CDIO delopgave 3 efterår 2016

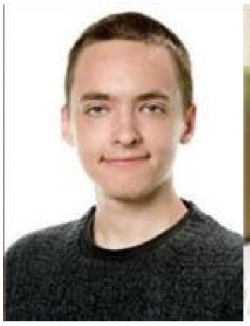
02312 - Indledende programmering, 02313 - Udviklingsmetoder til IT-systemer, 02315 - Versionsstyring og testmetoder.

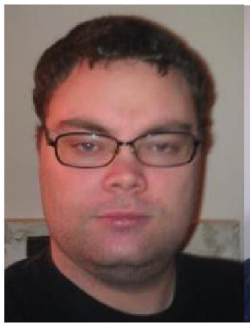
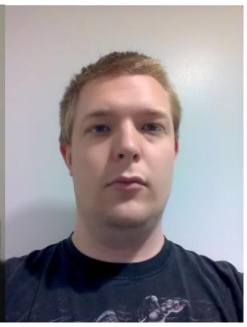
Projektnavn: CDIOdel3
Gruppenr: 42
Afleveringsfrist: Fredag d. 25/11 2016 Kl. 23:59

Denne rapport afleveres via Campusnet (der skrives ikke under).

Denne rapport indeholder 57 sider incl. denne side.

s165225; Al-Alak, Mehdi A.
s165221; Jensen Lasse
s165216; Jokil, Michael
s16523; Jørgensen, Jimmy
s165206; Soelberg, Andreas.

Al-Alak, Mehdi A. Jensen Lasse Soelberg, Andreas. Jørgensen, Jimmy Jokil, Michael

Timeregnskab

Navn	Analyse	Design	Imp	Test	Doku	Total
Jimmy	**6**	**0**	**17.5**	**5**	**7**	**35.5**
Mehdi	**5.5**	**2**	**4.5**	**0**	**6.5**	**18.5**
Andreas	**6.5**	**2.5**	**2**	**7.5**	**2.5**	**21**
Lasse	**5**	**3**	**14**	**1.5**	**6**	**29.5**
Michael	**5.5**	**6**	**3**	**2**	**8**	**24.5**
Total tid brugt.						**124.5**

Indholdsfortegnelse

Resume

Denne rapport omhandler en gennemgang af udarbejdelse for IOOuterActive's spil-vision. Indholdet af denne vision som vi modtog fra IOOuterActive skitserede løst rammerne for deres vision, men indholdet var for det meste op til os selv. Vi startede med at opsætte nogle krav, ud fra den vision vi havde fået. Ud fra kravene designede vi 1.version af et klassediagram og SSD-diagram. Herefter begyndte implementationen af programmet, hvor forskellige dele blev tilføjet pr iteration. Selve programmet er blev testet via JUnit og kørsel af programmet og de sidste fejl rettet.

Indledning

Det følgende spil, som bliver præsenteret i denne rapport, er lavet til IOOuterActive ud fra deres vision. Dette er et spil der kan spilles af 2-6 spillere med 21 felter som hver har en unik funktion. Ud fra de krav vi har udviklet ud fra IOOuterActive vision, har vi lavet analyse og design modeller, implementeret koden og testet spillet.

Hovedafsnit

Kravspecifikation

ID	Functional requirements	ID	Non-funuctional requirements
	Functionality		*Reliability*
F1	Der skal være 2 til 6 spillere	**NF1**	Spillet skal køre uden forsinkelser på over 0.5 sekunder.
F2	Spillerne skal slå med to terninger, som hver har 6 sider.	**NF2**	Spillet skal kunne spilles på DTU's databarer.
F3	Spillerne skal have en balance, som er synlig hele tiden.		*Supportability*
F4	Spillerne skal starte med en balance på 30000.	**NF3**	Spillet Strings skal kunne findes et samlet sted
F5	Start feltet skal ignoreres efter en spillers første tur		*Implementation constraints*
F6	En spiller går bankerot, når han har en balance på 0 eller mindre.	**NF4**	Spillet skal skrives i Java
F7	Spillet skal slutte når alle, på nær en spiller, er gået bankerot		
F8	Systemet skal springe en bankerot spillers tur over.		
F9	Spillerne skal gå rundt på brættet		
F10	Spillet skal indeholde 22 felter inklusiv et startfelt. (se bilag for liste)		
F11	Felterne skal have en unik effekt(se bilag)		
F12	Systemet skal komme med en besked når spilleren lander på et specifikt felt.		
F13	Felterne skal have en unik beskrivelse.		
F14	Spillerne skal have unikke navne.		
F15	Spillernavne skal højst indeholde 16 tegn		
F16	Når en spiller lander på et felt, skal spilleren i næste tur fortsætte fra samme felt (gælder ikke hvis spilleren er bankerot)		
F17	Hvis en spiller går bankerot og ikke har nok penge til at betale feltets ejer, betaler banken det resterende beløb		
F18	En spillers felter går til banken, hvis spilleren går bankerot		
F19	Spillerne skal kunne købe og eje felter.		
F20	Hvis et felt ikke har nogen ejer, koster det ikke noget at lande på feltet.		
F21	Systemet skal simulere kast af terninger.		
F22	Systemet skal fjerne en bankerot spillers bil fra brættet.		
F23	Spillerne skal selv vælge farve på deres bil.		

F24	Spillerne skal se hvem der ejer felterne.		
F25	Spillet skal være på engelsk.		
F26	Felterne skal have en titel.		
F27	Systemet skal vise felternes titel hele tiden		
F28	Spilleren skal have muligheden for at købe et felt, eller gå videre, hvis spilleren ikke ønsker at købe det.		
	Usability		
F29	Spillet skal have en start menu hvor man kan starte spillet, afslutte spillet eller se spillets regler.		
F30	Når en spiller starter sin tur skal han have en mulighed for at gå tilbage til menuen.		
F31	Systemet skal simulere rykning af spiller figur.		
F32	Spillet skal vises via en GUI.		
F33	GUI'en skal være intuitiv at bruge.		
F34	Al input skal foregå via GUI'en.		
F35	Når spillet er slut, skal systemet spørge om spilleren vil genstarte spillet		
F36	Systemet skal genstarte hvis spilleren vælger at genstarte spillet		
F37	Systemet skal nulstille dens information om det foregående spil.		

Konfigurationsstyring

For at være i stand til at køre programmet, skal man have Java version 1.7 eller nyere og Windows 7, 8 eller 10 installeret. Derudover skal man have eclipse og biblioteket GUI.jar.

Vejledning til import af Git-repository i Eclipse

1. Open din browser
2. Gå til denne webside https://github.com/LasseJensen213/42_CDIO_03
3. Kopier klonings URl til højre (https://github.com/LasseJensen213/42_CDIO_03.git)
4. Åben "Eclipse"
5. File -> Import
6. Tryk next
7. Git -> Projects from Git
8. Tryk next
9. Close URl
10. Tryk next
11. Sæt den kopierede URl i feltet
12. Tryk next
13. Tryk next
14. Vælg din sti til at være dit workspace
15. Tryk next

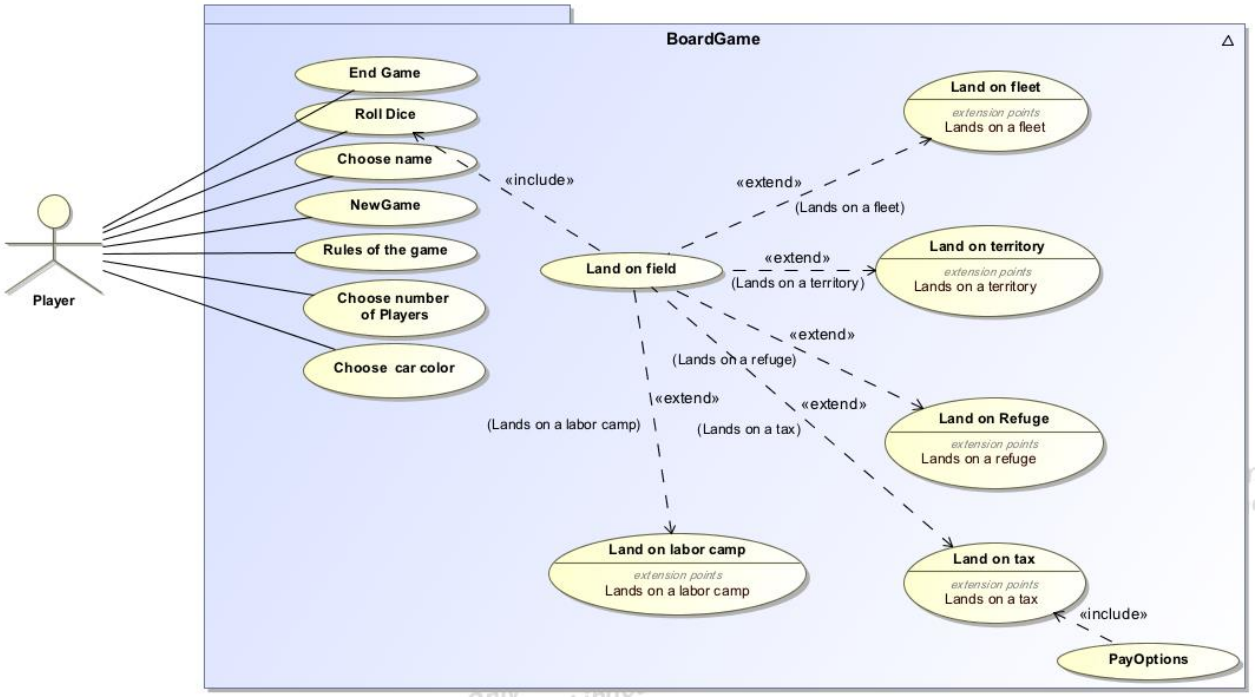
16. Tryk finish
17. Åben projektet i Eclipse
18. Åben pakken kaldet ”Game”
19. Åben klassen kaldet ”Menu”
20. Tryk ”Run” (Den grønne pil i toppen)
21. Spillet åbner nu et nyt vindue.

Kør program via .jar fil.

1. Åben din foretrukne browser
2. Gå til denne webside https://github.com/LasseJensen213/42_CDIO_03/tree/master/42_CDIO_03
3. Hent filen ”runnablejar.zip” eller klik her (Man bør være opmærksom at Windows, andre styresystemer og programmer ser .jar filer som potentielle trusler)
4. Åben filen ”runnablehar.zip”.
5. Eksporter mappens filer til et sted af ejet valg
6. Dobbelt klik på RunnableJar.jar for at åbne programmet.

Analyse

Use-case diagram



Ovenstående use-case diagram beskriver de use-cases som spilleren kan vælge imellem. Disse er tiltænkt for at programmet opfylder kravspecifikationerne.

Use-case beskrivelser

Use Case:	NewGame
Id:	1
Kort beskrivelse:	Spilleren starter spillet
Primær aktør:	Spillerne
Sekundær aktør:	-
Forudsætning:	ingen
Main flow:	1. Use case starter når der vælges ”Start game” vælges i menu.
Efterfølgende:	Spillet starter
Alternativt Flow:	

Use Case:	ChooseNumberOfPlayers
Id:	2
Kort beskrivelse:	Vælger antal spillere
Primær aktør:	Spillerne
Sekundær aktør:	-
Forudsætning:	NewGame skal ha' kørt
Main flow:	1. Use case starter når NewGame har kørt. 2. Systemet spørger om, hvor mange spillere som skal oprettes. 3. Spiller vælger antallet af spillere.
Efterfølgende:	Systemet opretter det valgte antal spillere
Alternativt Flow:	-

Use Case:	ChooseName
Id:	3
Kort beskrivelse:	Spilleren vælger et navn
Primær aktør:	Spilleren
Sekundær aktør:	Systemet
Forudsætning:	Spillet skal være startet og der skal være valgt antal spillere
Main flow:	1. Use case starter når ChooseNumberOfPlayers har kørt 2. Systemet beder spilleren om at indtaste et spillernavn 3. Spiller indtaster spiller navn.
Efterfølgende:	Systemet opkalder spilleren efter det valgte spillernavn
Alternativt Flow:	1. Use case starter når ChooseNumberOfPlayers har kørt 2. System autogenererer et spillernavn, fordi spilleren ikke selv valgte et navn.

Use Case:	ChooseCarColor
Id:	4
Kort beskrivelse:	Spilleren vælger farve på sin bil
Primær aktør:	Spilleren
Sekundær aktør:	
Forudsætning:	Spilleren skal være oprettet og have valgt spillernavn
Main flow:	1. Use case starter når ChooseName har kørt 2. Systemet spørger hvilken farve bil spiller vil have. 3. Spiller vælger figurfarve
Efterfølgende:	Spillerens bilbrik får den valgt farve
Alternativt Flow:	-

Use Case:	RulesOfTheGame
Id:	5
Kort beskrivelse:	Viser spilleren informationer omkring spillet
Primær aktør:	Spillerne
Sekundær aktør:	-
Forudsætning:	Systemet er startet.
Main flow:	1. Use case starter når der vælges ”Rules of the game” i menuen. 2. Systemet viser reglerne. 3. Spiller klikker ok. 4. Spiller retunerer til menu.
Efterfølgende:	Spiller er returneret til startmenuen

Use Case:	EndGame
Id:	6
Kort beskrivelse:	Afslutter systemet
Primær aktør:	Spillerne
Sekundær aktør:	-
Forudsætning:	Systemet er startet.
Main flow:	1. Use case starter når der vælges ”Close” i menuen. 2. Systemet spørger om spilleren er sikker på spilleren ville afslutte spillet. 3. Spiller vælger ja. 4. Systemet lukker ned.
Efterfølgende:	Systemet afslutter
Alternativ:	1. Use case starter når der vælges ”Close” i menuen. 2. Systemet spørger om spilleren er sikker på spilleren ville afslutte spillet. 3. Spiller vælger nej 4. Spiller returnerer til menu.

Use Case:	RollDice
Id:	7
Kort beskrivelse:	Spillerene slår med terningen og rykker det antal øjne terningerne tilsammen viser frem på pladen. Spilleren lander på et felt, som alle har en effekt.
Primær aktør:	Spilleren
Sekundær aktør:	
Forudsætning:	Spillerne skal være oprettede
Main flow:	1. Use Case startet når ChooseCarColor har kørt 2. Systemet beder spilleren om at slå med terningerne. 3. systemet viser summen af terningerne 4. systemet rykker spillerens brik frem på spilpladen, lig summen af terningerne. *Spilleren lander på Tribe Encampment* 5. Systemet spørger om spilleren vil købe feltet 6. Systemet sætter feltet til at være ejet af den pågældende spiller
Efterfølgende:	Næste spillers tur

For at se alternative flows, se bilag

Navneordsanalyse

Ud fra tidligt opstillede krav fik vi følgende navneordsanalyse

- Spiller
- Terning
- Balance
- Spilbræt
- Felt
- Tur
- effekt
- beskrivelse
- navn
- Menu
- placering

Da dette var tidligt i analyseprocessen, er navneordsanalysen ikke blevet opdateret.

Udsagnsordsanalyse

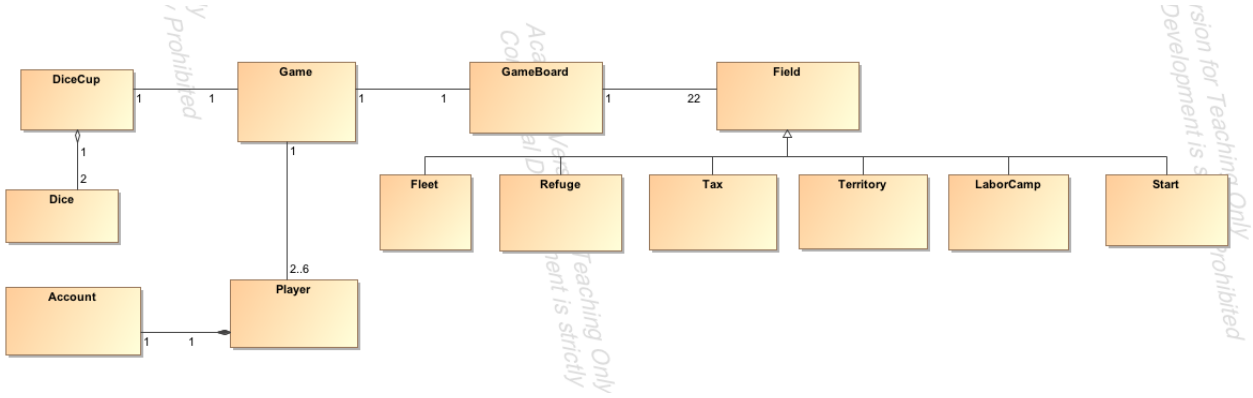
Ud fra tidligt opstillede krav fik vi følgende udsagnsordsanalyse

- Slå med terning
- Spring over spillers tur hvis bankerot
- Køb
- Gå rundt på bræt
- Eje
- Gå bankerot

Da dette var tidligt i analyseprocessen, er udsagnsordsanalysen ikke blevet opdateret.

Domæne model

Efter at have udarbejdet kravene til spillet skulle vi lave en analyse af spillet. For at finde ud af hvordan spillet skulle se ud og hvilke vigtige objekter der skulle med, valgte vi at en domæne model. Vi fik udarbejdet denne model:



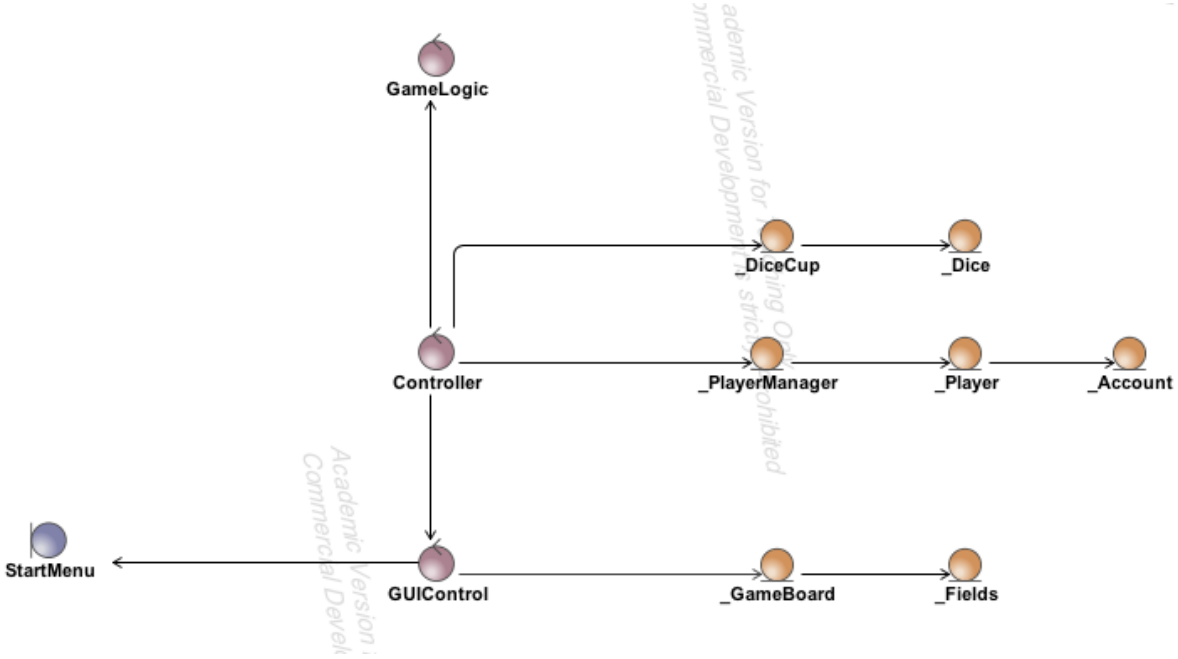
Vi fandt ud af, efter at have lavet vore domæne model, at ”GameBoard” var det helt centrale ved spillet, da alt ved spillet var forbundet til ”GameBoard”.

En anden vigtig ting vi fandt ud af var at vi skulle have noget alle forskellige typer felter kunne arve fra, ud fra det kom vi op med ”Field”. Her arver alle de forskellige felttyper fra ”Field”. En af disse kunne være ”Fleet”, dette er et felt der kan købes og hvis en anden spiller lander på din ”Fleet” skal han betale en sum alt afhængig af hvor mange Fleet’s du ejer. Udover de forskellige typer af felter med en funktion har vi også tilføjet et Start felt.

Den anden halvdel af modellen er selve spiller som vi har kaldt for ”Game”, der er forbundet til ”Player” som er selve spilleren, og til ”DiceCup” som indeholder terningerne ”Dice”. Da en spillet er afhænger af en penge balance, er ”Player” forbundet til en konto vi har kaldt ”Account”.

BCE diagram

For at finde ud af hvordan de forskellige klasser arbejder sammen skulle vi lave en BCE model, og vi kom frem til dette:

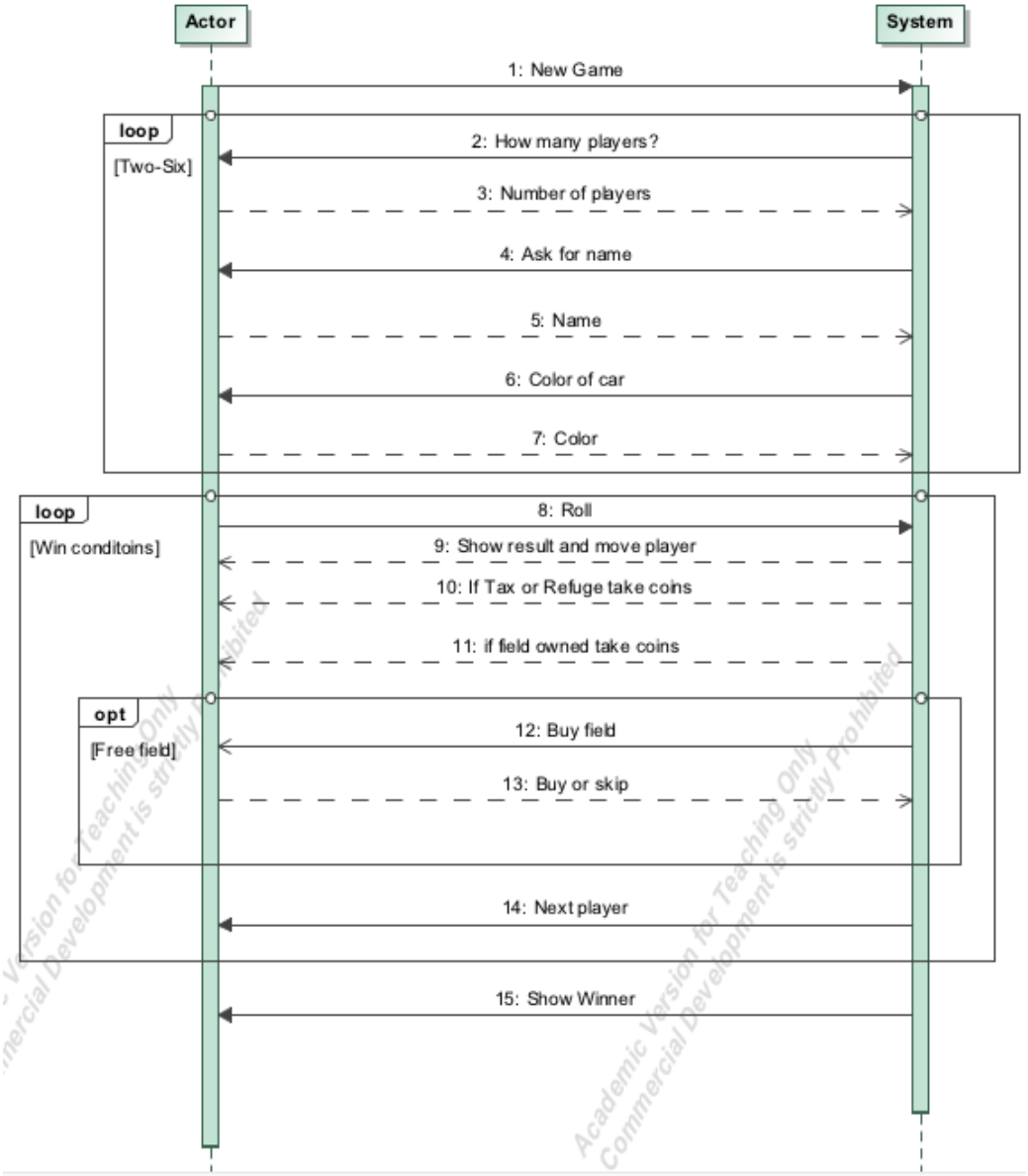


Vi startede med at tænke over hvilket klasser der skulle være boundaries, controllers og entities. Vi kom frem til at vores boundary var startmenuen da den er interaktionen mellem brugeren og systemet. Vores controller er bindeledet mellem hele systemet derfor er den mest centrale controller ”GUIcontrol”, som binder alle entities og boundareis sammen, ”Controller” og ”GameLogic” spiller samme rolle men igennem ”GUIcontrol”. Entities er det spillet omhandler og derfor kan vi ud fra domæne modellen, bestemme vores entities.

Systemsekvensdiagram

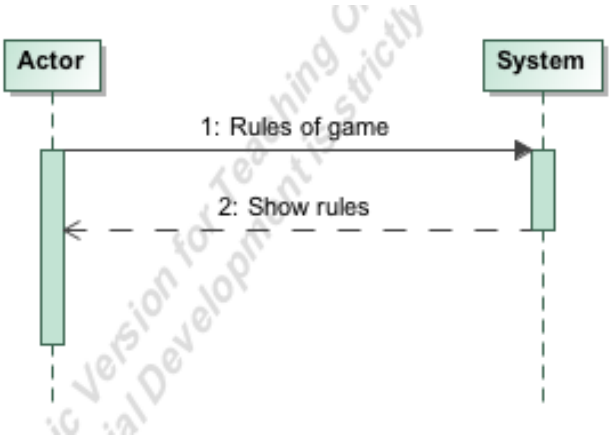
Efter at vi har lavet use cases og use case descriptions, kan vi nu lave vores system sekvens diagrammer (SSD), disse diagrammer viser de forskelige interaktioner der er mulige mellem bruger og system. Vi har lavet 3 SSD modeller da spillet starter i en menu hvor man har 3 muligheder.[1]

Den første model viser hvad den første funktion "New Game" gør. Ved at bruge nævnte funktionen vil systemet spørge for antal spillere, deres navne og hvilken farve biler de ønsker, dette kan gentage sig fra to til seks gange afhængigt af hvor mange spillere der er med. Når det er en spillers tur ruller terningerne, derefter vil man kunne se hvad man har slået og spillerens bil vil blive rykket , hvis feltet kræver betaling mister spilleren coins ellers kan spilleren købe feltet, og så er skifter turen til næste spiller. Dette vil fortsætte til en vinder:

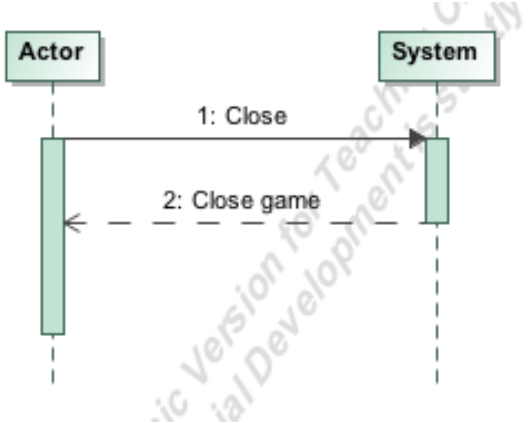


[1] CDIO 2

Næste model er for funktionen ”Rules of game”, hvis man bruger denne funktion vil systemet vise reglerne for spiller:

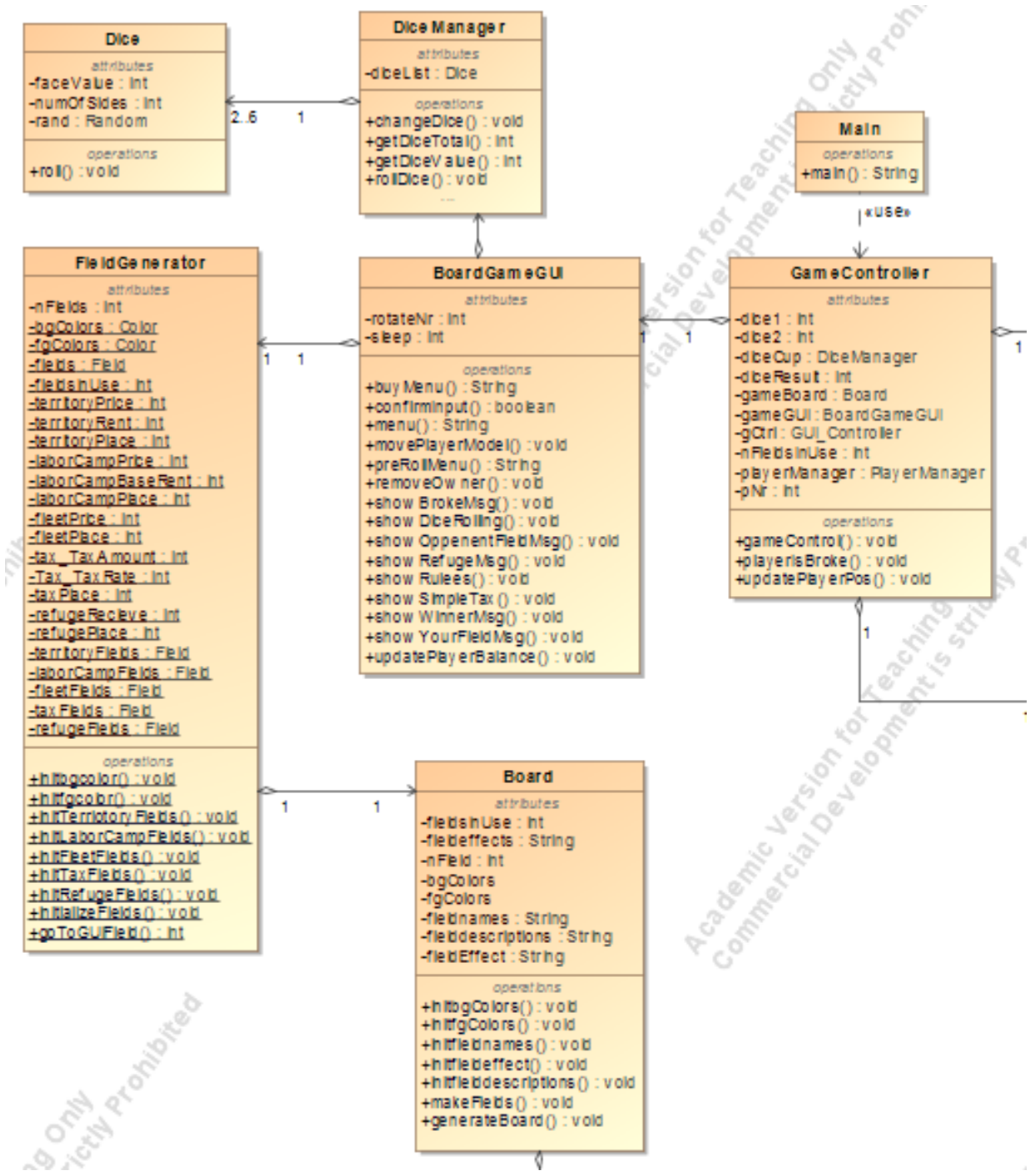


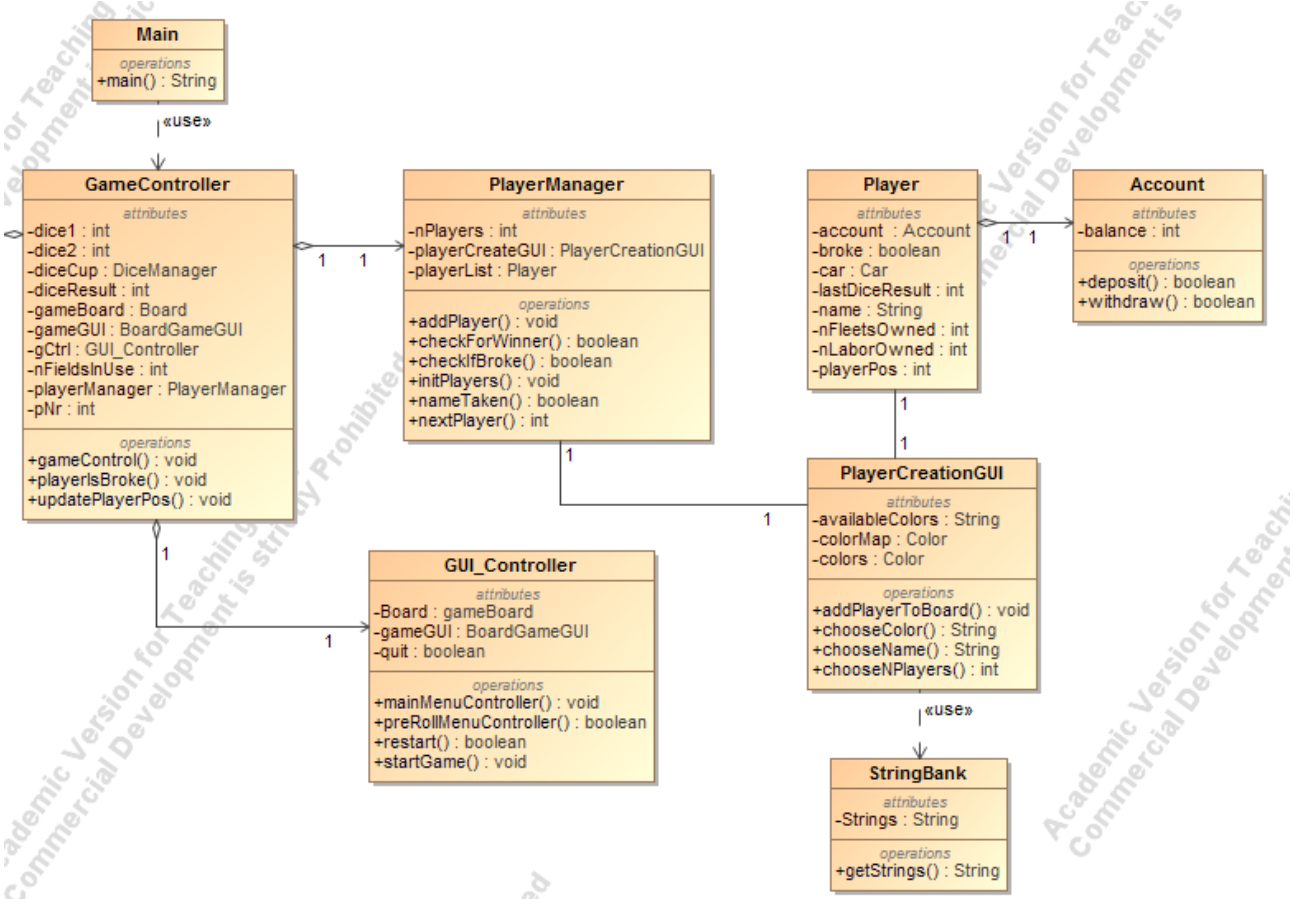
Sidste model er for funktionen ”Close”, hvis denne funktion bruges vil systemet lukke spillet ned:

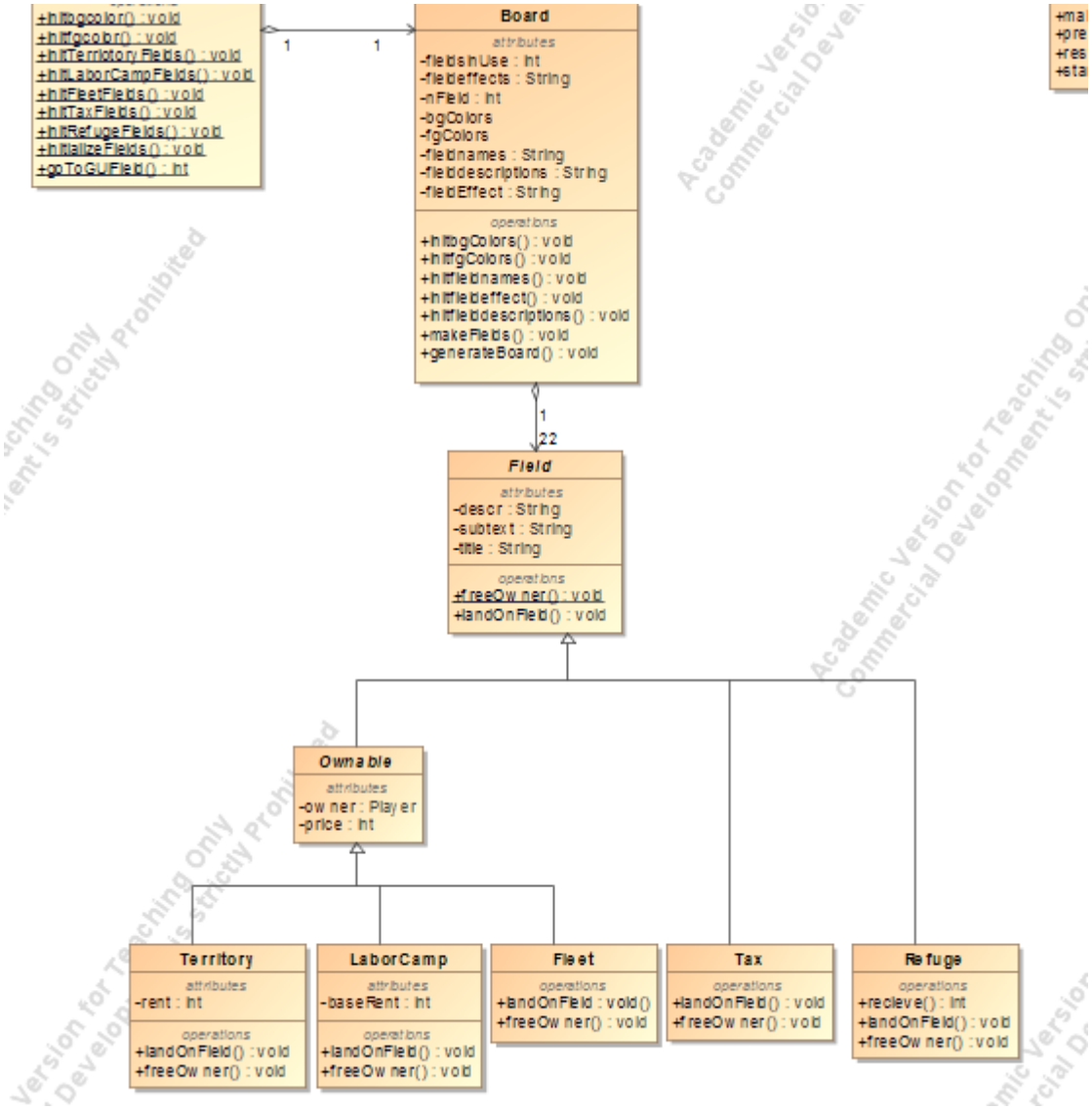


Design

Design klasse diagram

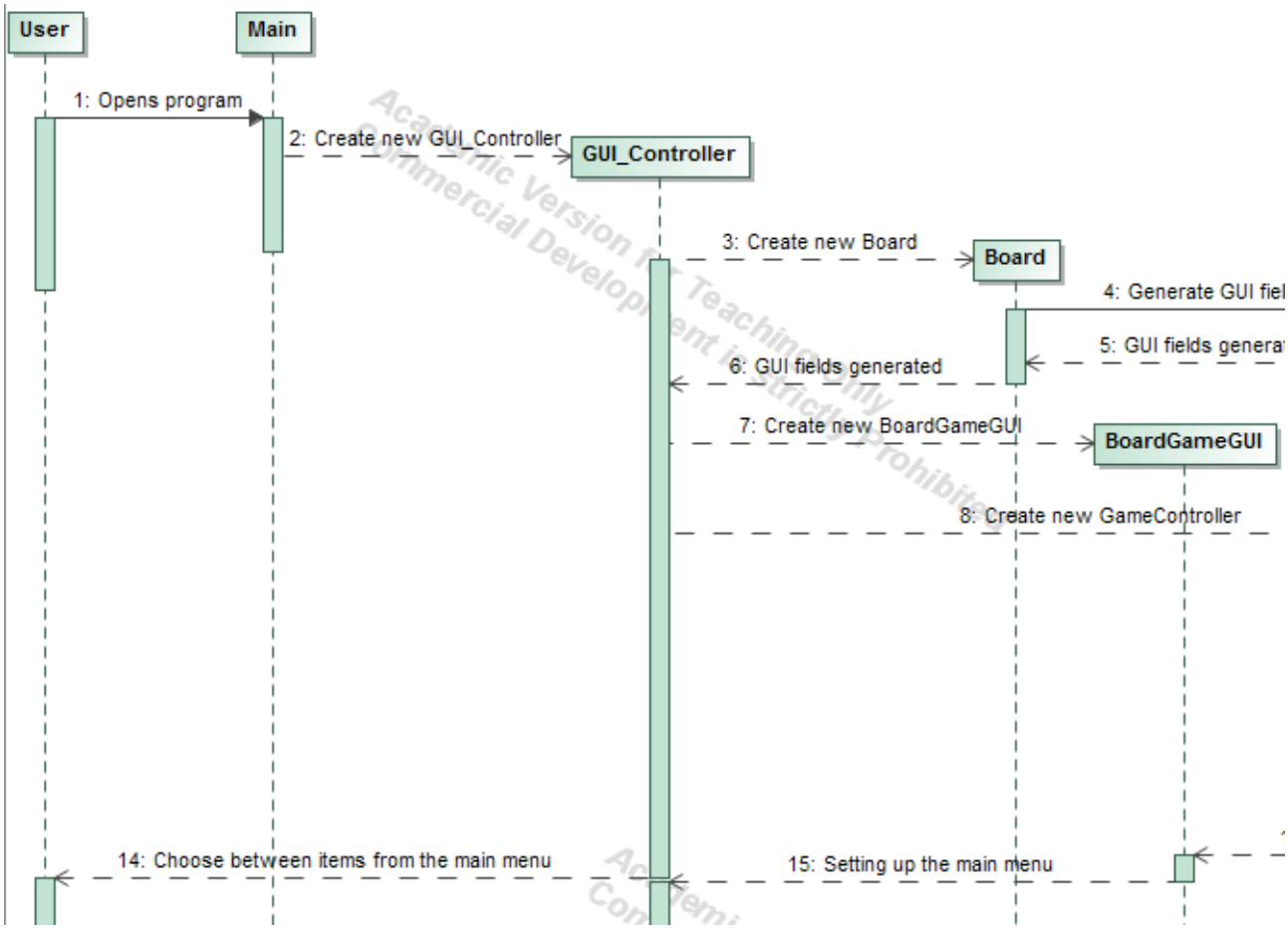


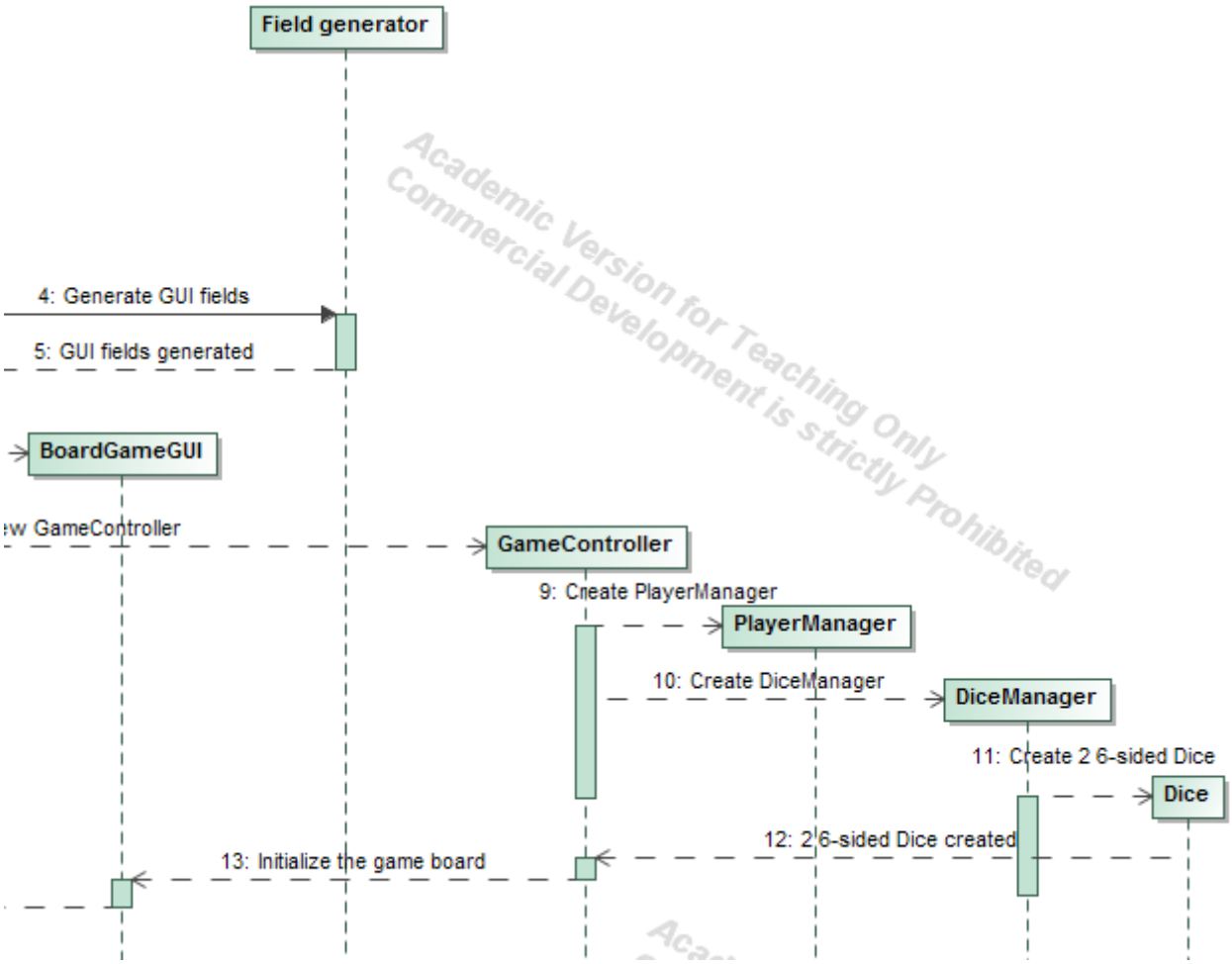

Main
operations
+main() : String
«use»
GameController
attributes
-dice1 : int
-dice2 : int
-diceCup : DiceManager
-diceResult : int
-gameBoard : Board
-gameGUI : BoardGameGUI
-gCtrl : GUI_Controller
-nFieldsInUse : int
-playerManager : PlayerManager
-pNr : int
operations
+gameControl() : void
+playerIsBroke() : void
+updatePlayerPos() : void
PlayerManager
attributes
-nPlayers : int
-playerCreateGUI : PlayerCreationGUI
-playerList : Player
operations
+addPlayer() : void
+checkForWinner() : boolean
+checkIfBroke() : boolean
+initPlayers() : void
+nameTaken() : boolean
+nextPlayer() : int
Player
attributes
-account : Account
-broke : boolean
-car : Car
-lastDiceResult : int
-name : String
-nFleetsOwned : int
-nLaborOwned : int
-playerPos : int
Account
attributes
-balance : int
operations
+deposit() : boolean
+withdraw() : boolean
GUI_Controller
attributes
-Board : gameBoard
-gameGUI : BoardGameGUI
-quit : boolean
operations
+mainMenuController() : void
+preRollMenuController() : boolean
+restart() : boolean
+startGame() : void
PlayerCreationGUI
attributes
-availableColors : String
-colorMap : Color
-colors : Color
operations
+addPlayerToBoard() : void
+chooseColor() : String
+chooseName() : String
+chooseNPlayers() : int
«use»
StringBank
attributes
-Strings : String
operations
+getStrings() : String
1
1
1
1
1
1
1
1
1
1
1
1

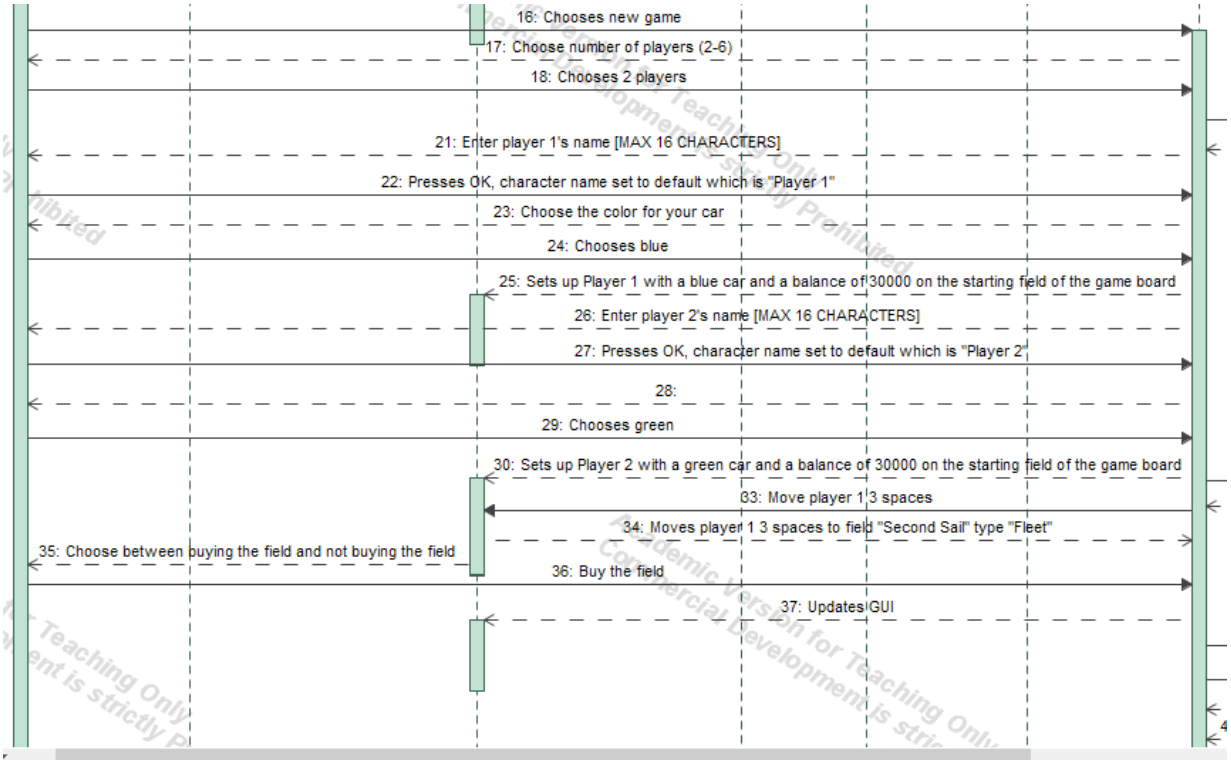


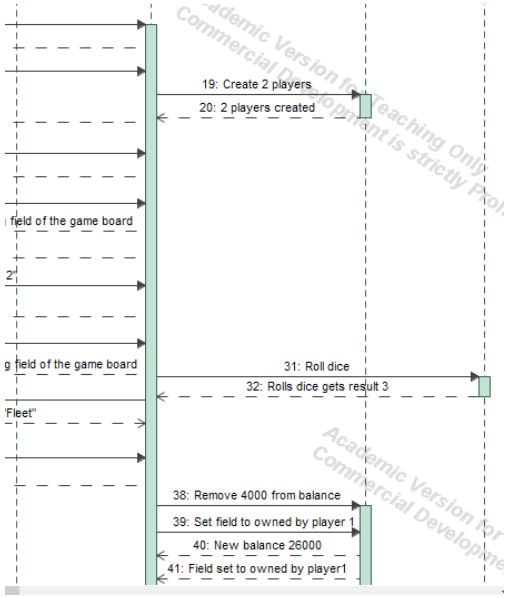
Design sekvens diagrammet beskriver relationerne klasserne, deres attributter og metoder. For at se det fulde design sekvens diagram, se vedhæftninger.

Design sekvens diagram




Field generator
4: Generate GUI fields
5: GUI fields generated
BoardGameGUI
ew GameController
GameController
9: Create PlayerManager
PlayerManager
10: Create DiceManager
DiceManager
11: Create 2 6-sided Dice
Dice
12: 2 6-sided Dice created
13: Initialize the game board


16: Chooses new game
17: Choose number of players (2-6)
18: Chooses 2 players
21: Enter player 1's name [MAX 16 CHARACTERS]
22: Presses OK, character name set to default which is "Player 1"
23: Choose the color for your car
24: Chooses blue
25: Sets up Player 1 with a blue car and a balance of 30000 on the starting field of the game board
26: Enter player 2's name [MAX 16 CHARACTERS]
27: Presses OK, character name set to default which is "Player 2"
28:
29: Chooses green
30: Sets up Player 2 with a green car and a balance of 30000 on the starting field of the game board
33: Move player 1 3 spaces
34: Moves player 1 3 spaces to field "Second Sail" type "Fleet"
35: Choose between buying the field and not buying the field
36: Buy the field
37: Updates GUI

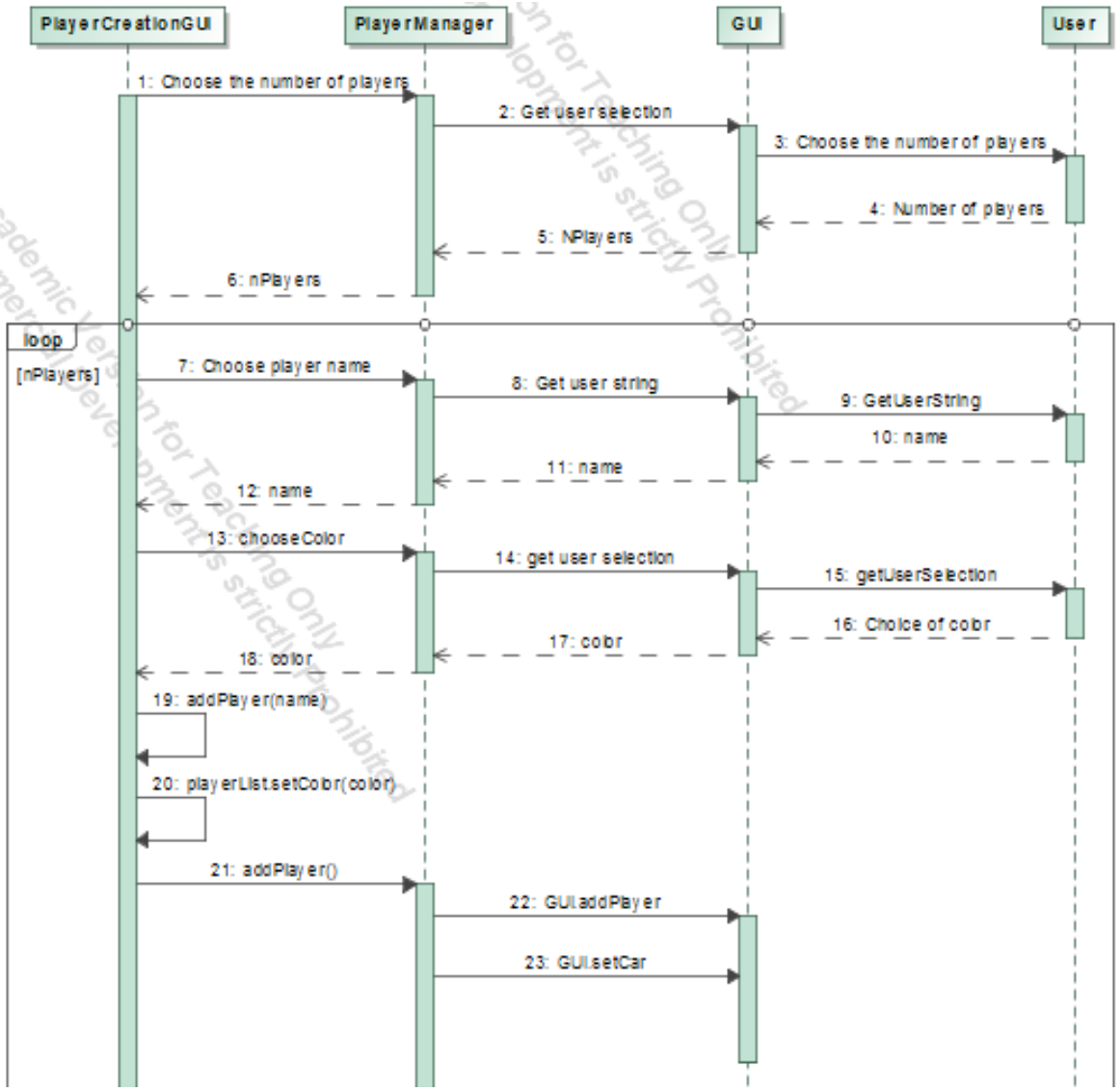


Se vedlagte MagicDraw filer for det fulde design sekvens diagram.

Dette diagram viser interaktionen mellem de forskellige dele af programmet og brugeren. I dette tilfælde kigges der, på den del, hvor spilleren starter et nyt spil og lander på det første ”Fleet” type felt: ”Second Sail”.
Det første der sker, efter at brugeren har åbnet programmet, er at vores main - metode initialiserer en GUI-Controller, der står for at generere spillebrættet og alle dets felter. Herefter oprettes en Game-Controller, der står for oprettelse af spillere og terninger. Efter brættet er blevet initialiseret, startes dette. Spilleren møder menuen, hvor spilleren får muligheden for at vælge mellem ”New game”, ”Rules of the game”, og ”Close” i en dropdown menu. I dette tilfælde antages det at spilleren vælger ”New game”. og derefter vælger at der er 2 spillere i alt. Spillene skal nu indtaste deres valgte navne og vælge farve til deres biler. Begge spillere starter ud med en balance på 30.000. Da spilleren har i sinde at starte spillet, vælges ”Roll dice”, hvor det samlede antal øjne er 3. Spilleren rykker til feltet ”Second Sail”, hvor muligheden for at købe feltet tilbydes. Spilleren køber feltet, hvorefter systemet trækker feltet pris (4000) fra spillerens balance. Spilleren sættes til feltets ejer, GUI’en opdateres, så det ses at spilleren nu ejer ”Second Sail”.

Initialize players

Følgende diagram er et udsnit af programmet der viser initialiseringen af spillerlisten. Klassen PlayerManager får først spilleren til at indtaste antallet af spillere. Hvorefter et loop køres, hvor alle spillere bedes indtaste deres navn og deres valg af figur farve. Disse informationer bliver tilføjet til PlayerManagers arraylist ’playerList’. Hvor man vha. getter og setter metoder kan tilgå disse informationer.


PlayerCreationGUI
PlayerManager
GUI
User
1: Choose the number of players
2: Get user selection
3: Choose the number of players
4: Number of players
5: NPlayers
6: nPlayers
loop
[nPlayers]
7: Choose player name
8: Get user string
9: GetUserString
10: name
11: name
12: name
13: chooseColor
14: get user selection
15: getUserSelection
16: Choice of colbr
17: colbr
18: colbr
19: addPlayer(name)
20: playerList.setColbr(colbr)
21: addPlayer()
22: GUI.addPlayer
23: GUI.setCar

Arv
Arv indenfor programmering handler om et bestemt forhold mellem klasser, der gør at en klasse kan nedarves fra en anden.

Hvad dette i praksis betyder er, at hvis man for eksempel gerne vil lave et program der skal holde styr på egenskaber for pattedyr og krybdyr, så i stedet for at lave to separate klasser, laver man i stedet en overordnet klasse der indeholder alle de ting som krybdyr og pattedyr har til fælles. Derefter laver man så en klasse med de ting der er specifikt for krybdyr, og en klasse med de ting som er specifikt for pattedyr, og så lader man de to klasser arve de ting der er fælles for krybdyr og pattedyr fra klassen dyr.

Abstract
Når en klasse er abstrakt betyder det at vi ikke er interesseret i at lave instanser af denne. Dette er typisk tilfældet når en klasse definerer en generalisering.

Hvis vi igen kigger på tilfældet med dyr for "Arv" ser vi at vi ikke på noget tidspunkt er interesseret i at lave instanser af klassen "dyr", da vi gerne vil holde styr på egenskaber for pattedyr og krybdyr. Altså er klassen "dyr" abstrakt.

Polymorfi
Det at alle fieldklasserne har en landOnField metode der gør noget forskelligt, kaldes polymorfi.

Polymorfi betyder at en objekt reference kan henlede til objekter af nedarvede klasser. Hvis vi igen bruger eksemplet med dyr, betyder det at vi kan sætte klassen dyr til at pege på objekter af både klassen ”pattedyr” og klassen ”krybdyr”.

Implementation

Overholdt GRASP

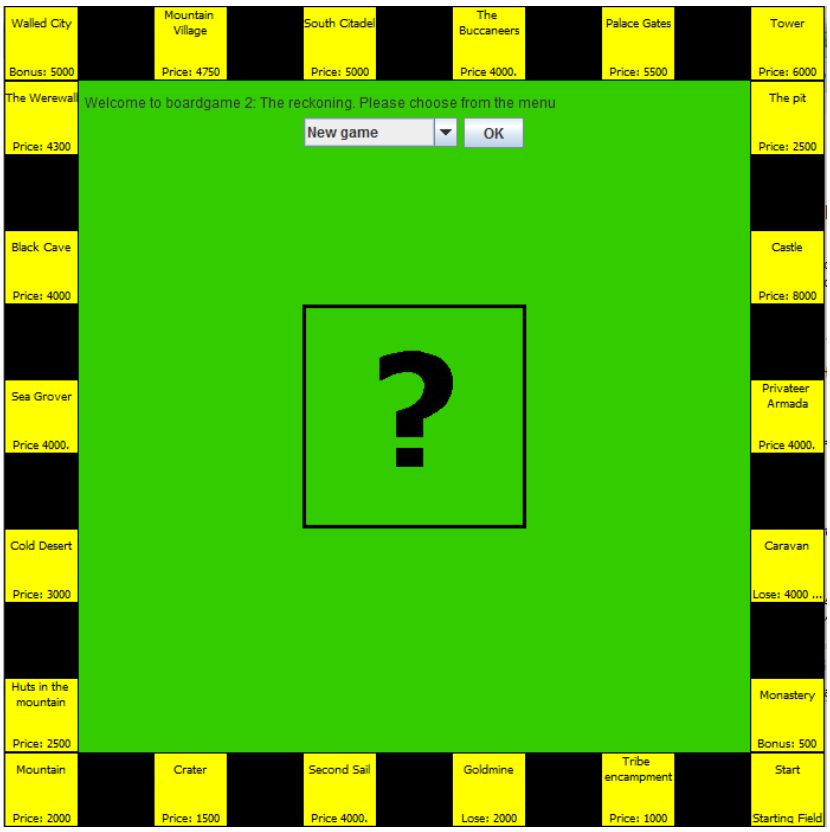
GRASP står for (General Responsibility Assignment Software Patterns), som er retningslinjer til objekt-orienteret-design. (OOD) I vores projekt endte vi med tre controllers (GUI_Controller, GameController og Board) en Field klasse, der er lavet efter polymorfiske principper og en del klasser, der både er information experts og creator f.eks. PlayerManager, der instantierer vores spiller objekter, og udfører operationer på dem. Vi har lavet boundary klasser, der sikrer at vi ikke har en direkte kobling til den supplerede GUI. Det understøtter også princippet om ”high cohesion” eller stærk sammenhængningskraft. Det har dog ikke altid været lige let at kunne overholde princippet om ”low coupling”, for at kunne overholde dette princip var vi f.eks. nødt til at indsætte flere argumenter i nogle metoder, i stedet for at passere et objekt, som argument. Det har gjort nogle metoder en smule svære at forstå, men ellers er det fornuftigt at overholde dette princip, da det nedsætter behovet for at omstrukturere en stor del kode, hvis man ændre et eller andet i en klasse.

Overordnet

Ca. Halvvejs gennem projektet, besluttede vi os at inddele programmet i en tre-lags arkitektur (Boundary, Controller, Entities). Vi havde sådan set allerede opdelt programmet i Controller og Entities, men havde ikke vores egen boundary klasse, da vi kaldte den supplerede GUI direkte. Det smarte ved at lave vores egen Boundary, som har til ansvar at styre input/output er, at man kan automatisere inputtet. Man kunne f.eks. lave en abstrakt GUI, så man har 2 konkrete gui’er der arver fra denne abstrakte GUI+, hvor i den ene, bliver inputtet automatiseret. Det ville gøre spillet noget nemmere at teste. Desværre løb vi tør for tid før vi kunne nå at lave den abstrakte gui og en gui til at teste med.

FieldGenerator og Board

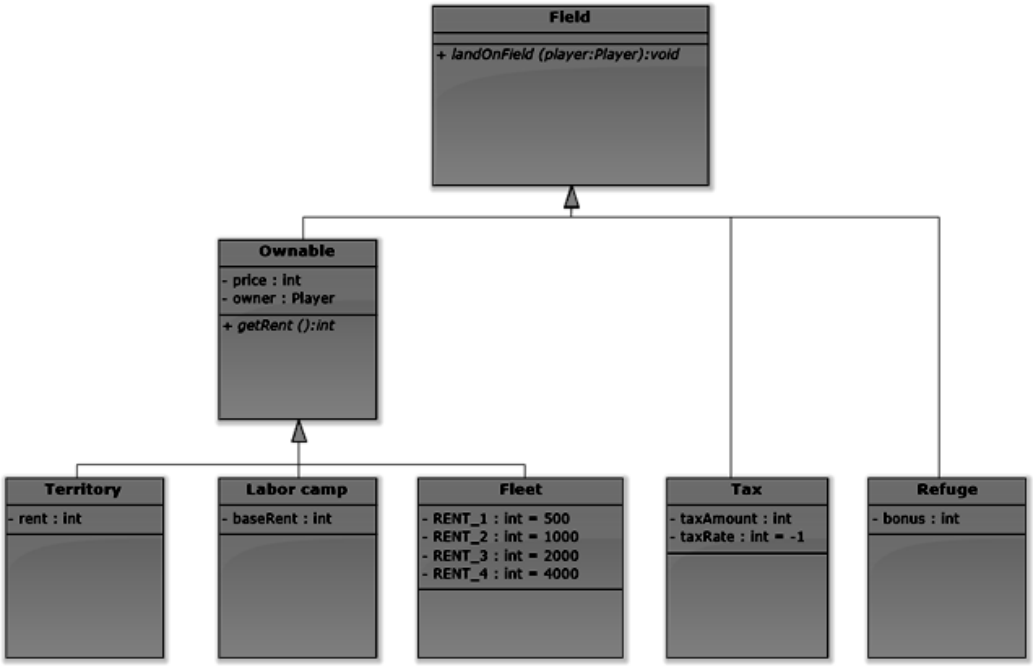
I GRASP termer, vil man nok sige at FieldGenerator har ansvaret, som information expert og creator med hensyn til field klasserne. I Første omgang ser klassen lidt forvirrende ud, men vi har bare valgt at opdele field typerne i sine egne arrays, med relevante informationer i tilhørende arrays. De to vigtigste arrays her er *fields[40]:Field* og *fieldsInUse[22]:int* grunden til at vi har disse to arrays er, fordi vi har valgt at "strække" vores 22 felter over de 40 felter, der går rundt omkring brættet i GUI'en. *fieldsInUse* siger, hvilke indeks vores rigtige felter skal have i GUI'ens 40 felter. FieldGeneratorens metoder *initTerritoryFields(), initFleetFields()...* osv. instantiere så vores fields i *fields* arrayet, man kan sige at metoderne "fylder" de forskellige arrays ud. Alle disse metoder bliver kaldt i vores Board klasses ene metode *generateBoard(),* her bliver vores Field objekter forbundet med GUI'ens Fields, sådan så vi faktisk kan vise GUI'en, da den ikke kan vises før man har kaldt metoden *GUI.create(fields:Field[]).*



Billede 1.0: vores felter udstrakt over GUI'ens felter

Field

Vi fik et forslag til hvordan den generelle struktur af Field klassen og dens subklasser skulle se ud. Og vi har fulgt den nogenlunde, dog har vi f.eks. givet Field klassen attributterne: *description , title , subtext* ,som primært er der for at klasserne skulle kunne samarbejde med GUI field klasserne. Og selvom man kan se at *Field* og *Ownable* ikke er i kursiv, altså de er ikke abstrakte, gjorde vi dem abstrakte, da vi var nødsaget til det, siden at vi ikke kunne lave metoderne abstrakte hvis klasserne ikke også var det. Vi har også en anden abstrakt metode i *Field* klassen:



Billede 1.1: forslag fra projektleder til struktur af Field klasser

freeOwner(player:Player, position: int) der sætter owneren af feltet til null, hvis den indsatte spiller referer til det samme sted i hukommelsen, som ejeren. Dvs. at de er det samme objekt. Grunden til at den også tager en 'position integer', er fordi den også kalder vores gui til visuelt at fjerne ejeren og der skal man bruge feltets position. Metoden var lidt af en "last minute" addition til Field klassen, da vi begyndte at sætte alle delene sammen. Det giver ikke særlig god sammenhængningskraft i koden, når metoden type sættes i *Field* klassen, og ikke *Ownable* Klassen. Det havde vi gjort fordi da vi skulle fjerne ejere, var vi nødt til at iterere over alle vores Fields i Field-arrayet, der ligger i FieldGenerator klassen. Vi havde ikke noget array kun til Ownable Fields. Det er ærlig talt lidt spaghetti kode agtig, hvis man må være lidt selvkritisk. Et forslag til forbedring, kunne være at give Field klassen en int, der gemmer deres egen position på brættet og i spiller klassen kunne man evt. Lave tre *ArrayList<int>*(en til territories, en til fleets og en til labor camps), hvor spillerens ejede felters positioner ligger. Så behøver man ikke itererer over alle felter, men kun spillerens array lister. Derefter kunne man jo også bruge listernes størrelser til at angive, hvor mange Labor camps, eller fleets, spilleren ejer.

Ownable

I Ownable klassen har vi ikke afveget fra forslaget, dog har vi getters/setters metoder, som vi også har i alle vores andre klasser. De vises bare i billede 1.1, da sådanne metoder er indforståede.

Territory/Labor camp/Fleet

De tre felter, der kan ejes har en fælles struktur i deres *landOnField()* metode. Hvis feltet ikke har nogen ejer, og spilleren har penge nok til at købe feltet, får han muligheden til at købe feltet. Hvis han ejer feltet, bliver der sendt en besked til GUI'en at han er landet på sit eget felt, og ejer en anden feltet, så må han jo

betale til ejeren. Noget vi gjorde lidt anderledes end i forslaget er i Fleet klassen. I stedet for at have 4 ints til renten, havde vi én int på 500 og bestemte derfor den endelige rente ud fra formlen $rente = basisrente \cdot 2^{n-1}$ hvor n er det antal fleets ejeren ejer. Det gjorde vi da vi nemt kunne se at for hver fleet ekstra fordoblede renten.

```
else if(this.getOwner()!=p){
    int fleetsOwned = this.getOwner().getFleetsOwned();
    fleetsOwned = (int)(500*Math.pow(2, fleetsOwned-1));
```

Her er fleetsOwned den endelige rente.

Billede 1.2: eksempel fra Fleet klassens landOnField() metode

PlayerManager

Denne klasse har til ansvar at skabe spiller objekter, og holde styr på dem i en arrayListe. Derudover kan den lave en række operationer på spiller klasserne. I GRASP termer ville man kalde den: ”Information expert” og ”Creator” med hensyn til Player klassen. Den har metoder, som *nextPlayer()*, der skifter turen til den næste spiller i array listen, som ikke er gået fallit. Dvs. Den returnerer en int, som er den næste spillers indeks i array listen. *checkForWinner()* returnerer en boolean, som er sand, hvis der kun er én spiller tilbage i array listen, som ikke er gået fallit. *nameTaken(name:String)* returnerer en boolean. Den er sand hvis en spiller i array listen allerede har det samme navn, som det indsatte. Og her er navne case sensitive. Det er vigtigt at have unikke navne, da GUI'en sortere spillere efter deres navne. *checkIfBroke()* kan tjekke om en spiller er gået fallit, hvorefter den vil returnere true, hvis han er fallit. *initPlayers()* er den metode, der bliver kaldt, når man gerne vil instantiere nogle nye spillere. Først vælger man hvor mange spillere man vil have, derefter vælger hver spiller navn og farve på bil. Skriver man ikke noget navn, får man et standard navn som ”Player 1” , ”Player 2” osv. I billede 1.3 kan man se at inputtet foregår via. playerCreateGui, som er et objekt af PlayerCreationGUI. I den første iteration af PlayerManager klassen foregik alt input inde i klassen selv, men da vi begyndte at lave vores egne boundary klasser flyttede vi alt, der havde med at skabe spillere over i sin egen klasse. Det gjorde vi bl.a fordi det understøttede principperne om High cohesion, og denne GUI benytter sin egen stringbank: PlayerCreation_Stringbank. Den anden Gui vi har er BoardGameGUI, som bruges af selve spillet og start menuen.

```
public void initPlayers()
{
    //First choose number of players
    nPlayers = playerCreateGUI.chooseNPlayers();

    for(int i = 0; i<nPlayers;i++)
    {
        String name = playerCreateGUI.chooseName(this);
        String color = playerCreateGUI.chooseColor();//Chooses the color for the car
        addPlayer(name);
        playerList.get(i).setCar(playerCreateGUI.getColor(color));
        playerCreateGUI.addPlayerToBoard(playerList.get(i));

    }

}
```

Billede 1.3: initPlayers i playerManager klassen

Player

```
private Car car;
private String name;
private Account account;
private boolean broke;
private int playerPos;
private int nFleetsOwned;
private int nLaborOwned;
private int lastDiceResult;
```

Billede 1.4: Player klassens attributter

Denne gang valgte vi at gemme spillerens position i spiller-objektet, i forhold til sidste projekt. I stedet for i et array i GameControlleren. Det gjorde vi blandt andet, fordi vi skulle bruge positionen andre steder end kun GameControlleren. F.eks. bruger vi positionen i landOnField metoden i vores Field klasser. Som man kan se i billede 1.5 tjekkes, der først om feltet har en ejer, og om spilleren har en tilstrækkelig balance, derefter, hvis man nu har valgt at købe feltet, bliver positionen brugt til at sætte ejeren i vores gui, så det er muligt, at se hvem der ejer feltet. Hvis vi ikke havde valgt at gemme positionen i spiller klassen, ville vi være nødt indsætte den som et andet argument i metoden, og det ville have ødelagt polymorfien. Grunden til at vi har spillerens sidste terning resultat som attribut, findes senere i samme metode.

```
public void landOnField(Player player){
    BoardGameGUI gui = new BoardGameGUI();
    if(super.getOwner() == null){
        if(player.getAccount().getBalance()>this.getPrice()){
            String input = gui.buyMenu(this.getTitle(), this.getPrice(), this.getRent());
            if(input.equals(Game_Stringbank.getFieldMsg(0)))
            {
                this.setOwner(player);
                player.setLaborOwned(player.getLaborOwned()+1);
                player.getAccount().withdraw(this.getPrice());
                gui.setOwner(player.getPlayerPos(), player.getName());
            }
```

Billede 1.5: første del af landOnField metode for labor camps

Hvis en anden spiller ejer feltet bliver renten bestemt efter funktionen $Rente = 100 \cdot LaborCampsEjet \cdot TerningResultat$, hvor igen hvis vi indsætter terningresultatet, som argument i metoden, vil det ødelægge polymorfien.

Metoderne i Player klassen er ikke specielt interessante, den har sådan set bare de normale getters/setters, og en konstruktør, der starter med at give spilleren en bil med tilfældig farve. Det eneste, der måske er lidt af en afvigelse er at setCar metoden modtager en farve, som argument og ikke et Car-objekt, dog bliver et Car-objekt med den indsatte farve instantieret.

DiceManager/Dice

Vores DiceManager er kopireret fra vores sidste projekt, den kan lave Dice objekter og gemme dem i en array liste, derudover kan den rulle med dens terninger og returnere hver ternings værdi samt summen af alle terningernes værdier. Konstruktøren tager som argumenter antal sider på terningerne og antal

terninger. Så man kan faktisk lave andre sidet terninger end 6 sidet terninger, men GUI'en understøtter kun 6 sidet terninger.

GUI_Controller

Dette er den første controller man tilgår, når programmet startes op. Det er en form for menu controller, hvor man kan vælge at starte et nyt spil. Ud over start menuen er der også en menu controller metode til preRollMenuen, eller den menu man ser i spillet, hvor man kan vælge at kaste med terningerne eller gå tilbage til startmenuen. Al input/output foregår via BoardGameGUI, så dette er faktisk ikke en boundary klasse. Det er også i GUI_Controller klassen, at metoden *generateFields()* i Board klassen kaldes, da vi er nødt til at registrere alle vores fields i GUI'en før vi kan lave nogle andre GUI relateret operationer.

GameController

Denne klasse er så selve spil controlleren, og spil logik centeret. *playerIsBroke()* metoden eksekverer en række operationer, når en spiller er gået fallit, så som at fjerne hans bil fra spilbrættet og frisætte alle hans ejede felter. Metoden tjekker ikke selv om spilleren er gået fallit, men bliver kun kaldt, når *checkIfBroke()* fra playerManager klassen returnerer en sand værdi. *updatePlayerPos()* denne metode rykker spilleren trinvis frem på spilbrættet ved brug af en forløkke. Grunden til at vi bruger en forløkke og ikke modulus, er fordi vi på hvert trin, skal tjekke om spilleren er landet på startfeltet, da start feltet skal ignoreres.

```
while(gCtrl.preRollMenuController(playerManager.get(pNr).getName()))
{

    diceCup.rollDice();
    diceResult = diceCup.getDiceTotal();
    dice1 = diceCup.getDiceValue(0);
    dice2 = diceCup.getDiceValue(1);
    gameGUI.showDiceRolling(dice1, dice2);
    playerManager.get(pNr).setDiceResult(diceResult);
    gameGUI.movePlayerModel(playerManager.get(pNr).getName(), playerManager.get(pNr).getPlayerPos(), diceResult);
    updatePlayerPos(pNr,diceResult);
    FieldGenerator.getFields(FieldGenerator.getFieldsInUse(
            playerManager.get(pNr).getPlayerPos())).landOnField(playerManager.get(pNr));
    if(playerManager.checkIfBroke(pNr))
    {
        playerIsBroke(pNr);
    }
    gameGUI.updatePlayerBalance(playerManager.getPlayerList());
    pNr = playerManager.nextPlayer(pNr);
    if(playerManager.checkForWinner())
    {
        gameGUI.showWinnerMsg(playerManager.get(pNr).getName());
        break;
    }

}
```

Billede 1.6 while løkke i gameControl() metoden i GameController klassen

Startfeltet er kun til spillernes første tur. Det skal siges at *updatePlayerPos()* rykker ikke på spillerens bil på spilbrættet. Det er *movePlayerModel()* i BoardGameGUI, der gør det. Pga. Opdelingen i tre-lags arkitekturen, ville vi holde disse operationer adskilt. Spillet ligger så i metoden *gameControl()*

Dette er selve spillet, der kører, når man har lavet spillerne. While løkken kører, når man vælger at kaste med terningerne (gCtrl er et GUI_Controller objekt) efter terninger er kastet, viser vi terningekastet i GUI'en, derefter flytter vi spillerens bil og opdaterer hans position. Spilleren lander på et felt, og vi kalder derfor feltets *landOnField()* metode. Efter han er landet på et felt, kan det være muligt at han er gået fallit, så vi tjekker om han er gået fallit, og hvis han er, kalder vi *playerIsBroke()*, herefter opdaterer vi alle spilleres balance i GUI'en og skifter til næste spiller. Bagefter tjekker vi så, om der er en vinder, og hvis der er en vinder må det jo være den spiller vi har skiftet over til, da *nextPlayer()* gav indekset til den næste spiller, der ikke er gået fallit.

Test

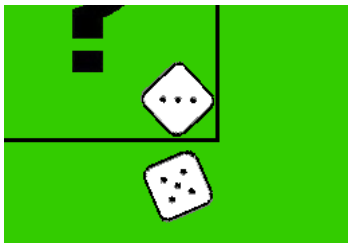
Test af krav specifikationer

Test ID: 1

Krav testet	Input	Forventet Output	Faktisk Output	Ok?
F1	”New Game” ”Please Choose number of Players” “2” “ok”	Opretter 2 spillere	Spillet opretter 2 spillere	ja

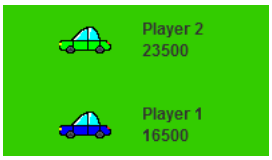
Test ID: 2

Krav testet	Input	Forventet Output	Faktisk Output	Ok?
F2 **F21**	Observer I GUI	Systemet benytter 2 terninger med 6 sider. Systemet skal simulere kastet.	Systemet benytter 2 terninger med 6 sider, som bliver simuleret i et kast.	ja



Test ID: 3

Krav testet	Input	Forventet Output	Faktisk Output	Ok?
F3	Observer I GUI	Man kan hele tiden se hver spillers balance	Man kan hele tiden se hver spillers balance	ja

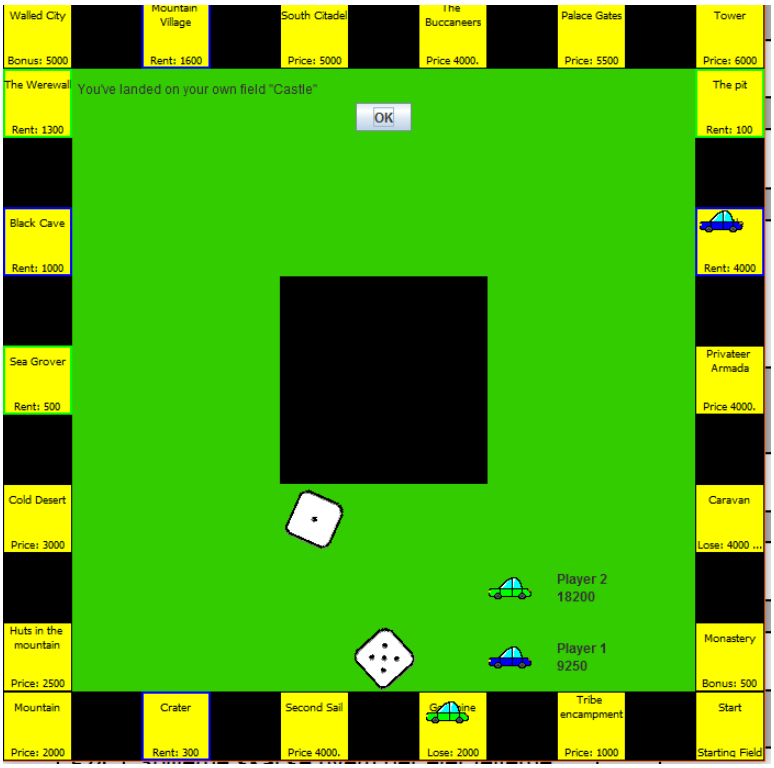


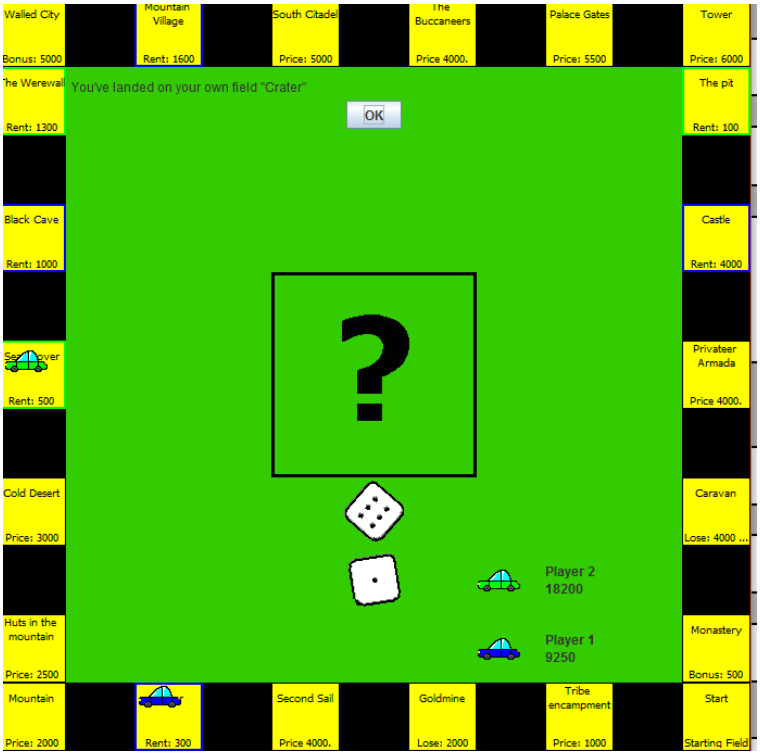
Test ID: 5

Krav testet	Input	Forventet Output	Faktisk Output	Ok?
F5	Observer spillernes balance får der fortages det første kast.	Alle spillere har en start balance på 30000	Alle spillere har en start balance på 30000	ja

Test ID: 4

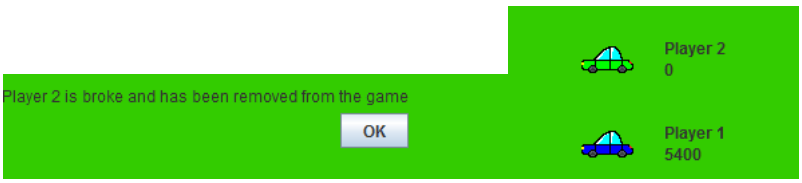
Krav testet	Input	Forventet Output	Faktisk Output	Ok?
F4	Observer spillerne bevæge sig på pladen	Player 1 skal springe ”startfeltet” over	Player 1 slår 7 og går 7 felter frem, hvor spilleren springer ”start” over	ja





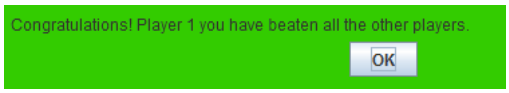
Test ID: 6

Krav testet	Input	Forventet Output	Faktisk Output	Ok?
F6	”New game” ”Choose number of players” ”choose name” “Choose car color” “Roll the dice” - forsæt indtil en spillere er bankerot	En spiller går bankerot når han ikke har flere penge	Spilleren går bankerot hvis ikke han har flere penge	ja



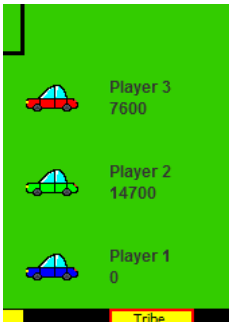
Test ID: 7

Krav testet	Input	Forventet Output	Faktisk Output	Ok?
F7	”New game” ”Choose number of players” ”choose name” “Choose car color” “Roll the dice” - forsæt indtil alle på nær en spiller er bankerot	Hvis den anden sidste spiller går bankerot er det den tilbageværende spiller som vinder.	Sidste levende spiller vinder.	ja



Test ID: 8

Krav testet	Input	Forventet Output	Faktisk Output	Ok?
F8	Observer turrunden, når en spiller er gået bankerot	En bankerot spiller får ikke flere ture og bliver sprunget over	Spilleren rykker fremad fra det felt spilleren stod på da spilleren kastede med terningerne.	ja

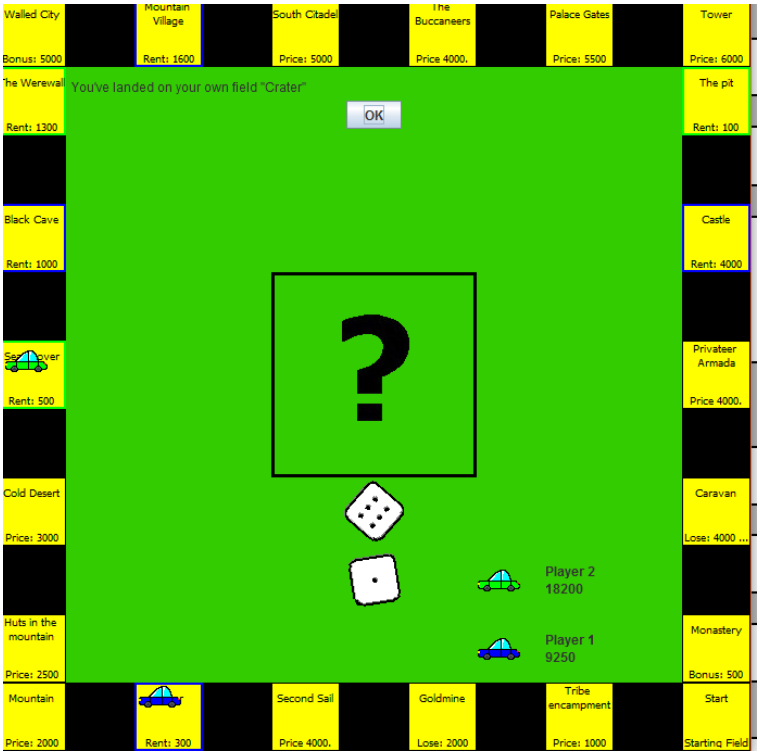


Test ID: 9

Krav testet	Input	Forventet Output	Faktisk Output	Ok?
F9	observer	Spilleren skal bevæge sig rundt på brættet	Spillerne bevæger sig rundt på brættet	ja

Test ID: 10

Krav testet	Input	Forventet Output	Faktisk Output	Ok?
F10	observer	Spillet skal indeholde 22 felter inklusiv et startfelt	Spillet indeholder 22 felter incl. Et startfelt.	ja

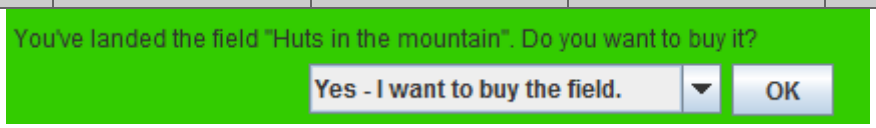


Test ID: 11

Krav testet	Input	Forventet Output	Faktisk Output	Ok?
F11	”New game” ”Choose number of players” ”choose name” “Choose car color” “Roll the dice” - forsæt indtil at spilleren er landet på hver type felt.	Hver type Felt er en unik effekt	Hver type felt har en unik effekt med varieret ift. høj/lav afgift/gevinst	ja

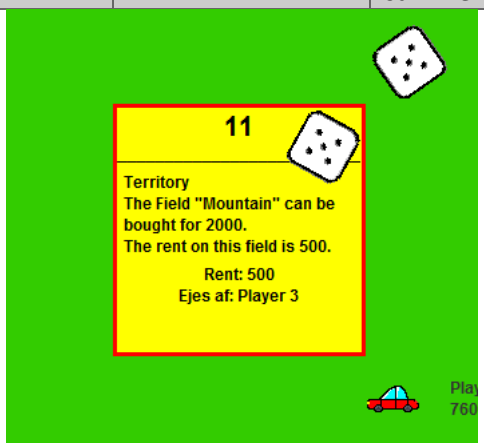
Test ID: 12

Krav testet	Input	Forventet Output	Faktisk Output	Ok?
F12	”New game” ”Choose number of players” ”choose name” “Choose car color” “Roll the dice” - Observer	Systemet printer en besked	Se billede	ja



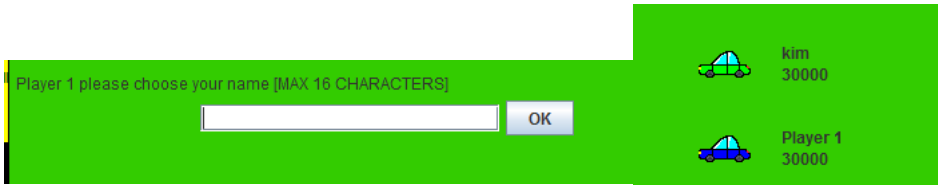
Test ID: 13

Krav testet	Input	Forventet Output	Faktisk Output	Ok?
F13	Hold musen over et felt og observer	Hvert felt har en unik beskrivelse	Hvert felt har en unik beskrivelse på når de 4 Fleet-felter som har samme beskrivelse	Delvis felter af typen ”Fleet”, har samme beskrivelse



Test ID: 14

Krav testet	Input	Forventet Output	Faktisk Output	Ok?
F14 **F15**	”New game” ”Choose number of players” ”choose name”	Spilleren har mulighed for at vælge et unikt navn, på maks. 16 tegn	Spilleren kan vælge sit eget unikke navn på maks. 16 tegn, ellers gør systemet det for spilleren.	ja



Test ID: 15

Krav testet	Input	Forventet Output	Faktisk Output	Ok?
F16	”New game” ”Choose number of players” ”choose name” “Choose car color” “Roll the dice” “Roll the dice”	Spilleren skal rykke frem fra det felt spilleren stod på, før sit kast	Spilleren rykker frem fra feltet hvor spilleren stod på, før sit kast.	ja

Test ID: 16

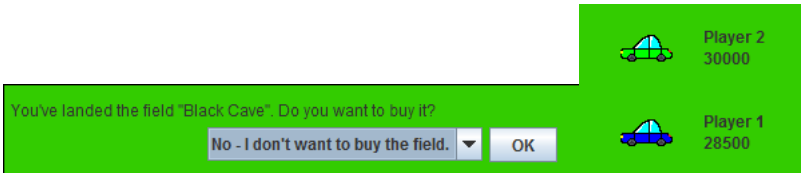
Krav testet	Input	Forventet Output	Faktisk Output	Ok?
F18	”New game” ”Choose number of players” ”choose name” “Choose car color” “Roll the dice” - forsæt indtil en spiller er bankerot	Alle spillerens felter går til banken, og kan købes igen af de resterende spillere	En bankerot spillers felter bliver sat frie, til at kunne blive købt igen	ja

Test ID: 17

Krav testet	Input	Forventet Output	Faktisk Output	Ok?
F19	”New game” ”Choose number of players” ”choose name” “Choose car color” “Roll the dice” - forsæt indtil en spiller lander på et felt spilleren kan købe	Spilleren køber feltet og er nu ejer	Efter spilleren har købt feltet, ejer han det.	ja

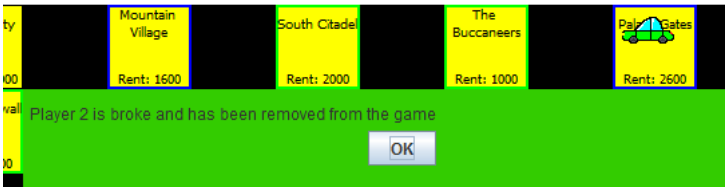
Test ID: 18

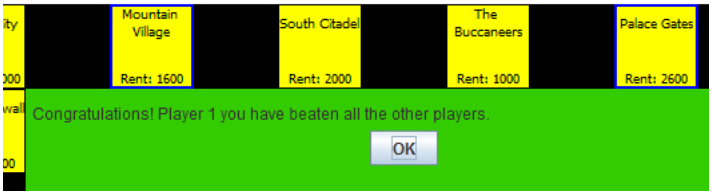
Krav testet	Input	Forventet Output	Faktisk Output	Ok?
F20	”New game” ”Choose number of players” ”choose name” “Choose car color” “Roll the dice” - forsæt indtil en spillere lander på et felt ingen ejer og som kan købes	Spilleren lander på et ikke-ejet felt og får tilbudt at købe eller skippe. Vælger spilleren ikke at købe sker der ikke noget med spillerens balance	Spilleren vælger ikke at købe og balancen forbliver den samme	ja



Test ID: 19

Krav testet	Input	Forventet Output	Faktisk Output	Ok?
F22	”New game” ”Choose number of players” ”choose name” “Choose car color” “Roll the dice” - forsæt indtil en spiller er bankerot	Den bankerotte spillers bil bliver fjernet fra brættet	Bilen forsvinder når spilleren går bankerot.	ja



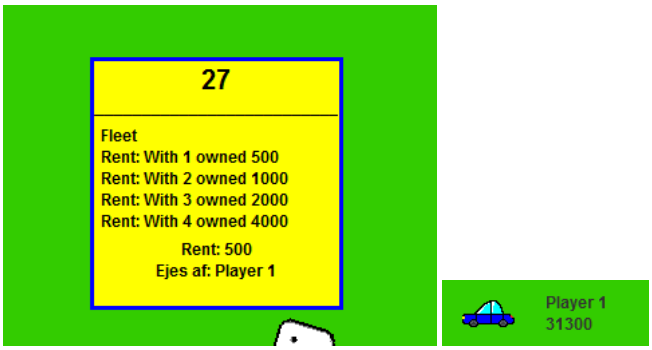


Test ID: 20

Krav testet	Input	Forventet Output	Faktisk Output	Ok?
F23	”New game” ”Choose number of players” ”choose name” “Choose car color”	Spilleren skal selv vælge bilens farve	Spilleren skal vælge bilens farve alt efter hvilken farve der er tilbage at vælge imellem.	ja

Test ID: 21

Krav testet	Input	Forventet Output	Faktisk Output	Ok?
F24	Observer	Spilleren kunne se hvem der ejer felterne	Spilleren kan se hvem der ejer felterne, da der kommer en farvet omrids rundt om det ejede felt, som er lig farven på bilen som ejeren har.	ja

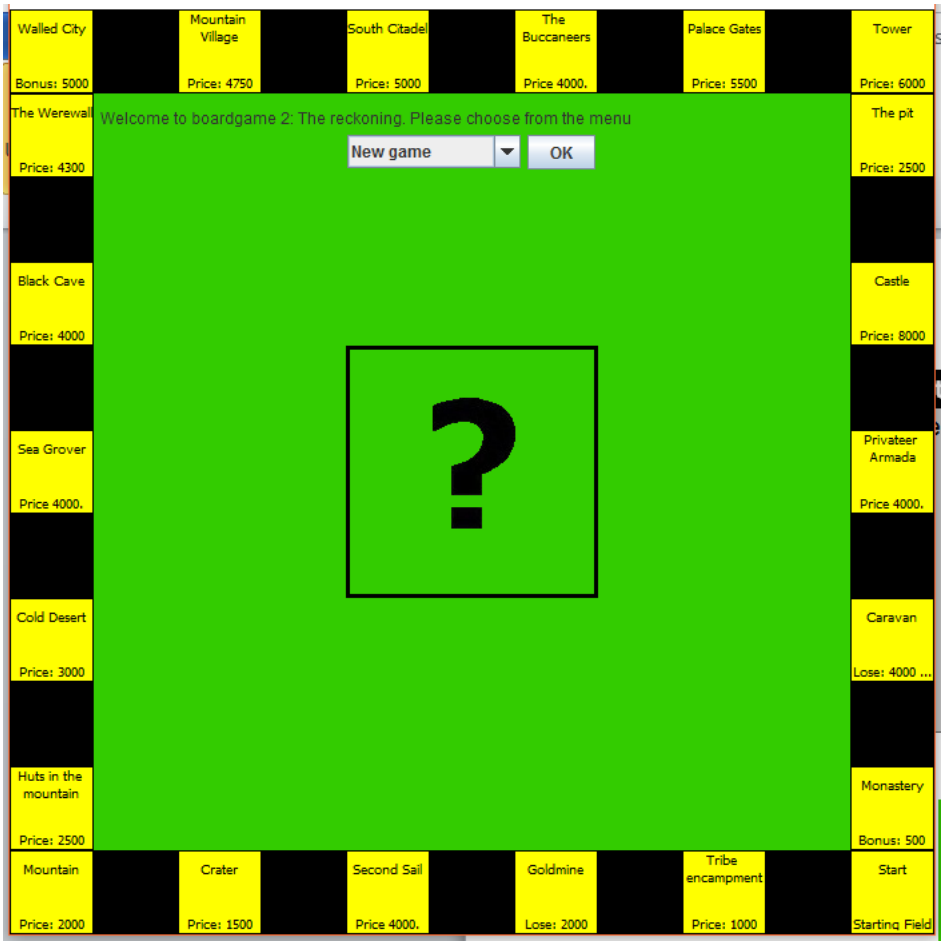


Test ID: 22

Krav testet	Input	Forventet Output	Faktisk Output	Ok?
F25	Observer	Spillet er på engelsk	Spillet er på engelsk	ja

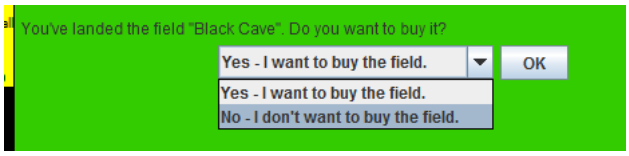
Test ID: 23

Krav testet	Input	Forventet Output	Faktisk Output	Ok?
F26 **F27**	Observer	Felterne skal have titler som bliver vidst hele tiden.	Felterne har hver en titel, som er vidst i GUI hele tiden.	ja



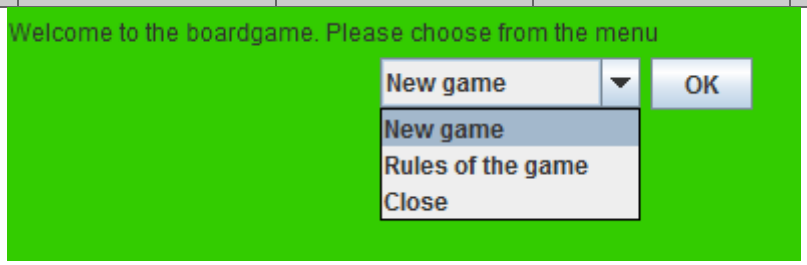
Test ID: 24

Krav testet	Input	Forventet Output	Faktisk Output	Ok?
F28	”New game” ”Choose number of players” ”choose name” “Choose car color” “Roll the dice” - forsæt indtil en spillere lander på et felt spilleren kan købe	Spilleren skal have muligheden for at købe et felt, eller gå videre, hvis spilleren ikke ønsker at købe det.	Spilleren får mulighed for at købe eller skippe.	ja



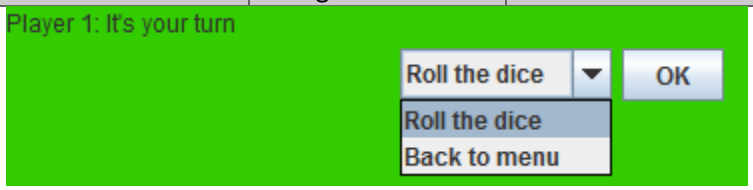
Test ID: 25

Krav testet	Input	Forventet Output	Faktisk Output	Ok?
F29	Start spil ”New game” ”Choose number of players” ”choose name” “Choose car color” Observer	Spilleren får en menu	Se billede	ja



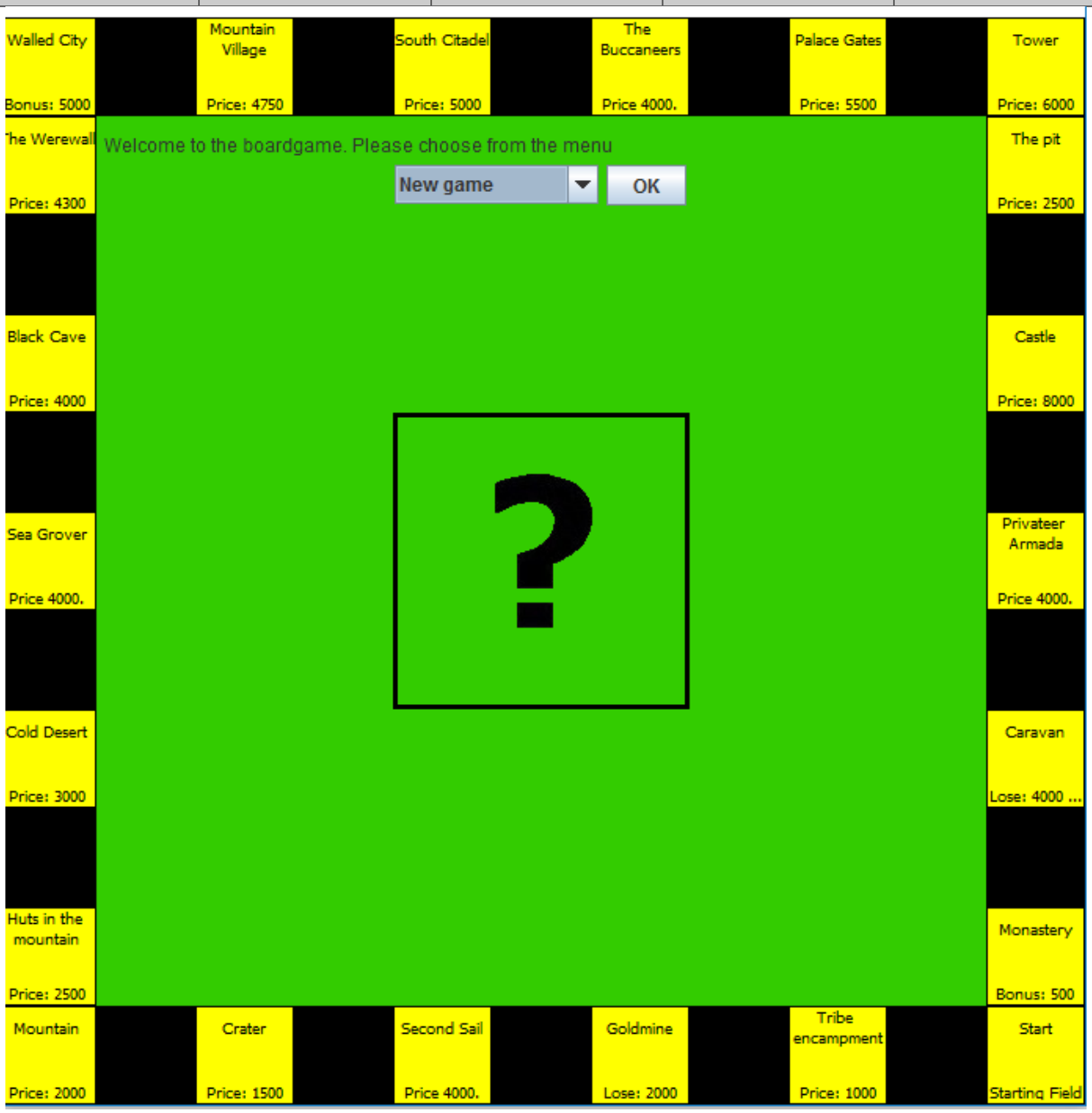
Test ID: 26

Krav testet	Input	Forventet Output	Faktisk Output	Ok?
F30	Start spil Observer	Spilleren får en mulighed for at gå tilbage til menuen	Se billede	ja



Test ID: 27

Krav testet	Input	Forventet Output	Faktisk Output	Ok?
F31	Start spil Observer	Spillet vises via en GUI	Se billede	ja

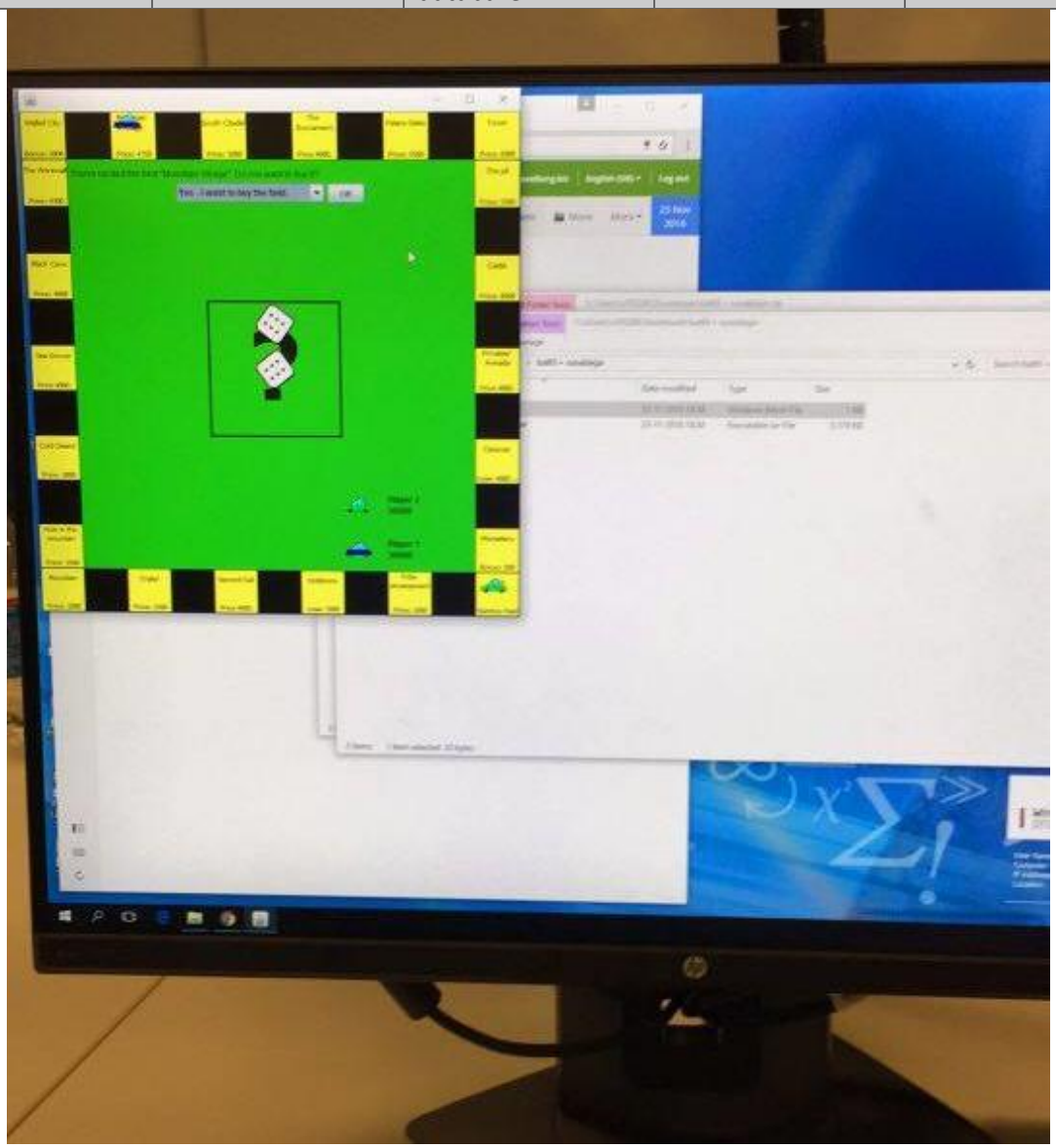


Test ID: 28

Krav testet	Input	Forventet Output	Faktisk Output	Ok?
F35	Start spil ”New game” ”Choose number of players” ”choose name” “Choose car color” “Roll dice” Observer	Spillet simulerer rykning af figur	Spillet simulerer rykning af figur	ja

Test ID: 29

Krav testet	Input	Forventet Output	Faktisk Output	Ok?
NF2	Start spil	Spillet kan køres på DTU’s databarer	Se billede	ja



Testmatrix

Krav		F1	F2	F3	F4	F5	F6	F7	F8	F9	F10	F11	F12	F13	F14	F15	F16	F17	F18
ID:	1	■																	
ID:	2		■																
ID:	3			■															
ID:	4				■														
ID:	5					■													
ID:	6						■												
ID:	7							■											
ID:	8								■										
ID:	9									■									
ID:	10										■								
ID:	11											■							
ID:	12												■						
ID:	13													■					
ID:	14														■	■			
ID:	15																■		
ID:	16																		■
		F19	F20	F21	F22	F23	F24	F25	F26	F27	F28	F29	F30	F31	F32	F33	F34	NF2	
ID:	2			■															
ID:	17	■																	
ID:	18		■																
ID:	19				■														
ID:	20					■													
ID:	21						■												
ID:	22							■											
ID:	23								■	■									
ID:	24										■								
ID:	25																		
ID:	26											■							
ID:	27												■						
ID:	28													■					
ID:	29														■				
ID:	30																■		
ID:	31																	■	

Konklusion

Vi har fremstillet et spil, ud fra den opstillede kravsspecifikation, som blev udarbejdet ud fra kundens vision samt vores egne forslag. Vi har testet spillet og kan konkludere, at det lever op til kravsspecifikationen, dog har det nogle ikke-funktionelle mangler, såsom når en spiller går bankerot, bliver feltets beskrivelse ikke opdateret. Det burde vise prisen i stedet for renten. Noget andet er når man har gennemført et spil, og starter et nyt spil op, bliver spillernes balance ikke opdateret ordentligt, dette løses dog efter det første kast med terningerne. Disse fejl er kun grafiske, og spillet virker tilfredsstillende rent funktionsmæssigt.

I forhold til sidste projekt har vi forbedret vores planlægning. Konsekvenserne af dette er en mere fornuftig fordeling omkring de forskellige discipliner i udviklingsprocessen. Det er ikke perfekt, men det går den rigtige retning.

Kilder:

- Software GUI (Ronnie Dalsgaard (s093487) with input and adjustments by Daniel Rubin-Grøn (daniel@koru.dk))
- Feltliste : Link

Bilag

Use case : RollDice

Use Case:	RollDice
Id:	7
Kort beskrivelse:	Spillerene slår med terningen og rykker det antal øjne terningerne tilsammen viser frem på pladen. Spilleren lander på et felt, som alle har en effekt.
Primær aktør:	Spilleren
Sekundær aktør:	
Forudsætning:	Spillerne skal være oprettede
Main flow:	7. Use Case startet når ChooseCarColor har kørt 8. Systemet beder spilleren om at slå med terningerne. 9. systemet viser summen af terningerne 10. systemet rykker spillerens brik frem på spilpladen, lig summen af terningerne. *Spilleren lander på Tribe Encampment* 11. Systemet spørger om spilleren vil købe feltet 12. Systemet sætter feltet til at være ejet af den pågældende spiller
Efterfølgende:	Næste spillers tur
Alternativt Flow:	**Alternativt Flow1** 1: Use Case startet når ChooseCarColor har kørt 2: Systemet beder spilleren om at slå med terningerne. 3: systemet viser summen af terningerne 4: systemet rykker spillerens brik frem på spilpladen, lig summen af terningerne. *Spilleren lander på Tribe Encampment* 5: Systemet spørger om spilleren vil købe feltet 6a: systemet skifter tur, da spilleren vælger ikke at købe feltet. **Alternativt flow2** 1: Use Case startet når ChooseCarColor har kørt

2:Systemet beder spilleren om at slå med terningerne.

3:systemet viser summen af terningerne

4:systemet rykker spillerens brik frem på spilpladen, lig summen af terningerne.
Spilleren lander på Tribe Encampment

5a: Systemet flytter penge fra spillerens account til feltejerens account.

Alternativt flow3
1: Use Case startet når ChooseCarColor har kørt

2:Systemet beder spilleren om at slå med terningerne.

3: systemet viser summen af terningerne

4: systemet rykker spillerens brik frem på spilpladen, lig summen af terningerne.
Spilleren lander på Tribe Encampment

5.b: Systemet skifter tur, da spilleren allerede ejer feltet.

Alternativt flow4
1: Use Case startet når ChooseCarColor har kørt

2: Systemet beder spilleren om at slå med terningerne.

3: Systemet viser summen af terningerne

4a: Systemet rykker spillerens brik frem på spilpladen, lig summen af terningerne.
Spilleren lander på Walled City

5: Spilleren modtager 5000

Alternativt flow5
1: Use Case startet når ChooseCarColor har kørt

2: Systemet beder spilleren om at slå med terningerne.

3: Systemet viser summen af terningerne

4b: Systemet rykker spillerens brik frem på spilpladen, lig summen af terningerne.
Spilleren lander på The pit

5a. Systemet Spørger om spilleren vil købe feltet

6: Systemet sætter feltet til at være ejet af den pågældende spiller

Alternativt flow6
1: Use Case startet når ChooseCarColor har kørt

2: Systemet beder spilleren om at slå med terningerne.

3: Systemet viser summen af terningerne

4b: Systemet rykker spillerens brik frem på spilpladen, lig summen af terningerne.
Spilleren lander på The pit

5a: Systemet Spørger om spilleren vil købe feltet

6b: systemet skifter tur, da spilleren vælger ikke at købe feltet.

Alternativt flow7
1: Use Case startet når ChooseCarColor har kørt

2: Systemet beder spilleren om at slå med terningerne.

3: Systemet viser summen af terningerne

4b: Systemet rykker spillerens brik frem på spilpladen, lig summen af terningerne.
Spilleren lander på The pit

5b: Systemet flytter penge fra spillerens account til feltejerens account.

Alternativt flow7
1: Use Case startet når ChooseCarColor har kørt

2: Systemet beder spilleren om at slå med terningerne.

3: Systemet viser summen af terningerne

4b: Systemet rykker spillerens brik frem på spilpladen, lig summen af terningerne.
Spilleren lander på The pit

5c: Systemet skifter tur, da spilleren allerede ejer feltet.

Alternativt flow8
1: Use Case startet når ChooseCarColor har kørt

2: Systemet beder spilleren om at slå med terningerne.

3: Systemet viser summen af terningerne

4c: Systemet rykker spillerens brik frem på spilpladen, lig summen af terningerne.
Spilleren lander på Goldmine

5: Systemet spørger om spilleren vil betale 2000 eller 10% af sin samlede pengeværdi.

6a: Systemet trækker 2000 fra spillerens Pengeværdi.

Alternativt flow9
1: Use Case startet når ChooseCarColor har kørt

2: Systemet beder spilleren om at slå med terningerne.

3: Systemet viser summen af terningerne

4c: Systemet rykker spillerens brik frem på spilpladen, lig summen af terningerne.
Spilleren lander på Goldmine

5: Systemet spørger om spilleren vil betale 2000 eller 10% af sin samlede pengeværdi

6b: Systemet trækker 10% fra spillerens Pengeværdi

Alternativt flow10
1: Use Case startet når ChooseCarColor har kørt

2: Systemet beder spilleren om at slå med terningerne.

3: Systemet viser summen af terningerne

4d: Systemet rykker spillerens brik frem på spilpladen, lig summen af terningerne.
Spilleren lander på Sea Grover

5a. Systemet Spørger om spilleren vil købe feltet

6a: Systemet sætter feltet til at være ejet af den pågældende spiller

Alternativt flow11
1: Use Case startet når ChooseCarColor har kørt

2: Systemet beder spilleren om at slå med terningerne.

3: Systemet viser summen af terningerne

4d: Systemet rykker spillerens brik frem på spilpladen, lig summen af terningerne.
Spilleren lander på Sea Grover

5a: Systemet Spørger om spilleren vil købe feltet

6b: systemet skifter tur, da spilleren vælger ikke at købe feltet.

Alternativt flow12
1: Use Case startet når ChooseCarColor har kørt

2: Systemet beder spilleren om at slå med terningerne.

3: Systemet viser summen af terningerne

4d: Systemet rykker spillerens brik frem på spilpladen, lig summen af terningerne.
Spilleren lander på Sea Grover

5b: Systemet flytter penge fra spillerens account til feltejerens account.

Alternativt flow13
1: Use Case startet når ChooseCarColor har kørt

2: Systemet beder spilleren om at slå med terningerne.

3: Systemet viser summen af terningerne

4d: Systemet rykker spillerens brik frem på spilpladen, lig summen af terningerne.
Spilleren lander på Sea Grover

5c: Systemet skifter tur, da spilleren allerede ejer feltet.

Alternativt flow14
1: Use Case startet når ChooseCarColor har kørt

2: Systemet beder spilleren om at slå med terningerne.

3: Systemet viser summen af terningerne

4e: Systemet rykker spillerens brik frem på spilpladen, lig summen af terningerne.
Spilleren lander på et givet felt som ikke er af typen Refuge

5: Spilleren kan ikke betale og er ikke længere med i spillet

6a: Systemet skifter til næste spillers tur

Alternativt flow15
1: Use Case startet når ChooseCarColor har kørt

2: Systemet beder spilleren om at slå med terningerne.

3: Systemet viser summen af terningerne

4e: Systemet rykker spillerens brik frem på spilpladen, lig summen af terningerne.
Spilleren lander på et givet felt som ikke er af typen Refuge

5: Spilleren kan ikke betale og er ikke længere med i spillet

6b: Den tilbageværende spiller er den sidste og vinder.

7: Systemet spørger om man vil spille igen eller afslutte.

	8a: Spillet starter igen. **<u>Alternativt flow16</u>** 1: Use Case startet når ChooseCarColor har kørt 2: Systemet beder spilleren om at slå med terningerne. 3: Systemet viser summen af terningerne 4e: Systemet rykker spillerens brik frem på spilpladen, lig summen af terningerne. *Spilleren lander på et givet felt som ikke er af typen Refuge* 5: Spilleren kan ikke betale og er ikke længere med i spillet 6b: Den tilbageværende spiller er den sidste og vinder. 7: Systemet spørger om man vil spille igen eller afslutte. 8b: Systemet lukker ned.

Feltliste:

1.	Tribe Encampment	Territory	Rent 100	Price 1000	
2.	Crater	Territory	Rent 300	Price 1500	
3.	Mountain	Territory	Rent 500	Price 2000	
4.	Cold Desert	Territory	Rent 700	Price 3000	
5.	Black cave	Territory	Rent 1000	Price 4000	
6.	The Werewall	Territory	Rent 1300	Price 4300	
7.	Mountain village	Territory	Rent 1600	Price 4750	
8.	South Citadel	Territory	Rent 2000	Price 5000	
9.	Palace gates	Territory	Rent 2600	Price 5500	
10.	Tower	Territory	Rent 3200	Price 6000	
11.	Castle	Territory	Rent 4000	Price 8000	
12.	Walled city	Refuge	Receive 5000		
13.	Monastery	Refuge	Receive 500		
14.	Huts in the mountain		Labor camp	Pay 100 x dice	Price 2500
15.	The pit	Labor camp	Pay 100 x dice	Price 2500	
16.	Goldmine	Tax	Pay 2000		
17.	Caravan	Tax	Pay 4000 or 10% of total assets		
18.	Second Sail	Fleet	Pay 500-4000	Price 4000	
19.	Sea Grover	Fleet	Pay 500-4000	Price 4000	
20.	The Buccaneers	Fleet	Pay 500-4000	Price 4000	
21.	Privateer armade	Fleet	Pay 500-4000	Price 4000	

Typer af felter:

Territory

- Et territory kan købes og når man lander på et Territory som er ejet af en anden spiller skal man betale en afgift til ejeren.

Refuge

- Når man lander på et Refuge får man udbetalt en bonus.

Tax

- Her fratrækkes enten et fast beløb eller 10% af spillerens formue. Spilleren vælger selv mellem disse to muligheder.

Labor camp

- Her skal man også betale en afgift til ejeren. Beløbet bestemmes ved at slå med terningerne og gange resultatet med 100. Dette tal skal så ganges med antallet af Labor camps med den samme ejer.

Fleet

- Endnu et felt hvor der skal betales en afgift til ejeren. Denne gang bestemmes beløbet ud fra antallet af Fleets med den samme ejer, beløbene er fastsat således:
  1. Fleet: 500
  2. Fleet: 1000
  3. Fleet: 2000
  4. Fleet: 4000