Projektopgave efterår 2015 – jan 2016
02312-14 Indledende programmering, 02313 Udviklingsmetoder til IT-Systemer og 02315 Versionsstyring og testmetoder.

Projekt navn: *CDIOdel3*
Gruppe nr.: *38.*
Afleveringsfrist: *lørdag den 28/11 2015 Kl. 05:00*

Denne rapport er afleveret via Campusnet

Denne rapport indeholder 29 sider inkl. denne side.


Silas El-Azm Stryhn
s143599


Ramyar Hassani
s143242


Mikkel Hansen
s143603

Frank Thomsen
s124206

Martin Rødgaard
s143043

Timeregnskab

CDIO del 3 Timeregnskab					
	Silas	Martin	Mikkel	Ramyar	Frank
Analyse	0	2	0	0	3
Design	5	2	1	2	1
Implementering	5	5	20	5	5
Dokumentation	7	7	5	3	7
I alt	17	16	26	10	16

Indholdsfortegnelse

Resumé

Vores rapport indeholder et analyseafsnit, hvor vi har analyseret og opdelt alle kundens krav efter FURPS+ metoden og MoSCoW metoden. Vi har også udarbejdet nogle diagrammer til analyseringen samt beskrivelser af disse.
I vores design afsnit har vi lavet og beskrevet diverse diagrammer, som gør vores spil mere overskueligt og forståeligt. Det er også i dette afsnit vi kommer ind på blandt andet arv og GRASP.
I implementerings afsnittet har vi blandt andet beskrevet hvordan vores fleet felter virker og koden bag det. Til sidst i rapporten vil vi komme ind på de 3 test cases samt deres rapporter og vores 5 jUnit tests som vi har lavet af vores spil.

Indledning

Denne rapport præsenterer gennemgangen af den softwaremæssige udvikling af et digitalt brætspil for spilfirmaet IOOuterActive. Firmaet ønsker et spil hvor to til seks personer skiftevis kaster med to terninger og hvor der findes forskellige felter man kan lande på.
Hver spiller har sin egen balance som de kan bruge på at købe disse felter, hertil skal det dog siges, at enkelte felter ikke kan købes men disse har en anden positiv eller negativ effekt på spillerens balance. Hvis et felt er ejet af en anden spiller betales der i stedet et beløb til ejeren. Spillet fortsætter til alle spiller på nær en enkelt er gået bankerot.
For at kunne udarbejde sådan et spil på den mest effektive måde og for at undgå eventuelle problemer der kunne opstå under udviklingen, har vi været igennem de fire stadier, der er nødvendige for at kunne danne sig en struktureret og overskuelig arbejdsplan: Analyse, design, implementering og test.
Rapportens basale formål er at give en grundlæggende forståelse for tankegangen igennem forløbet, hvilke overvejelser der er gjort og hvordan spillet er stykket sammen. Rapporten gennemgår således en analyse af kundens vision for spillet og udarbejdelsen af diverse diagrammer, som er med til at overskueliggøre tilgangen til udviklingen.

Hovedafsnit

Analyse

Vi har analyseret og udarbejdet en kravspecifikation ud fra opgavebeskrivelsen.
Dertil har vi også udarbejdet nogle diagrammer, der løbende er blevet ændret under udviklingen, efterhånden som vi har skrevet mere og mere på koden. Det vil sige, at diagrammerne der fremgår i denne rapport er et produkt af det færdige spil. Vi har ifølge opgavebeskrivelse udarbejdet hhv. Domæne- og BCE-modeller samt UseCase- diagram og beskrivelser og til sidst System-sekvensdiagrammer.

Kravspecifikation

Kravene har vi taget stilling til på følgende to måder.
Først tog vi stilling til funktionaliteten, dette brugte vi FURPS+ metoden til. Det andet vi har tog stilling til, var vigtigheden af kravet. Til dette har vi brugt MoSCoW metoden.

System			
1.1 Spillet skal kunne spilles af 2-6 spillere.	Funktionelt	Must have	Vi har valgt at sætte dette krav til must have, da kunden nævner ordet skal og fordi det er en vigtig del af spillet.
1.2 Hver spiller starter med en balance på 30000.	Funktionelt	Should have	Dette krav har vi sat til should have, da kunden ønsker en start balance på 30000.
1.3 Spillet slutter når alle på nær én spiller er bankerot.	Funktionelt	Should have	Kunden skriver at spillet slutter når der kun er en spiller som ikke er bankerot, dette har vi valgt er en should have da kunden ikke nævner ordet skal.
1.4 Et spillebræt bestående af 21 felter. (se Feltliste og Felttyper under bilag)	Funktionelt	Should have	Vi har valgt at sætte dette krav til should have da kunden ønsker 21 felter.
1.5 Spillerne skal kunne lande på et felt og fortsætte derfra på næste	Funktionelt	Must have	Igen er dette et krav hvor kunden nævner skal og derudover er dette utrolig vigtigt for spillets flow, derfor har vi valgt must have. Især fordi der er tale om mere end 12 felter.

slag.			
1.6 Man går i ring på spillebrættet.	Funktionelt	Should have	Det at man går i ring i selve spillet er nødvendigt for at spillet vil kunne slutte med at alle er bankerot og selve ideen med spillet. Det kan dog godt fungere uden at gå i ring men som sagt ville ideen med spillet forsvinde og det ville ikke fungere efter hensigten derfor har vi valgt should have til dette krav.

Eventuelle spørgsmål til kravspecifikationen:

- *Hvilket felt starter man på?*

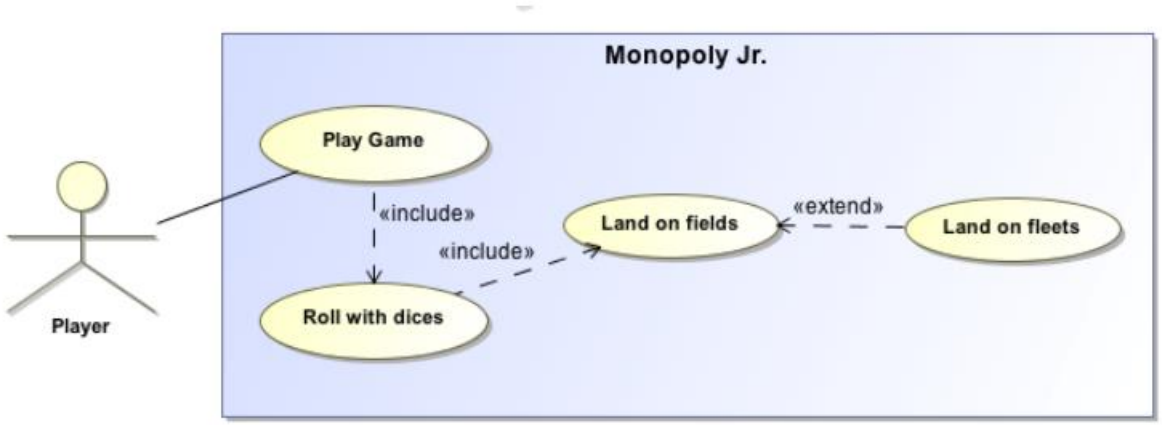
Vi har taget en executive beslutning om, at tilføje et startfelt hvor man starter.

- *Hvad sker der med felterne som en bankerot spiller ejede?*

Vi tog igen en executive beslutning om, at felterne en bankerot spiller ejede mister sin effekt.

Use Case Diagram

Vores Use Case diagram er et diagram, som bliver brugt til at kunne vise kunden, hvad idéerne til vores spil er. Hver Use Case er blevet beskrevet i vores Use Case beskrivelser.
Det der sker i diagrammet, er at vi har en Player, som spiller spillet, ved at slå med terningerne, og derefter rykke over på et felt. Vores land on fleets, er extended fra land on fields, da det er en af de mulige felter, som man kan lande på.



Use Case Beskrivelser

Use Case Name:	Play Game
Use Case ID:	1
Actors:	Player
Description:	Player starter spillet op.
Preconditions:	Spillet er i gang.
Postconditions:	Der bliver skiftet spiller undervejs.
Main flow:	1. Start spillet. 2. Vælg hvor mange spillere der skal være med. 3. Find startende spiller. 4. Den aktive spiller ruller med terningerne.
Alternative Flow:	1. Spilleren lukker spillet ned.

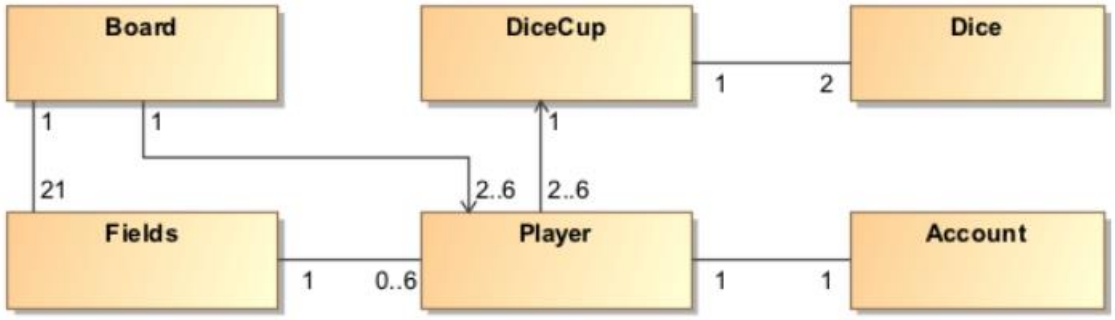
Use Case Name:	Roll with dices
Use Case ID:	2
Actors:	Player
Description:	Player slår med terningerne.
Preconditions:	Spillet er igang.
Postconditions:	Der bliver skiftet spiller undervejs.
Main flow:	1. Kast med terningerne. 2. Ryk det antal øjne, som terningerne viser. 3. Land på et felt.
Alternative Flow:	1. Spilleren lukker spillet ned.

Use Case Name:	Land on fields
Use Case ID:	3
Actors:	Player
Description:	Player lander på et felt.
Preconditions:	Spillet er igang.
Postconditions:	Felter kan blive købt og renter skal kunne betales.
Main flow:	1. Man kan lande på et felt. 1.1. Hvis feltet er et Territory felt: 1.1.1. Hvis feltet ikke er ejet: 1.1.1.1. Feltet kan købes. 1.1.2. Hvis feltet er ejet: 1.1.2.1. Der skal betales renter. 1.2. Hvis feltet er et Refuge felt: 1.2.1. Du modtager det pågældende felts bonus. 1.3. Hvis feltet er et LaborCamp felt: 1.3.1. Hvis feltet ikke er ejet: 1.3.1.1. Feltet kan købes. 1.3.2. Hvis feltet er ejet: 1.3.2.1. Betal summen på terningerne * 100 som rente. 1.4. Hvis feltet er et Tax felt: 1.4.1. Betal tax fra det pægældende felt, eller betal 10% af din balance. 1.5. Hvis feltet er et Fleet felt: 2. 1.5.1. Se Land on fleets. Use Case ID 4.
Alternative Flow:	1. Spilleren lukker spillet ned.

Use Case Name:	Land on fleets
Use Case ID:	4
Actors:	Player
Description:	Player lander på et fleet-felt.
Preconditions:	Spillet er igang.
Postconditions:	Felter kan blive købt og renter skal kunne betales.
Main flow:	1. 1.Man kan lande på et Fleet felt. 1.1. Hvis Fleet feltet ikke er ejet: 1.1.1. Mulighed for at købe feltet. 1.2. Hvis Fleet feltet er ejet: 1.2.1. Hvis ejeren ejer 1 Fleet felt: 1.2.1.1. Betal 500. 1.2.2. Hvis ejeren ejer 2 Fleet felter: 1.2.2.1. Betal 1000. 1.2.3. Hvis ejeren ejer 3 Fleet felter: 1.2.3.1. Betal 2000. 1.2.4. Hvis ejeren ejer 4 Fleet felter: 1.2.4.1. Betal 4000. 1.3. Hvis Fleet feltet er ejet af dig selv: 1.3.1. Se feltbeskrivelse. 1.3.2. Der sker ingenting. 1.4. Hvis feltet er ejet af en bankrupt spiller: 1.4.1. Se feltbeskrivelse. 2. 1.4.2. Der sker ingenting.
Alternative Flow:	1. Spilleren lukker spillet ned.

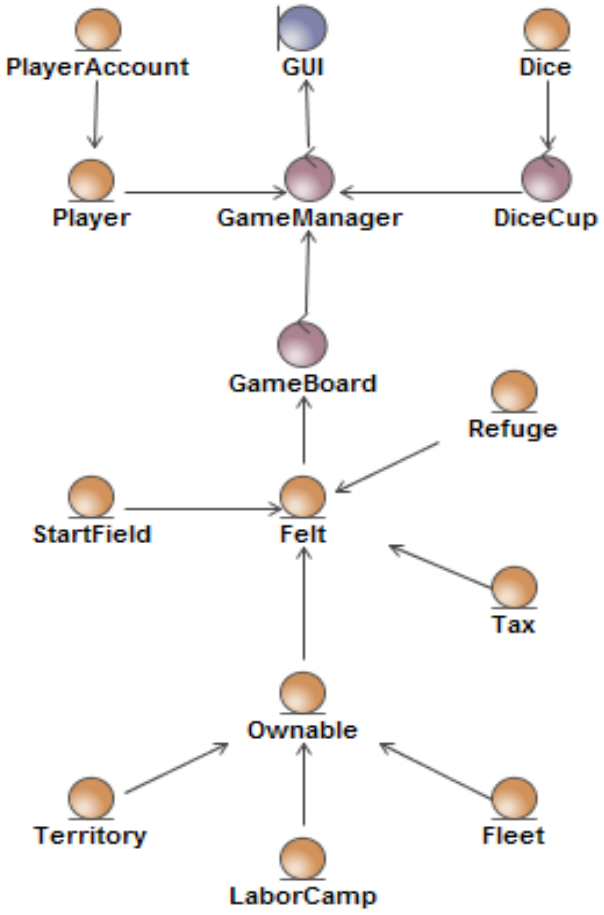
Domæne model

Vores domæne model er en model af virkeligheden, som vi kan vise vores kunde, så de får en ide om, hvilke elementer, som vi har valgt at have med i vores programmering. Dette er kun byggesten, og klasserne kan have ændret navne, og der kan være tilføjet flere i det endelige program.



BCE model

Vores BCE model, viser hvilket ansvar vores forskellige klasser har, en Boundary, styrer udseendet, vores Controller styrer spillet, og vores Entitys indeholder relevante metoder og informationer, som bliver brugt i vores Controllere.

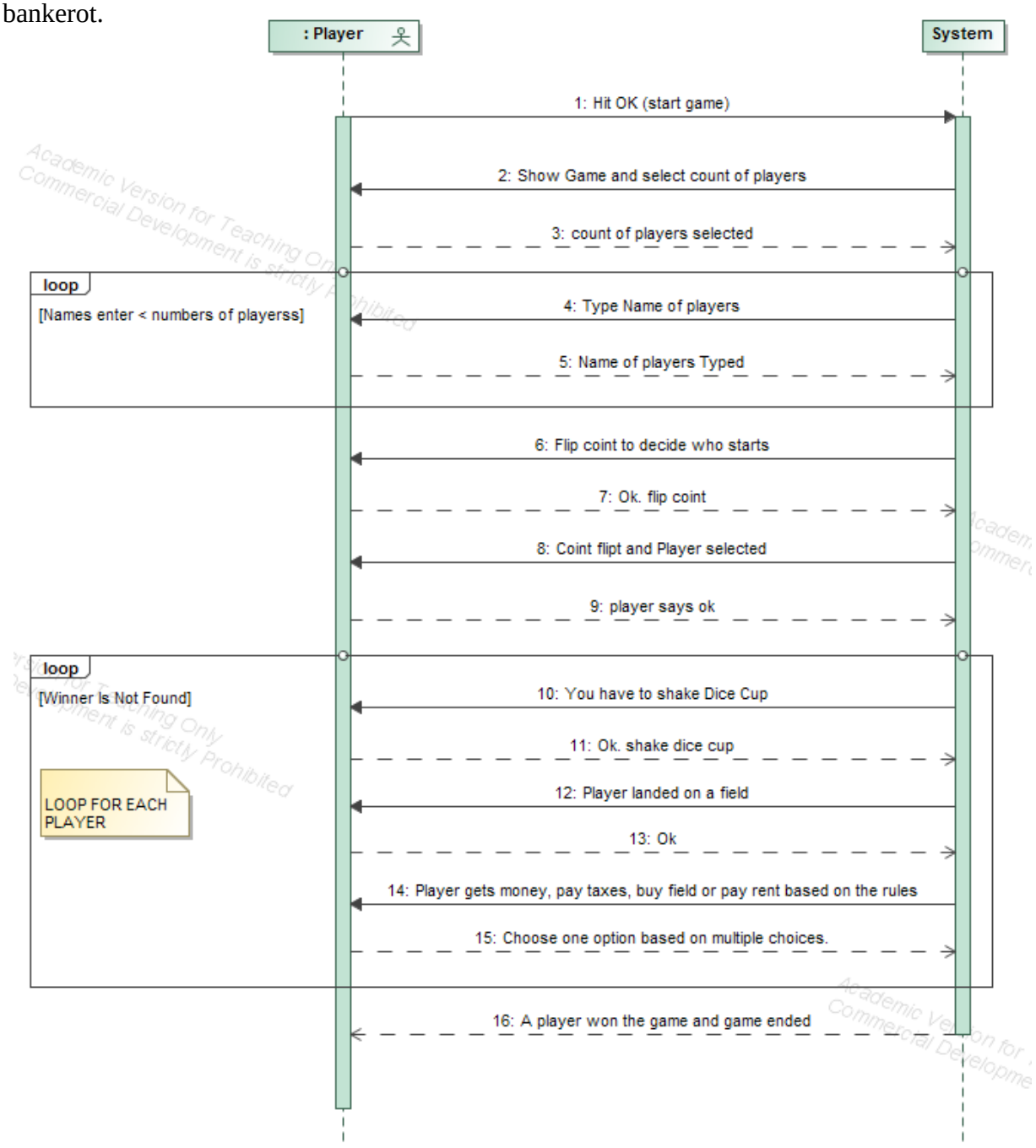


System-sekvensdiagram

Dette diagram viser hvordan brugeren og systemet kommunikere igennem selve spillet.

Brugeren starter spillet ved trykke OK. Systemet spørg brugeren om angive hvor mange spiller der skal spille spillet (2-6 spiller kan vælges). Brugeren skal efterfølgende skrive navnene på det antal spillere som skal spille og det foregår i et loop.

Så kaster brugeren en mønt for at vælge hvem der starter. Systemet svarer så at mønten er kastet og at en tilfældig spiller er valgt og spilleren trykker ok for starte. Så ruller brugeren med terningerne og dertil svarer systemet ved at vise terninger på skærmen. Så lander brugeren på et felt og systemet fortæller at du er landet på (navn på feltet) og brugeren skal trykke OK. Efterfølgende får brugerne en mulighed for købe, betale skat, få et beløb eller leje feltet (alle felter har en af mulighederne). Beløbet bliver trykket eller sat til din bankbog. Spillet slutter så når alle brugere undtagen én er bankerot.



Design

Det kan være en rigtig god idé at udarbejde nogle diagrammer i forbindelse med udviklingen af et system. Diagrammer er med til at overskueliggøre hvordan systemet fungerer i forskellige situationer og det kan være en god måde at visualiserer opbygningen af systemet på overfor kunden. På baggrund af dette har vi udarbejdet nogle diagrammer som giver et overblik over hvordan vores system er sat sammen samt hvilke funktioner der indgår.
I designfasen har vi udarbejdet et design-klassediagram og et design-sekvensdiagram, som løbende er blevet ændret i takt med udviklingen af spillet.

Arv

Man kan definere underklasser, der arver fra en anden klasse og kun lave den ekstra kode, der skal til for at definere underklassen i forhold til stamklassen, som også kan kaldes for superklassen. At arve fra en anden klasse, betyder at man genbruger en del af koden fra den pågældende klasse.
Arv er en vigtig del af at programmere, da det kan spare en for en del kode, da man kan have en del klasser, som ligner hinanden, men afviger med små forskelle fra hinanden. Et eksempel her kunne være vores forskellige felter fra vores spil. Her skal man kunne lande på forskellige felttyper, og alt efter hvilken felttype man lander på, skal feltet gøre noget forskelligt med spilleren pengebeholdning. Dog er felternes klasse meget ens, da de alle nedarver fra en overordnet abstract felt klasse.

Abstract

At en klasse er abstrakt betyder, at klassen ikke kan blive brugt, men at andre klasser kan nedarve fra den, og overskrive dennes metoder, og bruge dem i, som deres egne.
Metoderne i en abstrakt klasse, skal også være abstrakte, da en metode i en sådan klasse ikke kan have en body.
Så overordnet set, laver man en abstrakt klasse, for at kunne nedarve fra denne i underklasser, så man sparer tid og får en mere overskuelig kode.

Metoden *landOnField()*

Alle de forskellige felttyper har en forskellig *landOnField()* metode, fordi de hver især, gør noget forskelligt ved spillernes pengebeholdning. Det at den samme metode, som er i alle klasserne, fordi de alle nedarver fra den samme superklasse, *Felt*. Det der er forskellen på metoden i de forskellige klasser, er at den er blevet overskrevet og udvidet.

GRASP

Information Expert

Vi har overholdt Information Expert i form af, at vi bruger den klasse med mest af det relevante data til at udføre handlinger. For eksempel bruger vi vores DiceCup til at håndtere alle vores terningekast. Man kan altså sige, at dataen er indkapslet i DiceCup.

Low Coupling

Vi har overholdt det her pattern i nogle tilfælde, men ikke alle. Mellem GameBoard og Felt klasserne har vi en dobbelt kobling, ellers kunne vi ikke ændre i subTexten på et felt når det blev købt. Vi kunne godt have kørt vores program uden denne funktion, men vi syntes hurtigt, at det blev uoverskueligt hvem der ejede hvad hvis vi fjernede funktionen og derfor har vi valgt at gå lidt på kompromis med GRASP patterns.

High Cohesion

Vi har overholdt dette pattern ved, at bryde vores kode ned i flere forskellige klasser, der sørger for hver deres del af programmet. Normalt bruges det til, at understøtte “Low Coupling”, hvilket det også gør i vores program. Dog med den undtagelse, at vi et enkelt sted har en dobbelt kobling.

Creator

Vi har overholdt dette pattern ved, at lade de forskellige konstruktører oprette de objekter de skal bruge fra andre klasser. Nedenunder er vist et eksempel fra vores GameManager.

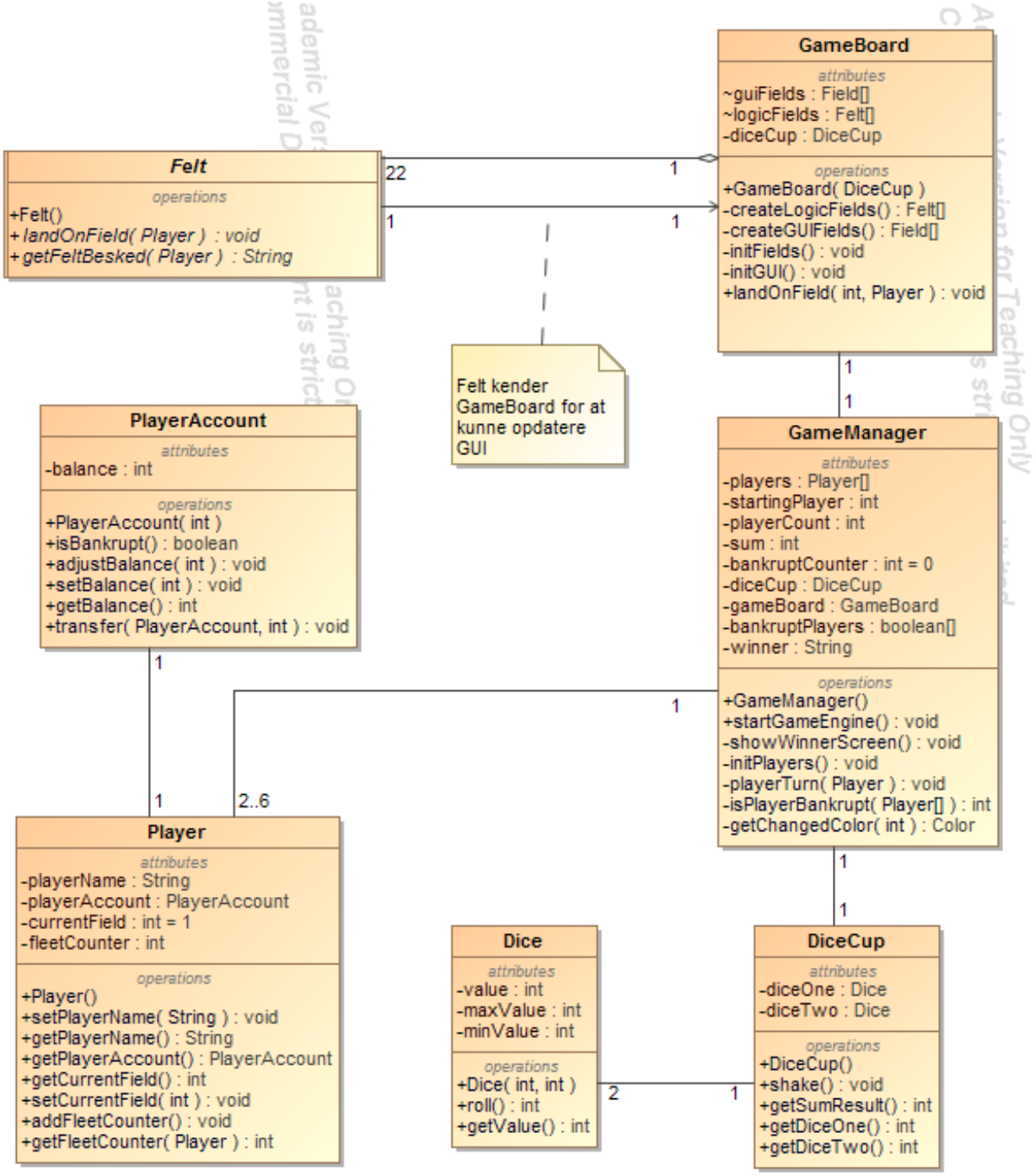
```
//GameManager constructor
public GameManager()
{
    this.diceCup = new DiceCup();
    this.gameBoard = new GameBoard(diceCup);
}
```

Controller

Vi mener, at vi har overholdt Controller pattern. Vi har en GameManager der styrer selve spillet, og den uddeligerer alle opgaverne til andre klasser, ligesom den bør. Vi har en DiceCup controller, så uddeligerer selve terningekastet til en Dice klasse og vi har en GameBoard klasse der sørger for at styre selve brættet. Man kan dog godt argumentere for, at GameBoard ikke opfylder dette pattern, da den selv står for at udføre alle de metoder der bliver kaldt, hvor imod, de to andre får andre klasser til det.

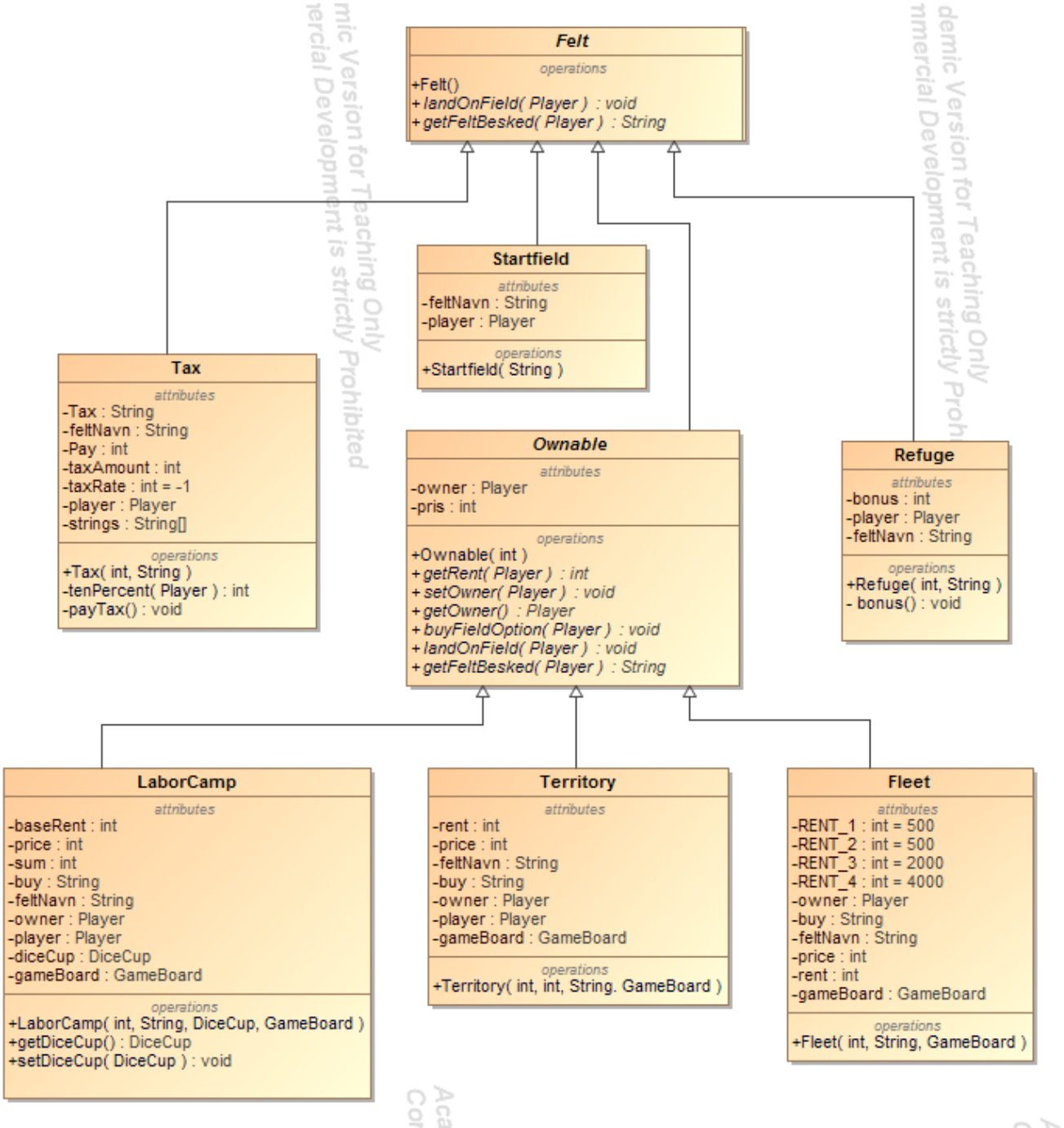
Design-klassediagram

Dette diagram viser hvordan klasserne i vores program hænger sammen. I klasserne har vi medtaget attributter, metoder, hvad metoderne tager imod og hvilken type både metoder og attributter er samt multipliciteter klasserne imellem.



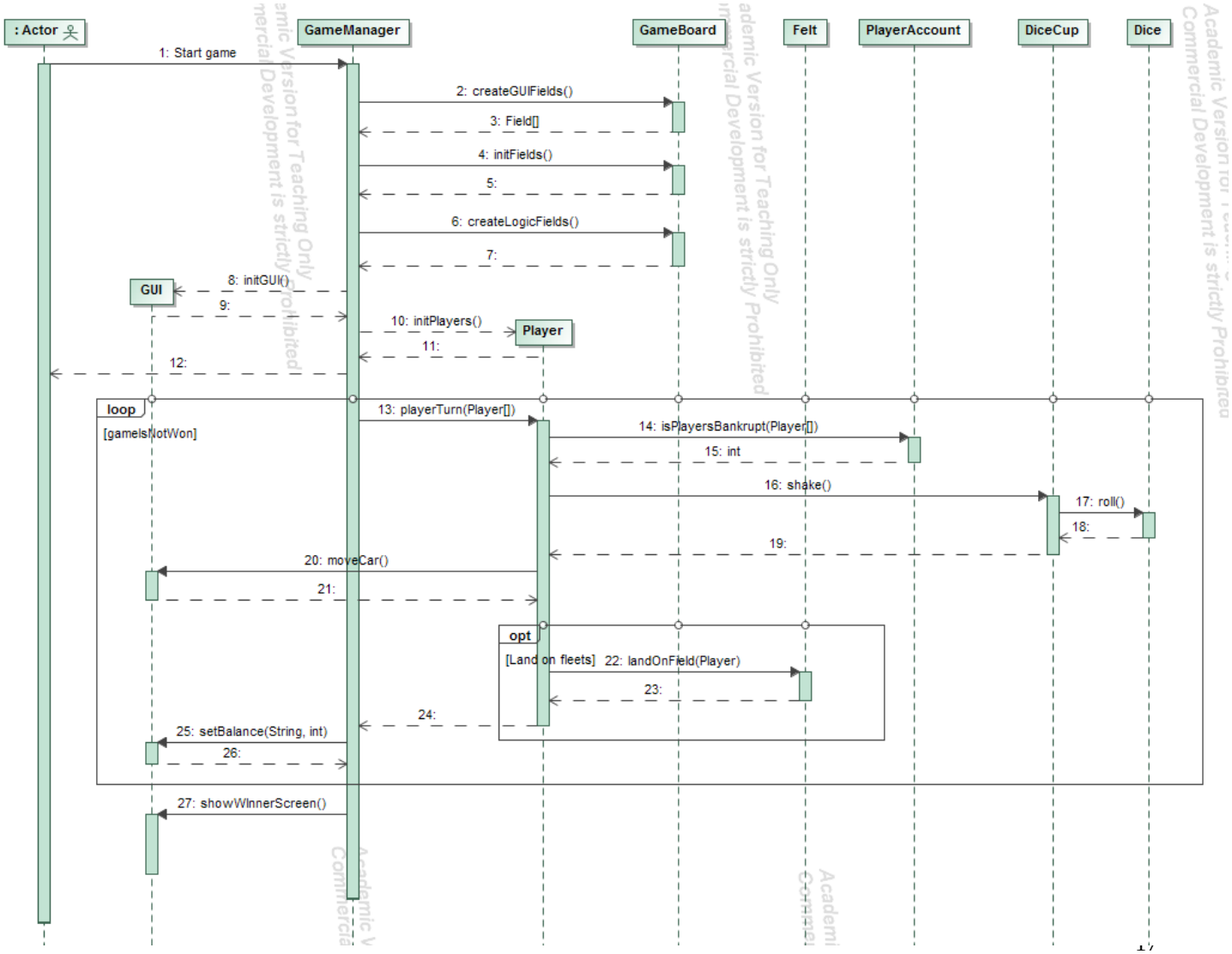
Design-klassediagram med nedarvning fra klassen Felt

Dette diagram er en udvidet version af vores Felt klasse, som viser de klasser der nedarver fra Felt. På disse er der ligeledes medtaget attributter, metoder, hvad metoderne tager imod og hvilken type både metoder og attributter er.



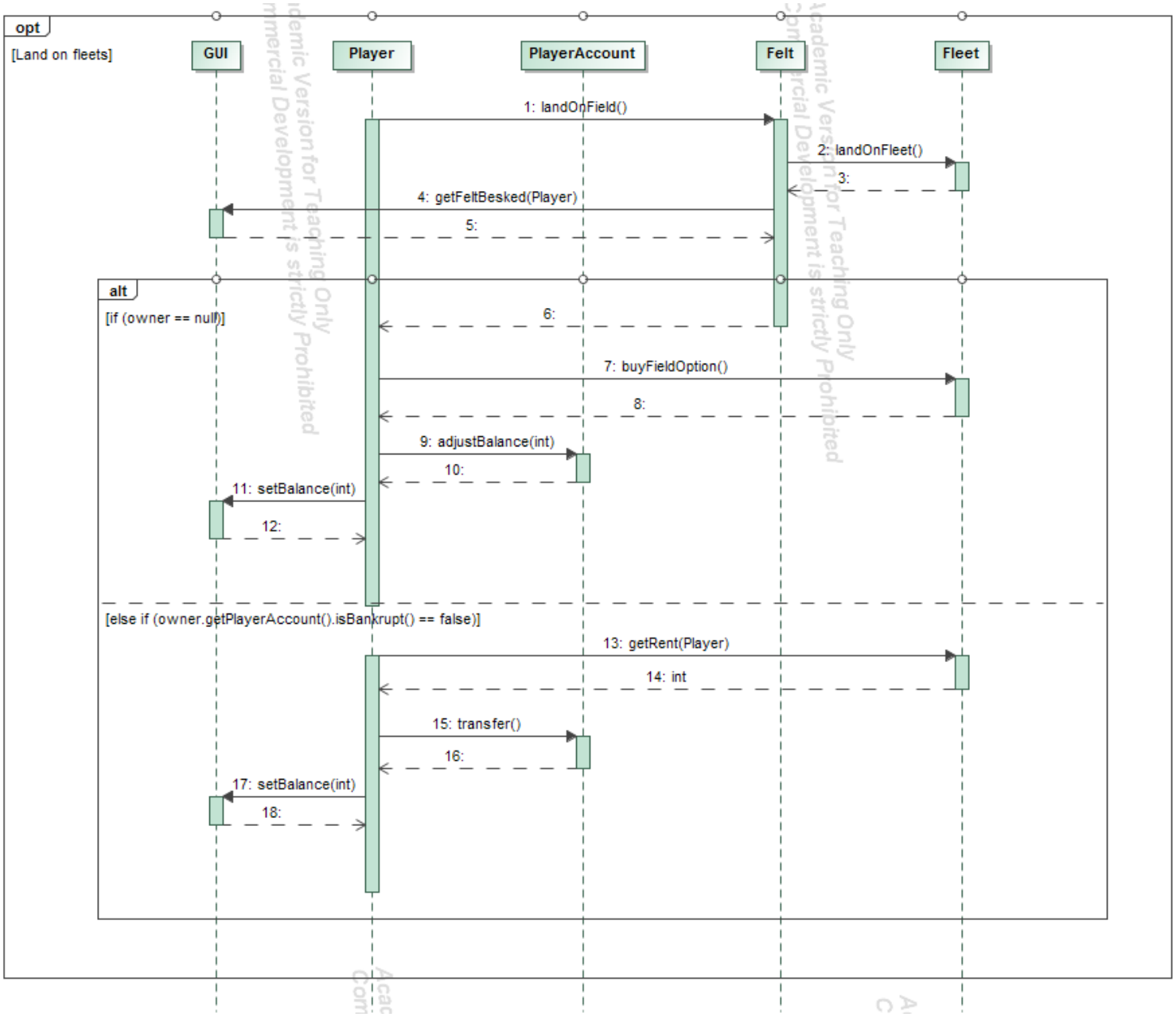
Design-sekvensdiagram

På dette diagram ses det, at spillet starter ved et input fra en spiller, hvorefter at GameManageren først sørger for at sætte selve GUI'en op og bagefter sætte spillerne op ud fra det input spilleren giver. Så kommer vi til selve spillet i form af playerTurn(Player[]) metode. Den tager imod et Player array og sørger for, at køre alle spillernes ture igennem. Dog springer den en spiller over såfremt at denne er bankrupt. Så ruller spilleren med terninger og bevæger sig frem alt efter hvor meget han slog. Når han er færdig med at flytte brikken, og vi ved hvilket felt han lander på, aktiveres landOnField(Player) metoden der ser på hvilken effekt det skal have. Så bliver GUI'en opdateret. Dette kører i loop så længe der ikke er fundet en vinder. En vinder bliver fundet ved, at alle spillere undtagen én er bankrupt.
Når vinderen er fundet, bliver showWinnerScreen() metoden kaldt, der blot viser hvem der har vundet på GUI'en.



Design-sekvensdiagram fragment af *landOnField()*

På diagrammet ses flowet i programmet når man lander på et Fleet felt. Feltet skriver først en besked på GUI'en. Herefter er der en if else statement der først ser på om feltet er ejet. Hvis feltet ikke ejet allerede, får spilleren mulighed for selv at købe feltet, hvilket resulterer i, at der bliver trukket penge fra hans konto og herefter bliver GUI'en opdateret.
Hvis feltet allerede er ejet og ejeren ikke er den samme spiller som landede på feltet samt at ejeren ikke er bankrupt vil feltets getRent(Player) metode blive kørt, som ser på hvor mange andre Fleet felter ejeren har og herefter returnere det korrekte beløb der skal betales. Slutteligt overføres pengene fra den ene spiller til den anden og GUI'en bliver opdateret.



Implementering

Se bilag for vejledning til installation og systemkrav.

Til vores implementering har vi brugt Github, hvor vi har lavet et repository, så alle gruppemedlemmer har kunne arbejde på det samme projekt. Dette gør det lettere at kunne gå tilbage i kodningen, hvis der opstår en fejl, og det gør det lettere at holde styr på, hvad vi hver især har lavet på projektet. Vi har i vores kode prøvet at sætte det op, på den mest overskuelige måde, og har herefter kommenteret stort set alt i de forskellige klasser, så det er nemmere at forstå vores kode.

GUI

Der er blevet gjort nogle enkelte rettelser i GUI'en, som vi mente ville gøre brugeroplevelsen af vores brætspil endnu bedre! Det har også givet spillet et mere flydende flow, i form af at det nu er muligt at benytte Enter-knappen til at trykke på knapperne i spillet, således at man ikke længere er nødsaget til at bruge musen til at trykke hver gang man skal videre i spillet. Dertil er der også også lavet en rettelse der gør, at spillet nu åbner op i midten af skærmen i stedet for i venstre øverste hjørne samt at titel-teksten i felterne er blevet ændret til en størrelse, som felterne kunne indeholde.

Fleet

Nedenfor har vi beskrevet hvordan vi har løst opgaven med fleet felterne. Vi valgte at gøre det ved at tilføje en attribut på hver player, hvor vi kunne tælle antallet af ejede fleets, da dette virkede som en simpel måde at gøre det på.

I vores Player klasse har vi lavet en attribut *private int fleetCounter*.
Denne attribut bliver i vores konstruktør *public Player()* sat til 0. Det vil sige når en spiller bliver oprettet har vi et sted vi kan tælle hvor mange fleets personen ejer.
I bunden af vores Player klasse har vi så lavet en *public void addFleetCounter()*, det som denne metode gør, er at lægge 1 til *fleetCounter* hver gang et fleetfelt købes.
Hvis man så kigger i vores Fleet klasse finder man en metode *public void buyFieldOption(Player player).*
I denne kan man se at *if(buy.equals("Yes"))* så bliver *player.addFleetCounter()* kaldt. Dette betyder at hvis en player svarer "Yes" til at købe en fleet bliver metoden *addFleetCounter()* kaldt på denne player og han får derfor lagt 1 til hvor mange fleets han ejer.

Vi har løst tilfældet at en player lander på en anden players fleet på følgende måde.
Vi har lavet en metode for dette, som vi har kaldt *public void landOnField(Player player).*
Metoden tjekker først om den er ejet af en anden, hvis ikke, bliver man sendt til *buyFieldOption(player)* som jeg nævnte før.
Hvis personen som ejer denne er bankerot sker der intet. Men hvis denne er ejet af en anden player så starter den med at hente *getRent(owner).*
Det vi har gjort for at få den til at hente det rigtige er.
I toppen af vores Fleet klasse har vi oprettet 4 antributter, *private final int RENT_1, RENT_2, RENT_3 og RENT_4*. Disse beskriver beløbet af de 4 mulige tilfælde.
Derefter har vi lavet metoden *public int getRent(Player player)* i denne ligger en switch case som via get metoden *getFleetCounter(owner)* får returneret hvor mange fleets ejeren har.
I switch casens cases blev de 4 attributter sat til *rent* og afhængig af den int som blev returneret fra *getFleetCounter(owner)* vælges den case, altså det beløb, som passer til det antal ejede fleets.
Til sidst i *public int getRent(Player player)* bliver *rent* så returneret til *public void landOnField(Player player)* hvor den så overfører beløbet fra player til owner vha. *player.getPlayerAccount().transfer(owner.getPlayerAccount(), rent).*

Test

Test Cases

Test case 1: Kan balance blive negativ (Negativ)

Preconditions:
1. Opret ny Player.
2. Sæt balance til 1000.
3. Opret Territory felt.
4. Sæt pris 2000.

Test:
1. Køb feltet.

Postconditions:
1. Kontrollér balancen er 0.

Test case 2: Overførsel af penge hvis man lander på et allerede ejet felt (positiv)

Preconditions:
1. Opret to players.
2. Sæt deres kontoer til 30000.
3. Opret Territory felt.
4. Sæt rent: 200.
5. Owner bliver sat som ejer af Territory feltet.

Test
1. Player betaler rent til Owner.

Postconditions:
1. Konrollér Owners balance er 30200.

Kontrollér Players balance er 29800.

Test case 3: Køb et felt (positiv)

Preconditions:
1. Opret en player.
2. Sæt konto til 30000.
3. Opret Fleet felt.
4. Sæt pris til 4000.
5. Land på et Fleet felt.

Test:
1. Køb feltet.

Postconditions:
1. Kontrollér Players balance er 26000.

Test Rapportering

Test Rapport 1	
	Dato 27/11-2015 Commit ID: df79324
Pre 1: Opret Player	OK
Pre 2: Sæt balance til 1000	OK
Pre 3: Opret Territory felt	OK
Pre 4: Sæt pris til 2000	OK
Test 1: Køb feltet	OK
Post 1: Kontrollér Balancen er 0	OK

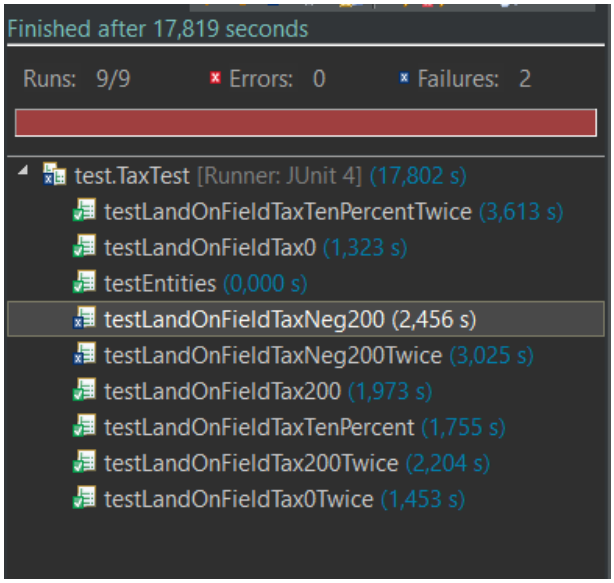
Test Rapport 2	
	Dato 27/11-2015 Commit ID: 7bb6aa2
Pre 1: Opret to Players (Owner, Player)	OK
Pre 2: Sæt balancer til 30000	OK
Pre 3: Opret Territory felt	OK
Pre 4: Sæt rent til 200	OK
Pre 5: Sæt Owner som ejer af Territory	OK
Test 1: Player betaler rent til Owner.	OK
Post 1: Kontrollér Owners balance er 30200	OK
Post 2: Kontrollér Players balance er 29800	OK

Test Rapport 3	
	Dato 27/11-2015 Commit ID: 7bb6aa2
Pre 1: Opret Player	OK
Pre 2: Sæt balance til 30000	OK
Pre 3: Opret Fleet felt	OK
Pre 4: Sæt pris til 4000	OK
Pre 5: Land på et fleet felt	OK
Test 1: Køb feltet	OK
Post 1: Kontrollér Players balance er 26000	OK

JUnitTest

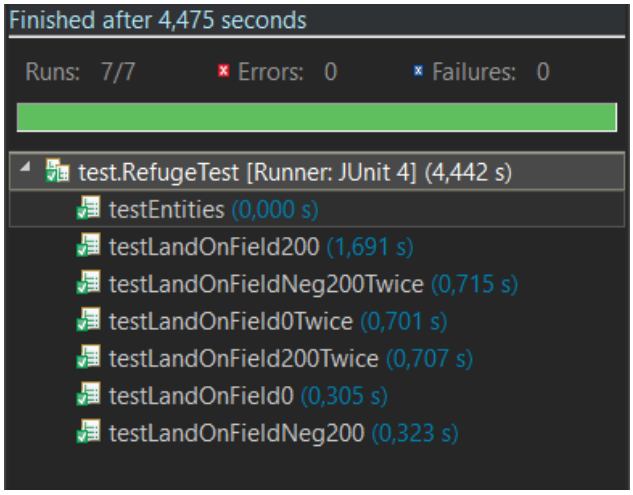
Fælles for alle vores testcases er, at hastigheden på selve testen er utroligt langsom da den skal starte GUI'en op og herefter vente på user inputs. Hvis det ikke var en del af processen ville de ikke tage så mange sekunder, men være overstået på ca. 20-30ms.

TaxTest



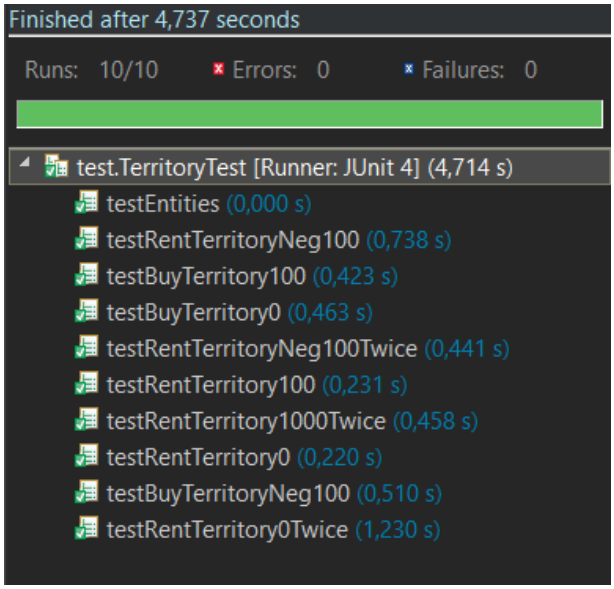
Her ses det, at alle tests i vores TaxTest lykkedes, undtagen dem hvor den gerne vil ligge penge til spillerens pengebeholdning. Det er ikke muligt på grund af den metode vi har brugt til at konvertere vores Strings vi får fra knapperne til ints. Den fjerner alle ikke-digits, og derfor bliver minusset også fjernet, så programmet ser det som om, at der blot står 200 i tax argumentet. Resten af testene går dog fint igennem såfremt man lige følger de kommentarer der er skrevet i test klassen.

RefugeTest



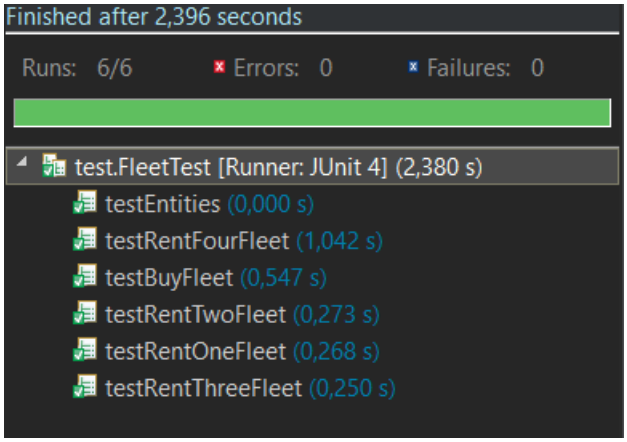
Her kan man se, at alle de tests der var opgivet i projektbeskrivelsen går rent igennem uden nogen problemer.

TerritoryTest



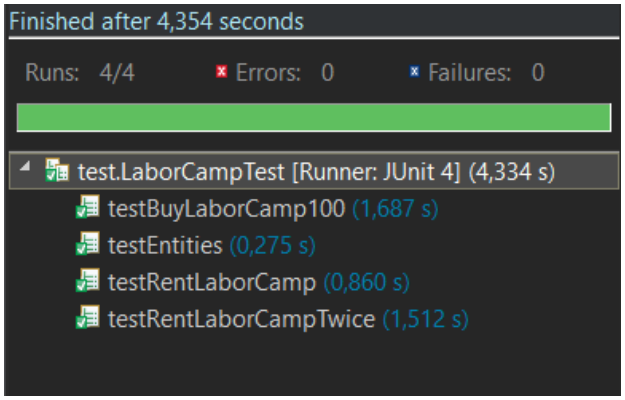
Her fremgår det, at vi har lavet ti forskellige tests med Territory. Den første går bare ud på, om alle objekterne er oprettet korrekt. Herefter lavede vi test på, at man kan købe hver af de tre Territory objekter og til sidst har vi seks tests der bare tester at den regner Rent rigtigt.

FleetTest



Fleet testen går ud på, at den opretter felterne korrekt, at man kan købe et fleet felt samt at den henter de rigtige priser når en spiller lander på et fleet felt der er ejet af en anden spiller, alt efter hvor mange fleet felter den anden spiller ejer. Alle tests gik rent igennem uden nogen problemer.

LaborCampTest



I testen af LaborCamp tester vi, at alt er oprettet korrekt, at det er muligt at købe en LaborCamp samt at betalingen når man lander på en anden spillers LaborCamp bliver beregnet korrekt (terningekast * 100). Her er det værd at nævne, at vi har valgt, at man ruller med to terninger.

Konklusion

Vi har fået lavet vores spil, som spilles af to til seks spillere, der bruger et raflebæger til at slå med terningerne. Hvorefter spillerne rykker rundt på brættet med de mange forskellige felter. Spillerne kan købe langt de fleste felter og på den måde tjene penge fra de andre spillere. Spillet fortsætter indtil alle på nær en enkelt spiller er gået bankerot.
I rapporten har vi lavet diagrammer for at få en forståelse af, hvordan vi ville lave vores program og hvordan vi ville formidle vores idéer til programmet videre til andre. I opgaven har vi brugt forskellige metoder, fordelt i vores klasser, som i sidste ende alle bliver kaldt i vores main klasse (Program). Opgavens indhold er blevet lavet ud fra kravspecifikationerne , og kan nu bruges som det skal. Alle regler er blevet implementeret samt nogle fede features som at man kan se hvem som ejer de forskellige felter og et coin toss i starten af spillet der bestemmer hvem som starter, i stedet for det altid er spiller 1.
Spillet er til sidst blevet testet med diverse tests, som blandt andet undersøger om en spiller kan købe et felt, om de forskellige felter virker og om penge korrekt kan bliver overført fra en spillers balance til en anden.

Bilag

Git

For at få fat i vores projekt, kan man importere det direkte til Eclipse vha. linket https://github.com/Chaaaaap/CDIO3.git. Dette gøres ved at trykke File → Import → Git → Projects som Git → Clone URI → Indtaster linket i URI-feltet → Next → Finish. Når dette er gjort henter Eclipse selv hele projektet ned. Når først man har hentet projektet ned til Eclipse, skal man blot åbne klassen "Program" og køre denne, for at starte spillet.
Man har dog også mulighed for at hente projektet ned som en .ZIP fil, der kan importeres i Eclipse som et eksisterende projekt. Dette gøres ved at trykke File → Import → General → Existing projects into workspace.

Systemkrav

Vores spil kan køres på alle maskiner der som minimum kører med Windows XP. Det eneste systemkrav der er, er at man skal have installeret JAVA 1.8, for at kunne bruge det library som vi har anvendt (jre 1.8.0_60). Man kan dog godt bruge en nyere udgave, så skal man blot ind og ændre i, hvilke libraries der er tilknyttet projektet. Det er ikke noget større problem, men hvis man ikke føler sig sikker på hvordan man tilføjer / fjerner libraries vil vi ikke anbefale det.

Feltliste

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

Type af felter

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

Litteraturliste

Kommentar fra brugeren polygenelubriants, http://stackoverflow.com/questions/2338790/get-int-from-string-also-containing-letters-in-java – 27/11-2015