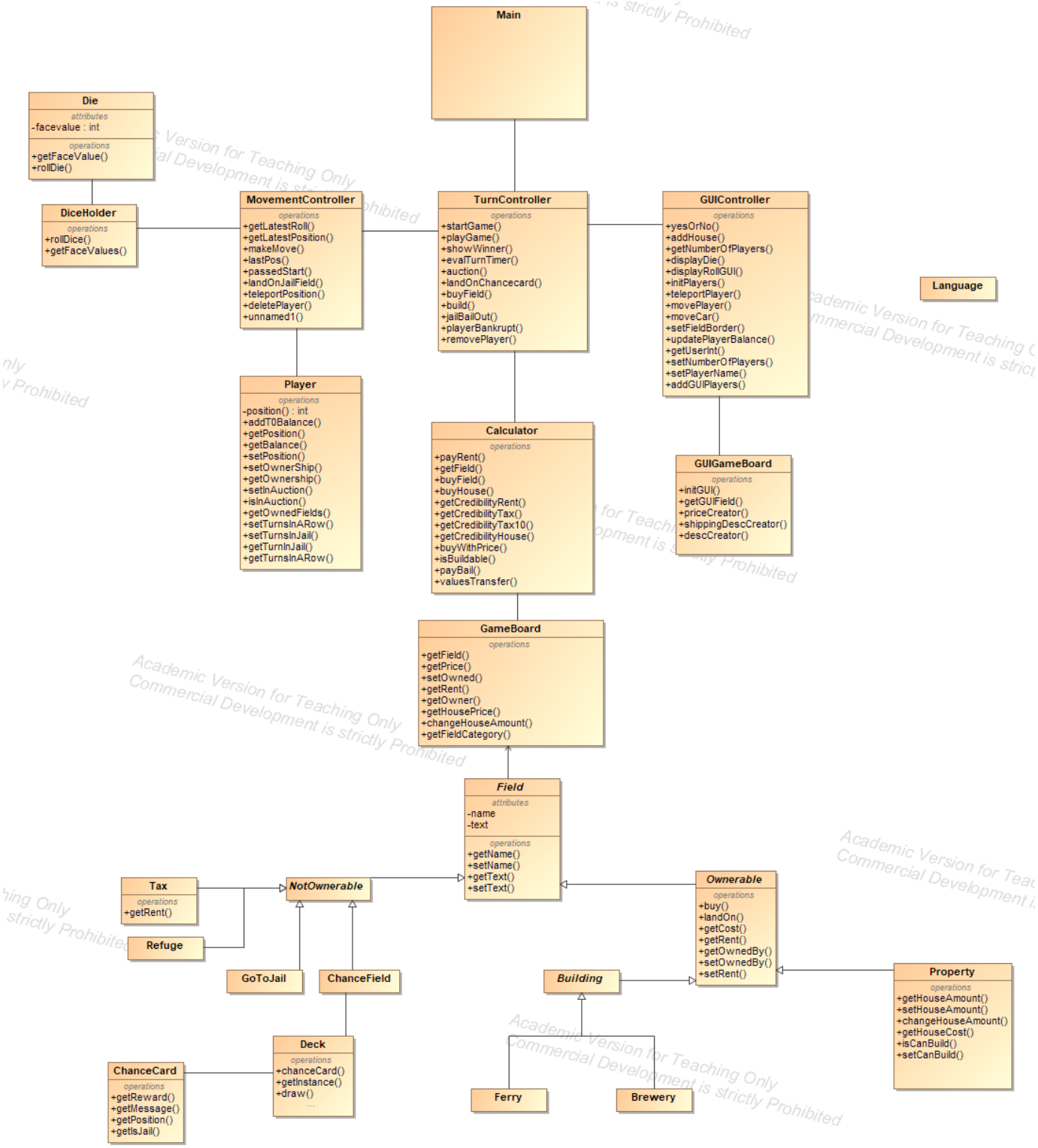


Her ses et system sekvens diagram, over spillets forløb, spillet venter på at spillerne indtaster antallet af spillere, hvorefter en løkke kører der lader spillerne indtaste deres navne, når navnene er indtastet starter spillet. spillets løkke kører så længe at 2 eller flere spillere har midler nok til at spille imod hinanden, når der kun er en spiller tilbage med nok midler, erklæres denne spiller som vinder.

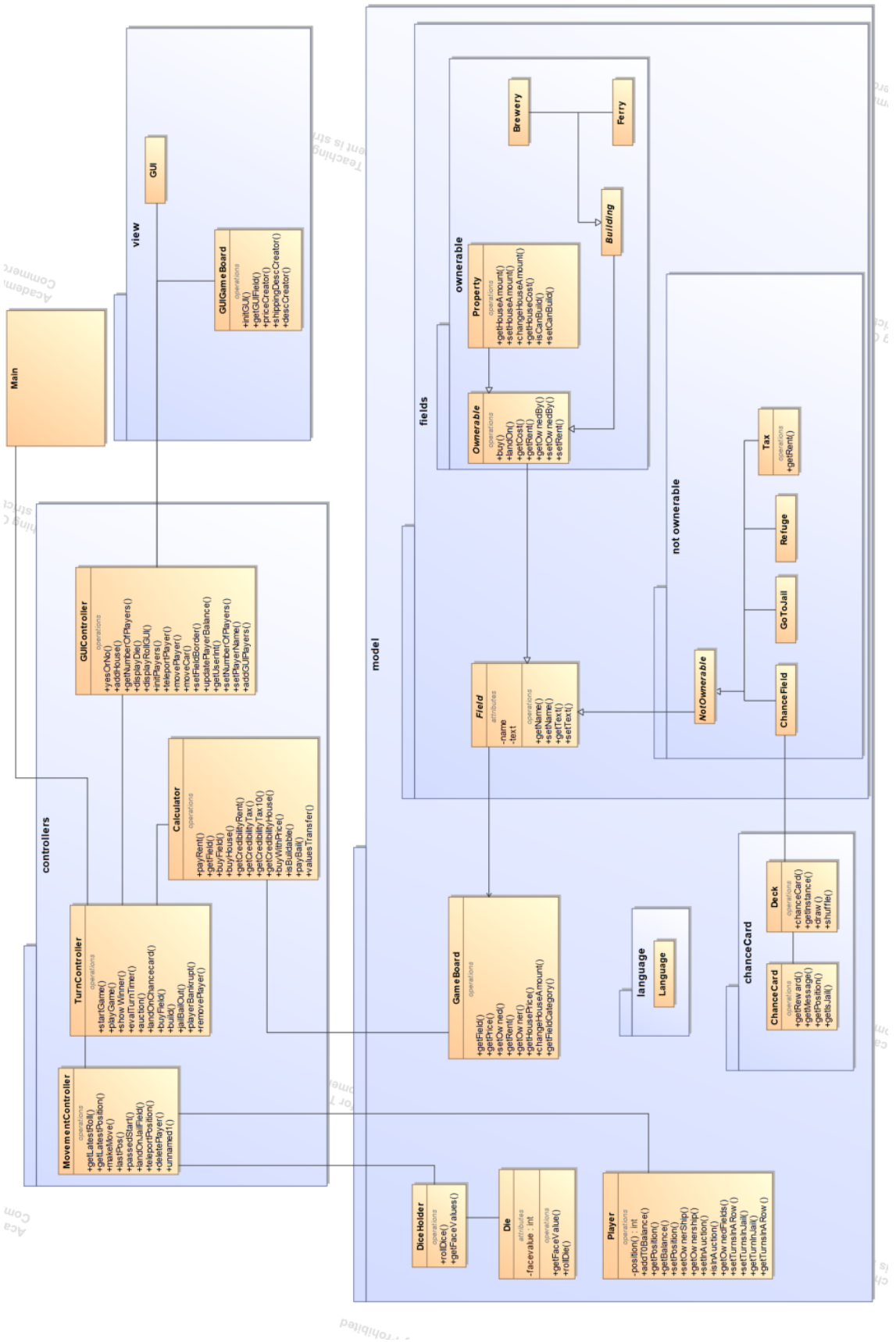
5 Design

Nedenfor ses vores Design klassediagram. Diagrammet har ligesom programmet gennemgået flere iterationer og er blevet "forbedret"løbende, efter vi er stødt ind i designmæssige udfordringer. Vores diagram var markant simplere i analyse fasen, men da vi arbejdede efter unified process, fik vi løbende forbedret vores diagram og derved vores programarkitektur. Vi har brugt diagrammet til at skabe et overblik over programmets klasser, kobling og samhørigheden. Vi har løbende kunne fjerne overflødige klasser samt ikke-brugte metoder.

5.1 Design klasse diagram



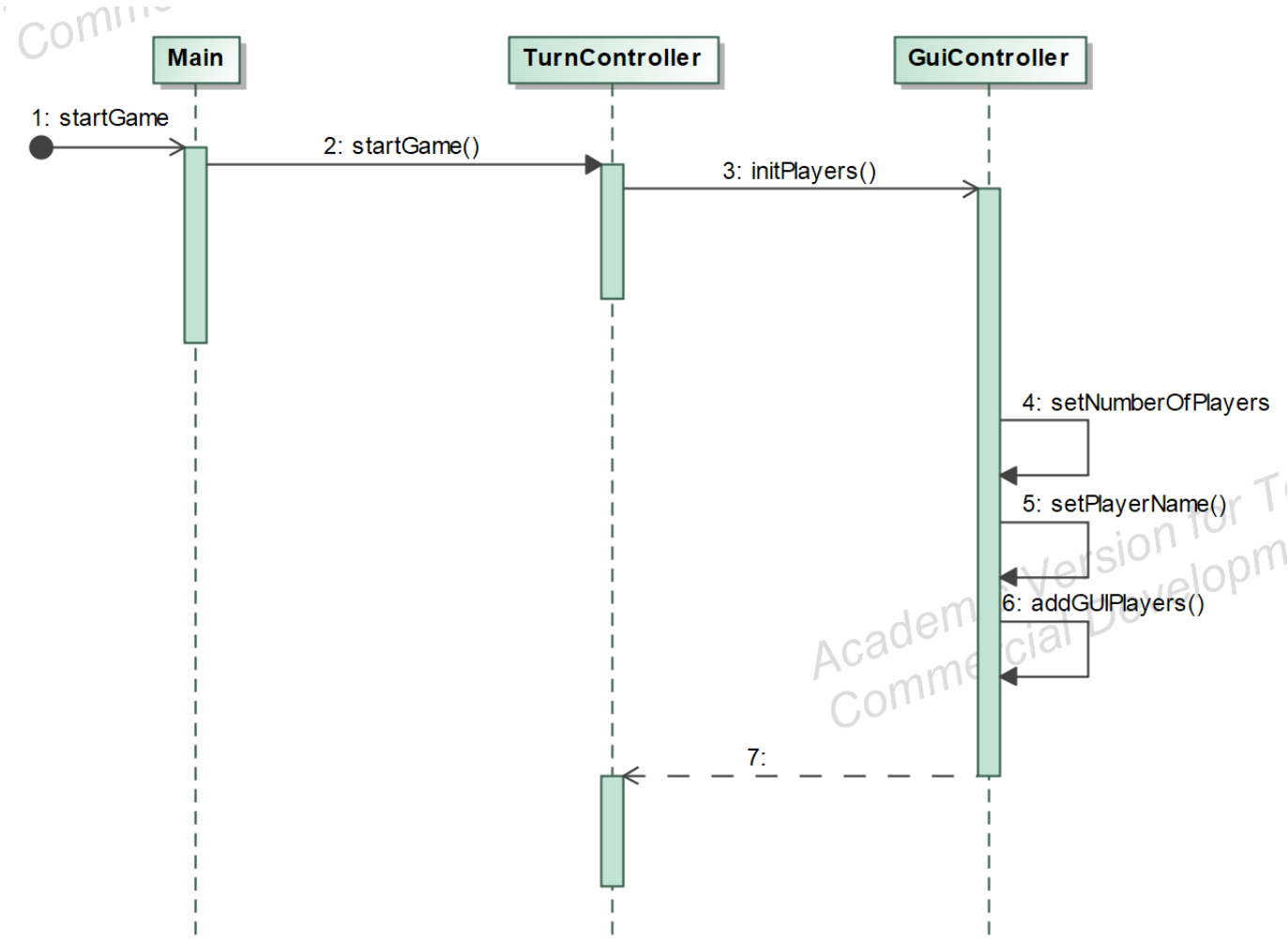
5.2 Pakkediagram



Ovenfor ses vores pakkediagram

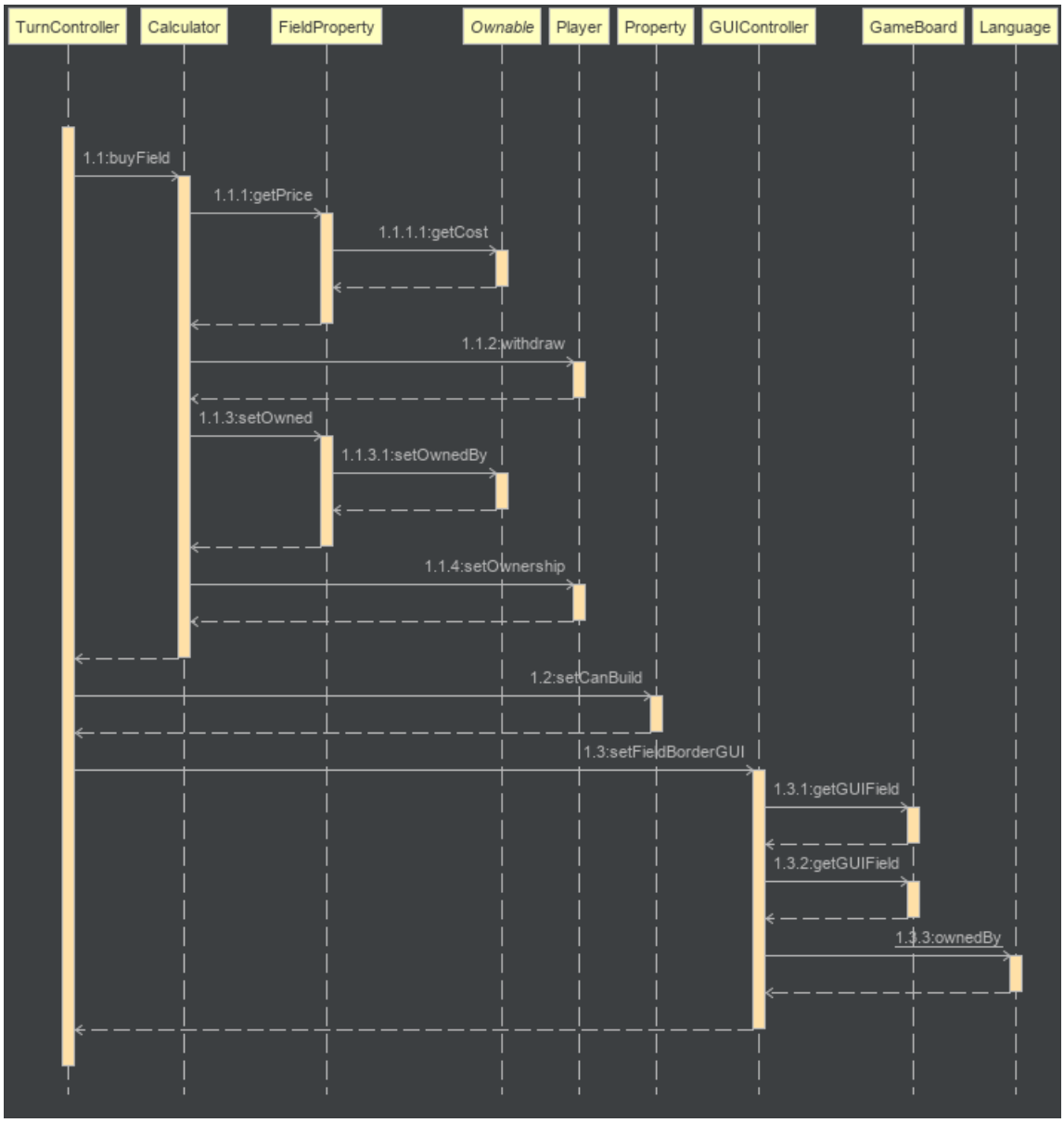
5.3 Sekvens Diagram

startGame



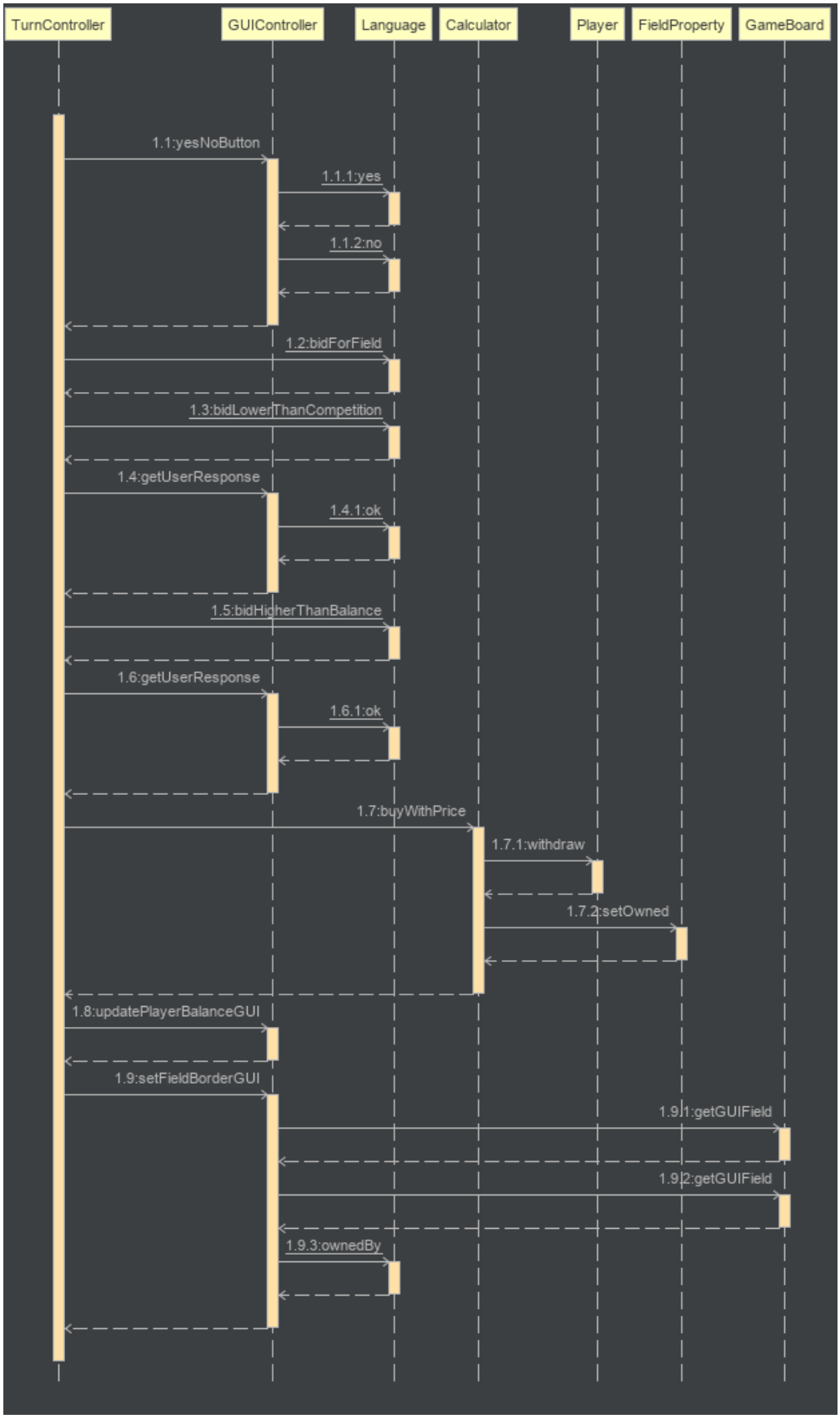
Et diagram over de metode kald der foretages når spillet startes. Vores TurnController, kalder initPlayers() på vores GUIController, som opretter spillerne i vores GUI, så spilleren kan se disse på brættet.

buyField



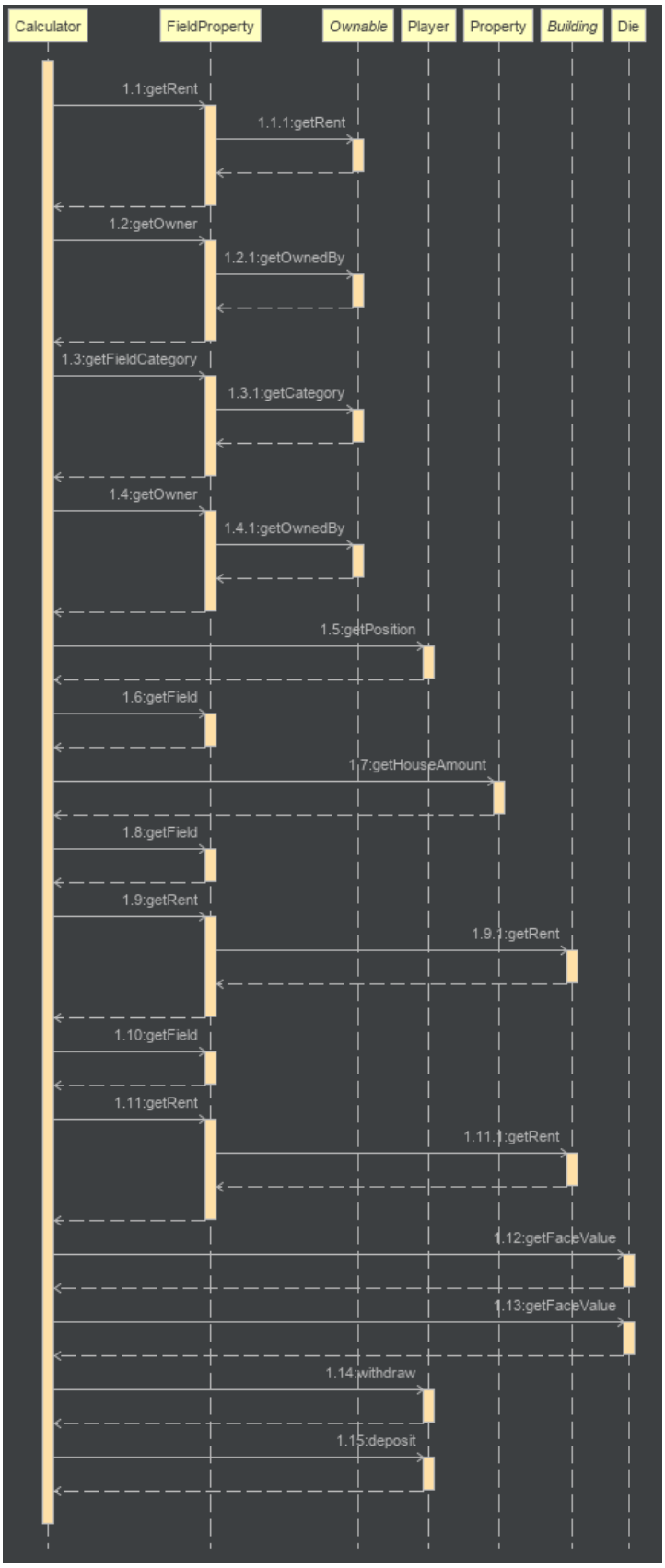
Et diagram over de metode kald der foretages når der købes et felt. Vi har uddelt ansvaret fra logikken i vores TurnController til en anden klasse, altså Calculator, der tager sig af udregninger mellem felter og spillerne. Til sidst opdateres disse til vores GUI.

auction



Her ses vores sekvensdiagram over auktioner, når spillere takker nej til at købe en grund der ikke er ejet. Den sender forespørgsler til spillerne om eventuelle bud, og derefter får spilleren med højeste bud grunden, og mister penge tilsvarende til dette. Derefter opdateres vores GUI.

payRent



Dette diagram viser interaktionen mellem spilleren, og vores forskellige typer felter når disse er ejet af andre spillere. Den checker for hvert felttype hvad det vil koste at lande på denne, og trækker så det korrekte beløb fra den givne spiller, og sætter dette beløb ind på feltets ejers konto.

6 Implementering

- Hvilke tanker har vi gjort os?
  - Vi har i projektet arbejdet efter Unified Process, og dermed arbejdet i iterationer. Vi startede vores projekt med at sætte os grundigt ind i reglerne for Matador, og dannede os et overblik over projektet. Med det samme er det tydeligt at der er nogle af regler der vil være overflødige, fordi de er baseret på de få resourcer der er kunne være i et fysisk monopoly spil som fx. penge, hus- , og hotelbrikker. Disse regler er som sagt overflødige, da spillet kører på en computer er vi ikke begrænsede af fysiske genstande som huse/penge.
- Opdeling af krav
  - Vi opstillede herefter en kravspecifikation til produktet ud fra spillets regler, via MoSCoW principperne. Herefter lavede vi en tidsplan ud fra disse. Dette gav et godt overblik over hvad vi skulle nå til hvornår. Herefter udarbejdede vi vores Use Cases.
- Arkitektoniske overvejelser
  - Efter vi havde gjort os nogle generelle overvejelser om arkitekturen for projektet, gik vi i gang med at udarbejde et "Minimum Viable Product". Dette muliggjorde at vi hurtigt fik en simpel prototype op at kører, hvilket gjorde at vi kunne komme i gang med at løse eventuelle problemer vi havde identificeret vha vores P/I matrix.
  - Vi fandt hurtigt ud af at der var mange klasser og metoder vi kunne genbruge fra tidligere projekter, dog med få småændringer, dette gjorde at vi forholdsvist hurtigt fik et fornuftigt produkt op at kører.
  - Vi valgte at inddele vores klasser inspireret af Model View Controller(MVC), da det gjorde det en del nemmere for alle gruppens medlemmer at arbejde på forskellige dele af programmet, da det var tydeligt i hvilke pakken der skulle arbejdes, alt efter hvilke features man implementerede.
  - Vi valgte at bruge polymorfi til felterne, da der er en del forskellige felter, med forskellige udfald, og vi mente derfor polymorfi ville være den optimale måde at opbygge spilbrættet på
  - Da der er en del felter, valgte vi at lave en Board klasse som indeholdte de ting spilleren skulle kunne se på pladen, og en GameBoard Klasse som indeholder de udfald et bestemt felt har.
  - Vi ville have vores chancekort så tæt på virkeligheden som muligt, altså én bunke med kort hvor kortene bliver trukket ud én af gangen, og derefter lagt til siden. Vi har derfor lavet Deck klassen som en singleton klasse, da vi netop kun vil have én instans af kortbunken på ethvert givent tidspunkt i programmets køretid. Dette sikrer, at man ikke trækker det samme kort to gange. Det lykkedes os dog ikke at implementere at få kort fjernet fra bunken inden afleveringsfristen.

6.1 Interresante kodeeksempler

```
32      public void playGame() throws InterruptedException {
33          for (playerIndex = 0; guiController.getNumberOfPlayers() > 1; playerIndex++) {
34              guiController.getUserResponse(playerIndex);
35
36              //hvis man er i fængsel
37              if (movementController.getPlayers()[playerIndex].getInJail()) {
38                  jailBailOut(movementController.getPlayers()[playerIndex]);
39              }
```

Vi har valgt at beskrive playGame() metoden i TurnController klassen, da det er spillets logik. Det første playGame tjekker hvilken spiller der har tur. Spilleren der har turen bliver identificeret af playerIndex variablen. Det næste der sker er at tjekke om den spiller, der har turen, er i fængsel. Hvis de er i fængsel bliver metoden jailBailOut kaldt i linje 38, hvor spilleren kan forsøge at komme ud vha. fængselskort eller betale 1000 kr.

```
41              guiController.rollButtonGUI();
42              model.Player player = movementController.makeMove(playerIndex);
43              int die1 = movementController.getLatestRoll()[0].getFaceValue();
44              int die2 = movementController.getLatestRoll()[1].getFaceValue();
45              int diesum = die1 + die2;
46
47              guiController.displayRollGUI(die1,die2);
```

I linje 41 skriver GUI'en ud, at spilleren skal slå med terningerne. Når spilleren har trykket ok, bliver der i linje 42 brugt metoden makeMove. Denne bevæger spilleren alt efter om de er i fængsel, hvor den kræver at de ruller to ens, ellers vil de ikke blive rykket. Metoden returnerer også player objektet ud fra spillerens indeks. Fra linje 43 - 45 bliver terningernes individuelle værdier og sammenlagte værdi hentet. I linje 47 bliver terningernes værdier sendt til GUI'en og vist på skærmen.

```
49          if (!(player.getTurnsInJail() > 0)) {
50              guiController.movePlayerGUI(playerIndex, movementController.getLatestPosition(player, diesum), diesum);
51          }
52
53          boolean playerInJail = player.getInJail();
54          if (!playerInJail) {
55              guiController.displayGUIMsg(movementController.passedStart(movementController.getLatestPosition(player,diesum)
56                      , player.getPosition()));
57              guiController.updatePlayerBalanceGUI(playerIndex, player.getBalance());
58
59              int fieldNumber = player.getPosition();
60              field.Field plField = calculator.getField(fieldNumber);
```

I linje 49 tjekker spillet, om spilleren skal rykkes på spillebrættet. Det gøres ved at se om spilleren har slået et slag i fængslet og om de er blevet sat til 0 ture i fængslet efter slaget. I linje 54 tjekker spillet så om spilleren er i fængsel, og hvis de ikke er, bliver der tjekket om de har passeret start i linje 55. Linje 57 opdaterer så spillerens balance i GUI'en. Der bliver herefter lavet en effekt, alt efter hvilket felt spilleren er landet på.

```
61              //hvis købbart felt
62              if (plField instanceof Ownable) {
63
64                  //Hvis feltet ikke er ejet, købes det
65                  if ((((Ownable) plField).getOwnedBy() == null)) {
66                      if (calculator.getCredibilityBuy(player, fieldNumber) &&
67                              guiController.yesNoButton(Language.queryFieldBuy(((Ownable) plField).getCost()))) {
68                          buyField(playerIndex, player, fieldNumber, plField);
69
70                          //hvis spilleren ikke kan, eller vil købe feltet går det til auktion
71                      } else {
72                          auction(player, plField);
73                      }
74                  } else {
75                      //Hvis feltet ejes og du er ejer, kan du byg hus
76                      if (plField instanceof field.Ownable.Property && player.equals(((Ownable) plField).getOwnedBy()) &&
77                      calculator.getCredibilityHouse(player, fieldNumber,  amount: 1)) {
78
79                          if (calculator.isBuildable(fieldNumber)) {
80                              build(playerIndex, player, fieldNumber, (Property) plField);
81                          }
82                      }
```

Hvis feltet spilleren lander på er købbart, bliver de sendt ind i den nedenstående sektion af koden, fra linje 62 til linje 69. Calculator tjekker herførst om du har penge nok, og hvis du har, får du muligheden for at købe eller ej. Hvis det er et ej, sætter du grunden til auktion mellem de andre spillere.

```
83              //ellers betal husleje
84              if (((Ownable) plField).getOwnedBy() != null && ((Ownable) plField).getOwnedBy() != player) {
85                  if (calculator.getCredibilityRent(player, fieldNumber)) {
86                      guiController.getUserResponse(Language.payRentToPlayer(((Ownable) plField).getOwnedBy().getName(), ((Ownable) plField).getRent()));
87                      Player owner = calculator.payRent(player, fieldNumber, movementController.getLatestRoll());
88                      for (int i = 0; i < guiController.getNumberOfPlayers(); i++) {
89                          if (movementController.getPlayers()[i].equals(owner)) {
90                              guiController.updatePlayerBalanceGUI(i, owner.getBalance());
91                          }
92                      }
93                  } else {
94                      playerBankrupt(player, fieldNumber);
95                  }
96              }
97          }
98      }
```

Hvis du lander på et chancekort felt, får du tildelt et chancekort fra deck'et, hvor du gør som chancekortet siger. Derefter bliver det chancekort du trak smidt ud af deck

```
100             //chancekort
101             if (plField instanceof ChanceField) {
102                 landOnChancecard(player, plField);
103             }
104             //Gå i fængsel-felt
105             if (plField instanceof field.NotOwnable.GoToJail) {
106                 movementController.landOnJailField(playerIndex);
107                 guiController.removeCarGUI(playerIndex,  newLocation: 30);
108                 guiController.teleportPlayerGUI(playerIndex,  newLocation: 10);
109             }
```

Husleje-delen af methoden nedenunder ser hvilken spillers tur det er, samt hvem der ejer grunden. Derefter beregner calculator klassen det fuldstændige beløb der skal betales, ved at tjekke hvilken grund det er, og om de andre felter af samme farve er ejet. Samt om der er huse/hoteller på feltet. Efter beløbet er beregnet tjekker den om spilleren har tilstrækkeligt med penge til at kunne betale, og hvis ikke, går spilleren fallit/Bankrupt.

```
111             //tax
112             if (plField instanceof field.NotOwnable.Tax) {
113                 if (fieldNumber == 4) {
114                     if (guiController.getUserDecision(Language.queryPayTax(),
115                             option1: Language.pay() + " 10%", option2: Language.pay() + " 4000 " + Language.currency())) {
116                         if (calculator.getCredibilityTax10(player)) {
117                             calculator.payTax10(player);
118                         } else {
119                             playerBankrupt(player, fieldNumber);
120                         }
121                     } else {
122                         if (calculator.getCredibilityTax(player, fieldNumber)) {
123                             calculator.payTax(player, fieldNumber);
124                         } else {
125                             playerBankrupt(player, fieldNumber);
126                         }
127                     }
128                 } else {
129                 if (calculator.getCredibilityTax(player, fieldNumber)) {
130                     calculator.payTax(player, fieldNumber);
131                 } else {
132                     playerBankrupt(player, fieldNumber);
133                 }
134                 }
135             }
136 
137         }
```

Denne del af metoden bruges til feltet med indkomstskat, hvor man får valget at betale enten 4000 kr eller 10% af de penge man har. Igen, hvis man ikke har nok penge til at betale, bliver fallit methoden igangsat.

```
138             if (!playerBankrupt) {
139                 //opdaterer spiller balance i GUI
140                 guiController.updatePlayerBalanceGUI(playerIndex, player.getBalance());
141             }
142 
143             //giver ekstratur hvis der er slået dobbeltslag
144             playerIndex = evalTurnTimer(playerIndex, player,die1,die2);
145         }
146     }
```

Fra linje 138 til 141 ses den før omtalte bankrupt-funktion, som igangsætter en metode, der hedder playerBankrupt. Bliver forklaret længere nede. Ellers gør den sidste del af playGame methoden sådan, at hvis man har slået to ens, får man en tur mere.

```
319     private void playerBankrupt(Player player, int fieldNumber){
320         guiController.getUserResponse(Language.lostGame());
321         Field field = calculator.getField(fieldNumber);
322         if (field instanceof Fields.Ownerable){
323             int[] fields = calculator.valuesTransfer(player, fieldNumber);
324             int newOwnerNumber = 0;
325             for (int i = 0; movementController.getPlayers().length > i; i++){
326                 if (movementController.getPlayers()[i].equals(((Ownerable) field).getOwnedBy())){
327                     newOwnerNumber = i;
328                 }
329             }
330             for (int i = 0; fields.length > i; i++){
331                 guiController.setFieldBorderGUI(fields[i], newOwnerNumber);
332             }
333             guiController.updatePlayerBalanceGUI(newOwnerNumber, movementController.getPlayers()[newOwnerNumber].getBalance());
334         } else {
335             int[] fields = calculator.valuesTransfer(player);
336             for (int i = 0; fields.length > i; i++){
337                 guiController.remmovePlayerOwned(fields);
338             }
339             guiController.remmovePlayerOwned(fields);
340         }
341         guiController.updatePlayerBalanceGUI(turnTimer,  newBalance: -1);
342         removePlayer();
343         turnTimer--;
344         playerBankrupt = true;
345     }
```

Ovenfor kan man se playerBankrupt-funktionen, der bruges til at fjerne spillere fra spillet og GUI boardet, samt overføre deres værdi og grunde til en anden spiller, i tilfælde af at de ikke har råd til at betale huslejen til dem. Hvis de går fallit på grund af et chancekort eller et skattefelt, skal der selvfølgelig ikke overføres nogle værdier. Dog skal spillerens felter igen gøres tilgængelig for køb, så de andre spillere kan fortsætte spillet.
Så if-sætningen på linje 322 er den, der skal bruges i tilfælde af at man taber fordi man ikke kan betale husleje. Det bliver gjort ved at tjekke hvilket felt spilleren står på, da købbare felter er de eneste felter, hvor man kan tabe til andre spilllere. Hvis man ikke har tabt til en anden spiller, ryger man ned i linje 334, hvor der kun sker det at alle ens grunde bliver gjort tilgængelig igen. Til sidst i methoden, i linje 341, bliver ens balance sat til -1 for at vise at man ikke længere er med i spillet. turnTimer bliver gjort mindre med 1, så den næste spiller får sin tur, da player arrayet nu er 1 kortere. Bankrupt bliver sat til true, så der ikke gives ekstra tur til spilleren hvis de har slået to ens.

7 Dokumentation

7.1 GRASP Principperne

Vi har forsøgt at overholde GRASP principperne, dvs. at vi har lavet klasser som er information experts og klasser der står for beregninger.
Vi har eksempelvis en Player-klasse, som indeholder alle informationerne om spillere, og så har vi en MovementController-klasse, som står for at beregne f.eks. spillerens nye position på brættet. Denne agerer som en player-controller.

- Information Expert
  - Et eksempel på brug af et information expert pattern i vores kildekode, er vores Player klasse. Denne klasse holder nemlig kun på de informationer der er relevant for den, som fx spillerens pengebeholdning, ejerskab, og fængselsstatus. Dette sikrer en lavere kobling, da andre klasser ikke er påkrævet at holde styr på disse, samt høj samhørighed, da alt information om spilleren ligger i spillerklassen. Dette gør programmet lettere at vedligeholde for fremtidige udviklere.
- Controllers
  - TurnController:
    - Denne klasse står for den generelle logik omkring spillerens tur. Hvad der skal ske i hvilke tilfælde, hvilke effekter der skal sættes igang alt efter hvor de lander, om de er i fængsel, om de skal have en ekstra tur m.m.
  - MovementController:
    - Denne klasse står for at rykke spilleren på brættet. Dette indebærer hvis de lander på "gå i fængsel"feltet og tager også højde for om spilleren skal rykke i fængsel, når de har slået to ens tre gange. Der tages også højde for om de slår sig ud af fængsel og hvad spilleren sidst har slået.
  - GUIController:
    - Denne klasse indeholder alle metoder som benytter GUI'en, herunder når bilernes position bliver vist og når spilleren skal give input. Denne klasse står kun for at benytte den udleverede GUI's metoder.
- Creator
  - GameBoard
    - Denne klasse står for at oprette felterne som bruges i model. Klassen indeholder et array af felterne og har hovedsagligt til ansvar at hente et felt der er landet på til controllerne. Klassen har også et ansvar for at ændre i felters oplysninger.
  - MovementController
    - Klassen står for at oprette player-objekterne og Diceholder-objektet. Klassen står også for at hente player objektet op til de andre controllere. Det kan nævnes at man i et fremtidigt projekt med fordel kunne bruge en anden klasse som creator for player-objekterne.
  - GUIGameBoard
    - Har til ansvar at oprette alle felterne som GUI'en henter tekst og formatering fra.

7.2 Coupling

Der er i løbet af projektet blevet gjort mange overvejelser om at opnå lavest mulig kobling mellem klasserne. Disse kan blandt andet ses i vores TurnController, hvor vi har uddelt dens ansvar til andre klasser, som f.eks. MovementControlleren, der tager sig af nogle af de ting som vi ellers kunne have haft i blot én klasse. Dette sikrer lavere kobling mellem vores TurnController og andre klasser end der ellers ville have været.

7.3 Cohesion

Vi har generelt i vores program opnået høj samhørighed. Klasser er opdelt således at de har fokuserede ansvarsområder, eksempelvis har vi klasser som Player og Die, der begge har sigende klassenavne og holder på metoder der er relevante for dem. Dette gør det let at overskue koden for nye udviklere og hurtigt finde de relevante metoder man ønsker at tilgå.

8 Test cases

I dette kapitel vil vi teste vigtige dele af vores program, det vil gøres enten via en manuelt spiltest eller jUnit test. De klasser vi har anset som vigtige er vores controller klasser, da de indholder spil logik. Vi har ikke anset det som vigtigt at teste infomationsklasser da de kun indholder getter og setter metoder. Testene har det formål at teste alle de mulige inputs til metoder, og tjekke om de udføre/retunere den ønskede handling.

8.1 MovementController

Test Case ID	TC01
Summary	Test at spilleren bliver placeret korrekt på Gameboarded
Requirements Tested	R1, R4
Precondition	getFacevalue() skal havde returneret et terningekast
Postcondition	Spillet fortsætter
Test Procedure	Sammenligning af spillerplacering og forventet placering efter terningekast
Test Data	Metode test, data ikke nødvendigt
Expected Result	AssertEqual = true
Actual Result	True
Status	Passed
Tested By	Mikkel Thorsager
Date	14/1-2019
Test Environment	IntelliJ IDEA

8.2 addToBalance

Test Case ID	TC02
Summary	Test af spillerens balance, og om den opdaterer korrekt vha. deposit metoden
Requirements Tested	Betaling af div. køb
Precondition	Spiller modtager/mister penge
Postcondition	Spilleren har korrekt balance
Test Procedure	Sammenligning af forventet og aktuel balance
Test Data	Metode test, data ikke nødvendigt
Expected Result	True
Actual Result	True
Status	Passed
Tested By	Jonas Henriksen
Date	10/1-2019
Test Environment	IntelliJ IDEA

8.3 setOwner

Test Case ID	TC03
Summary	Test af setOwner
Requirements Tested	R10, R11, R12, R13, R14, R15, R16
Precondition	En spiller har købt en grund
Postcondition	Feltet har korrekt owner
Test Procedure	Sammenligning af aktuel og aktuel owner
Test Data	Metode test, data ikke nødvendigt
Expected Result	True
Actual Result	True
Status	Passed
Tested By	Jonas Henriksen
Date	10/1-2019
Test Environment	IntelliJ IDEA

8.4 test af Jail

Test Case ID	TC04
Summary	test af Gå i Fængsel
Requirements Tested	En spiller kan ryge i fængsel
Precondition	Spilleren lander på I Fængsel
Postcondition	Spilleren sættes i fængsel
Test Procedure	Sammenligning af forventet inJail boolean og aktuel inJail boolean
Test Data	Metode test, data ikke nødvendigt
Expected Result	True
Actual Result	True
Status	Passed
Tested By	Jonas Henriksen
Date	17/1-2019
Test Environment	IntelliJ IDEA

8.5 getCredibilityBuy

Test Case ID	TC05
Summary	Tester om man kan købe felter på det rigtige tidspunkt
Requirements Tested	En spiller kan købe eller ikke købe felt
Precondition	Spilleren lander på et Property felt
Postcondition	Metoden retunere true eller false
Test Procedure	Chekker om boolean er true eller false, når spillern har 30000, feltets pris og feltets pris - 1
Test Data	Metode test, data ikke nødvendigt
Expected Result	True, True, False
Actual Result	True, True, False
Status	Passed
Tested By	Alexander
Date	19/1-2019
Test Environment	IntelliJ IDEA jUnit

8.6 getCredibilityHouse

Test Case ID	TC06
Summary	Tester om man kan købe huse på det rigtige tidspunkt
Requirements Tested	En spiller kan købe eller ikke købe huse
Precondition	Spilleren lander på et Property felt som er ejet af spilleren
Postcondition	Metoden retunere true eller false
Test Procedure	Chekker om boolean er true eller false, når spillern har 30000, Hus prisen og hus prisen - 1 gørs for 1 og 2 huse
Test Data	Metode test, data ikke nødvendigt
Expected Result	True, True, False
Actual Result	True, True, False
Status	Passed
Tested By	Alexander
Date	19/1-2019
Test Environment	IntelliJ IDEA jUnit

8.7 buyWithPrice

Test Case ID	TC07
Summary	Tester om man kan købe felter med en bestemt pris (bruges i auktionen
Requirements Tested	En spiller kan købe Huse
Precondition	Spilleren lander på et Property felt
Postcondition	Metoden retunere true eller false
Test Procedure	Chekker om feltes ownedBy er lig spilleren, når spillern køber grund for 10000
Test Data	Metode test, data ikke nødvendigt
Expected Result	True, True
Actual Result	True, True
Status	Passed
Tested By	Alexander
Date	19/1-2019
Test Environment	IntelliJ IDEA jUnit

8.8 payRent

Test Case ID	TC08
Summary	Tester om man betaler den rigtige pris for at lande på en andes felt
Requirements Tested	Leje bliver trukket rigtigt
Precondition	Spilleren lander på et Property felt som er ejet af en anden spiller
Postcondition	Spilleren betaler leje
Test Procedure	Chekker om spilleren bliver trukket lejen og lejen gange 2 hvis modstanderen ejer alle felter af samme type
Test Data	Metode test, data ikke nødvendigt
Expected Result	True, True
Actual Result	True, True
Status	Passed
Tested By	Alexander
Date	19/1-2019
Test Environment	IntelliJ IDEA jUnit

8.9 Auction

Test Case ID	TC09
Summary	Hvis en spiller vælge ikke at købe et felt, skal det på auktion
Requirements Tested	Auktion
Precondition	Spiller vælger ikke at købe feltet
Postcondition	Feltet bliver købt til højste bud, eller ikke købt hvis der ikke er noget bud
Test Procedure	Køer forskellige bud rundter igennem
Test Data	test1 spiller 2 byder 6000, spiller2 vælger ikke at byde test2: Spiller1 byder 6000 og 8000, spiller2 byder 7000 og ikke noget bud test3: spiller1 og spiller2 byder ikke på grunden
Expected Result	Solgt til spiller1 for 6000, solgt til spiller1 for 8000, ikke solgt
Actual Result	Solgt til spiller1 for 6000, solgt til spiller1 for 8000, ikke solgt
Status	Passed
Tested By	Alexander
Date	19/1-2019
Test Environment	IntelliJ IDEA jUnit

9 Konfigurationsstyring

9.1 Krav til styresystem

OBS[3]

Software

- 64-bit Windows (10, 8, 7)
- MacOS 10.11 eller nyere
- Java Runtime Environment (JRE)
- Java 8 Development kit (JDK, denne pakke inkluderer JRE)
- IntelliJ IDE

Hardware

- 2 GB ram minimum, 8 GB anbefalet.
- 2.5 GB Harddisk plads, SSD harddisk anbefalet.
- 1024x768 minimum skærm opløsning.

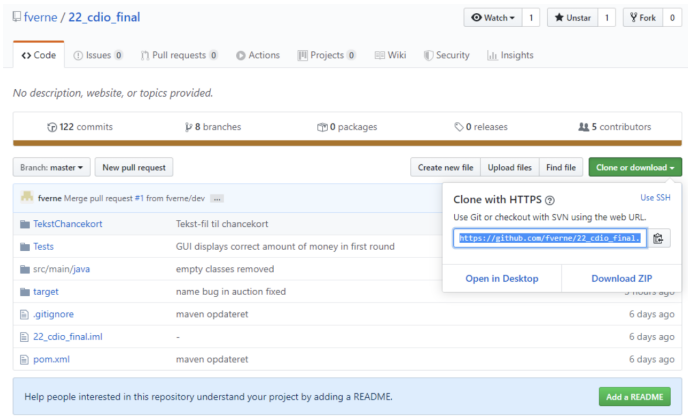
Installation af nødvendig software

1. Java 8 Windows - Mac
2. IntelliJ. (Installations vejledning: Windows - Mac) eller Download IntelliJ direkte

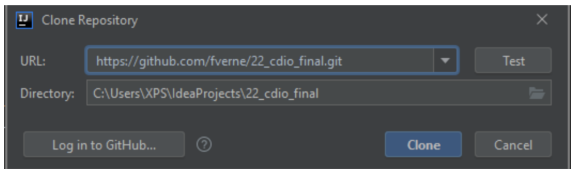
[3]Dele er genbrugt fra gruppe 20a CDIO2 opgave.

9.2 Import af projekt fra GitHub

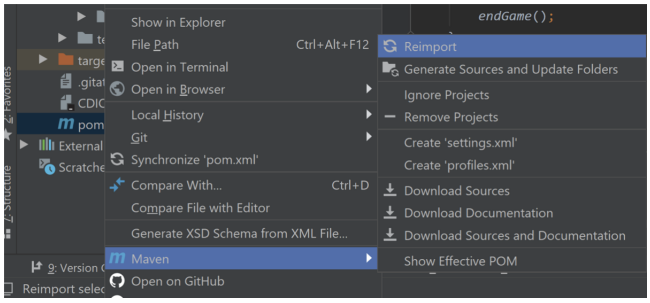
1. gå til linket `https://github.com/fverne/22_cdio_final.git`
2. Importer projektet ved at trykke på clone or download, kopiere linket i POP-UP menu



3. Nu åbner vi Intellij, og klik på Check out from version control, og derefter tryk på Git.



4. Når intelliJ åbner skal man finde pom.xml filen ude i menuen i venstre side, derefter skal man højreklikke og trykke reimport



5. klik derefter på "Kør"ikonet for at kører programmet

10 Refleksion

10.1 Projekt forløb

Vi startede projektet med at analysere et fysisk matadorspil grundigt. Ud fra denne analyse, udarbejdede vi en prioriteret kravliste, samt inddelte kravene alt efter kategori. Derefter gik vi igang med use cases, vi udarbejdede to use cases, "start spil" og "spil spil". Derefter lavede vi to fully dressed use cases. Vi brugte vores fully dressed use cases til at anskueliggøre flowet i matadorspillet ved brug af alternative flow og extensions. Vi udarbejdede en navneordanalyse udfra vores use cases og kravliste.

Navneordsanalyse blev senere så brugt til at navnegive vores klasser skabe vores første design klassediagram. Vi kunne nu danne os et overblik over hvilke klasser vi ville komme til at bruge. Design klasse diagrammet blev løbende opdateret så ændringer i koden stemte overens med diagrammet, da det hjælpe os med at have et konstant overblik over vores kobling, kohæsion, metoder og i hvilke klasser metoderne var placeret.

Vi udarbejdede en risikoanalyse for at anskueliggøre hvilke kernefunktionaliteter vi skulle få styr på først. Denne blev relevant i de senere iterationer, for da vi havde fået styr på de største risici kunne vi arbejde på Quality of Life improvements. Vi udarbejdede et system sekvens diagram for at danne os et overblik over hvordan en helt basal version af spillet skulle fungere.

Efter den første analysefase, lavede vi en arbejdsplan udfra vores kravspecifikation, og udarbejdede en plan for hver af vores 4 iterationer, som gik fra den 6/1 til den 20/1. Da vi var hurtigere end forventet i startet, valgte vi også at "opdatere" vores arbejdsplan løbende, ud fra hvilke "vigtige" funktioner vi manglede, og hvor langt vi var med de påbegyndte opgaver. Vores plan var at fryse koden et par dage inden afleveringsfristen, og kun arbejde på minor bugfixes, samt tests og rapportering.
Efter vi havde lavet vores første arbejdsplan, brainstormede vi vores ideer til designet, vi var enige om at vores designfase ikke skulle vare for længe og heller ikke være særligt detaljeret, da vi sikkert ville få mere ud af, at fordele de forskellige krav og opgaver, og komme i gang med at programmere, da vi sikkert ville lave mange ændringer undervejs og fra den ene iteration til den næste.

Da vi kom til den 17/1 uddelte vi igen vores opgaver, da vi mangler at få vores fallit og vinder metoder til at fungerer helt korrekt, satte vi 2 til at sidde og lave dem sammen, 2 andre gik i gang med at teste og skrive eventuelle ændringer ned, som skulle laves den 18/1 og 19/1, og de 2 sidste gruppemedlemmer ville arbejde på rapporten, som de så gjorde alle de sidste dage frem til den 19/1, da det blev konkluderet at være for risikabelt at havde alt for meget arbejde den 20/1.

10.2 Konklusion

Det er lykkes os at opfylde næsten alle de krav vi stillede i opgaven, den eneste vi ikke nåede at få med var pantsætning af ejendomme.

Vi har i denne opgave, taget godt brug af det vi har lært i de forrige CDIO opgaver. Vi har lært hvor vigtigt det var at kunne arbejde struktureret efter en tidsplan ved fx brug af Unified process og dens arbejde i iterationer. Dette gjorde at folk ville snakke sammen ved både begyndelsen og slutningen af hver iteration, for at fordele opgaver, sparre med hinanden, samt at kigge hinandens kode igennem.

Ved brug af vores prioriterede kravliste har vi også hurtigt kunne identificere hvilke krav vi ville bearbejde i hver iteration. Vi tilbragte derfor kort tid med umiddelbar analyse og design af vores program, da vi igennem Unified Process ville vende tilbage til det alligevel. Den hurtige start med programmering sikrede også at vi hurtigt havde en prototype, som gjorde vi kunne uddellegere arbejdet ind imellem gruppemedlemmer uden at vi lavede på det samme.

Vi besluttede os for tidligt i projektet at opdele vores projekt efter Model-View-Controller principperne, da nogle af os havde god erfaring med den fra CDIO 3, og at den gør det nemmere at overholde lav kobling og høj samhørighed, fordi det er en del af principper bag MVC. Ved brug af vores UML og at vi har opdateret den til at være en kopi af vores program, har vi kunnet holde overblik over hvilke klasser der er, deres metoder, om der skulle indføres nye metoder, constructors, og variabler.

Vi har overordnet været rigtig tilfreds med vores gruppearbejde, da det føles som om alle har været med undervejs og at vores færdige produkt lever op til den standard vi stillede for den. Vi synes også vores rapport er informerede nok. Da vi har brugt vores analyse og design fase arbejde til at understøtte vores implementeringsfase markant mere end nogle af os har gjort i forrige opgaver.

11 Bilag

11.1 Bilag A - Link til GitHub

`https://github.com/fverne/22_cdio_final`