CDIO3

Projektnavn: CDIO3
Gruppenr: 37
Rapporten indeholder 55 sider inkl. denne side.
DTU Compute, fag 02312, 02313, 02315

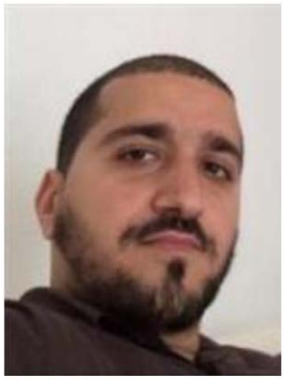
Deltagerliste:

Ehlers, Annika Bhagya
s153667

Hasan, Abdiwahabs
s144819

Mohamud, Abdi Shukri Aden
s136615

Sultanov, Aslanbek Usamovitj
s132993



Tas, Haydar
s136504

Yousef, Mahmud Mohammad
s136208

Dato	Deltager	Design	Impl.	Andet	I alt	Bemærkninger
27-11-2015	Abdi	0	8	0	8	Laver test
27-11-2015	Annika	2	4	2	8	Laver test
27-11-2015	Aslanbek	0	8	0	8	Laver test
27-11-2015	Haydar	0	8	0	8	Laver test
27-11-2015	Mahmmoud	0	8	0	8	Laver test
27-11-2015	Abdiwahab	0	8	0	8	Laver test
26-11-2015	Abdi	2	2	0	4	
26-11-2015	Annika	1	2	1	4	
26-11-2015	Aslanbek	2	2	0	4	
26-11-2015	Haydar	2	2	0	4	
26-11-2015	Mahmmoud	2	2	0	4	
26-11-2015	Abdiwahab	2	2	0	4	
25-11-2015	Abdi	2	0	0	2	
25-11-2015	Annika	0	0	2	2	
25-11-2015	Aslanbek	0	0	2	2	
25-11-2015	Haydar	0	2	0	2	
25-11-2015	Mahmmoud	0	2	0	2	
25-11-2015	Abdiwahab	1	1	0	2	
24-11-2015	Abdi	1	0	1	2	
24-11-2015	Annika	1	0	1	2	
24-11-2015	Aslanbek	1	0	1	2	
24-11-2015	Haydar	1	0	1	2	
24-11-2015	Mahmmoud	1	0	1	2	
24-11-2015	Abdiwahab	1	0	1	2	
23-11-2015	Abdi	0	1	1	2	
23-11-2015	Annika	0	1	1	2	
23-11-2015	Aslanbek	0	1	1	2	
23-11-2015	Haydar	0	1	1	2	
23-11-2015	Mahmmoud	0	1	1	2	
23-11-2015	Abdiwahab	0	1	1	2	
20-11-2015	Abdi	1	0	0	1	
20-11-2015	Annika	1	0	0	1	
20-11-2015	Aslanbek	1	0	0	1	
20-11-2015	Haydar	0	1	0	1	
20-11-2015	Mahmmoud	0	1	0	1	
20-11-2015	Abdiwahab	0	1	0	1	
19-11-2015	Abdi	1	0	0	1	
19-11-2015	Annika	1	0	0	1	
19-11-2015	Aslanbek	0	1	0	1	
19-11-2015	Haydar	1	0	0	1	
19-11-2015	Mahmmoud	0	1	0	1	
19-11-2015	Abdiwahab	0	1	0	1	
18-11-2015	Abdi	1	0	0	1	
18-11-2015	Annika	1	0	0	1	
18-11-2015	Aslanbek	1	0	0	1	
18-11-2015	Haydar	1	0	0	1	
18-11-2015	Mahmmoud	1	0	0	1	
18-11-2015	Abdiwahab	1	0	0	1	

17-11-2015	Abdi	0	0	1	1	
17-11-2015	Annika	0	0	1	1	
17-11-2015	Aslanbek	1	0	0	1	
17-11-2015	Haydar	1	0	0	1	
17-11-2015	Mahmmoud	0	0	1	1	
17-11-2015	Abdiwahab	0	0	1	1	
16-11-2015	Abdi	0	0	1	1	
16-11-2015	Annika	0	0	1	1	
16-11-2015	Aslanbek	0	1	0	1	
16-11-2015	Haydar	0	0	1	1	
16-11-2015	Mahmmoud	0	0	1	1	
16-11-2015	Abdiwahab	0	0	1	1	
13-11-2015	Abdi	0	0	1	1	
13-11-2015	Annika	0	0	1	1	
13-11-2015	Aslanbek	0	0	1	1	
13-11-2015	Haydar	0	0	1	1	
13-11-2015	Mahmmoud	0	0	1	1	
13-11-2015	Abdiwahab	0	0	1	1	
12-11-2015	Abdi	0	1	0	1	
12-11-2015	Annika	0	1	0	1	
12-11-2015	Aslanbek	1	0	0	1	
12-11-2015	Haydar	1	0	0	1	
12-11-2015	Mahmmoud	0	0	1	1	
12-11-2015	Abdiwahab	0	0	1	1	
11-11-2015	Abdi	1	0	0	1	Opg. fordeling
11-11-2015	Annika	1	0	0	1	Opg. fordeling
11-11-2015	Aslanbek	0	0	1	1	Opg. fordeling
11-11-2015	Haydar	1	0	0	1	Opg. fordeling
11-11-2015	Mahmmoud	1	0	0	1	Opg. fordeling
11-11-2015	Abdiwahab	0	0	1	1	Opg. fordeling
I alt		42	76	38	156	

Indhold

Resumé

Projekts hensigt er at oplyse om projektets tilblivelse fra start til ende, så kunden kan erfare, at opgaven er løst på tilfredsstillende vis og lever op til forventningerne.

Vi skal udvikler et program som skal simulerer et terningespil. Programmet skal kunne repræsenterer 2-6 spiller (aktør) som på skiftevis kaster med to terninger og efter resultatet kan lande på et felt mellem 2-12. Spillerne har 30.000 point hver at spille med. De kan lande på forskellige felter som kan have positive eller negative effekt på spillerens pengekonto.

Programmet skal også kunne udskrive et tekst som omhandler det aktuelle felt. Spillet er slut når alle på nær én spiller går bankerot.

Indledning

I kurserne Indledende Programmering og Udviklingsmetoder til IT-Systemer blev klassen introduceret for 3 delopgaver, som vi skal lave i løbet af semestret og som i sidste ende skal end med et velfungerende matador spil.

CDIO delopgave 3 som går ud på at lave en videreudvikling af vores terningespil fra delopgave 2 med flere funktioner og krav alt efter kundens ønsker.

Vi starter vores projekt med analyse og kravspecifikation som bygger på kundens vision af miljøet i programmet og yderligere diskuterer vi i gruppen for optimal fremgangsmåde og bestemmer de funktionelle og ikke funktionelle krav der er ønsket af kunden.

Da det er nødvendigt at programmet skal kunne kører på forskellige platforme har vi besluttet os for at skrive programmet i Java og har gjort brug af vores IDE, Eclipse. Kodens design illustreres i et designklassediagram, mens designsekvensdiagrammer viser forløbet i de mest centrale dele af systemet.

Testene er udført vha. JUnit, og har direkte forbindelse til kravene. UML og andre diagrammer bliver designet i et program der hedder Magic Draw.

Alt andet dokumentation skrives, redigeres og forberedes vha. Microsoft Office pakken.

Analyse

Kravspecifikation

Funktionelle krav:
F1: Spil mellem 2-6 personer
F2: Spillerne starter med 30.000 point
F3: Spillet slutter, når alle spillere på nær én er bankerot
F4: Brætspil hvor spillerne går i ring
F5: Udbygning af kravspecifikationer fra CDIO2 - *se bilag 1.5*

Navneordsanalyse

For at finde passende navne til klasser og metoder, er det en god idé at lave en navneordsanalyse, hvor man kigger på både navneord og udsagnsord. Hertil udvælger vi de passende navneord, som vi kan lave til klasser og udsagnsord, der kan blive til metoder.

Navneord:

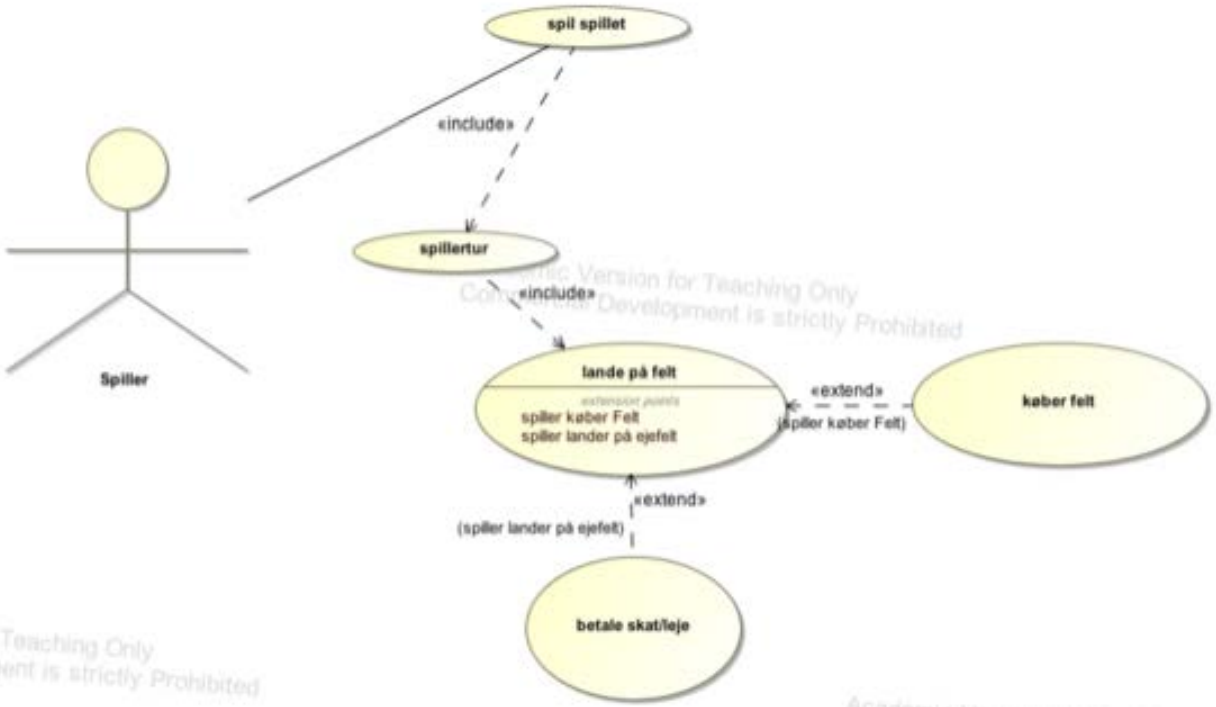
- Terning
- Spiller
- Felter
- Spilleplade/bræt

Da vi i forvejen havde klasser fra forgående pr

Udsagnsord:

-

Use case model



Figur 1 Use case diagram

ID: 1
Use case: Spil spillet
Kort beskrivelse: Matadorspillet gøres klar
Pre-condition: 2-6 spillere finder æsken frem med matadorspillet
Main flow:
1.1 Spillerne skiftes indtil en vinder er fundet
1.1.1 Se use case 'spillertur'
1.1.2 Hvis der kun er én spiller tilbage udråbes vinderen
1.2 Vinderen får besked
Primære aktører: 2-6 spillere
Post-condition: Spillet fortsætter indtil alle undtagen én går konkurs

ID: 2
Use case: Spillertur
Kort beskrivelse: Spillerne spiller deres tur og rykker til det felt de har slået
Pre-conditions: Spiller x er klar til spillertur
Primære aktører: aktive spiller
Main flow:
2.0 spilleren kaster med terningerne
2.1 spilleren lander på et felt – se use case 'Lande på felt'
Post-condition: Spillet fortsætter indtil alle har haft en tur

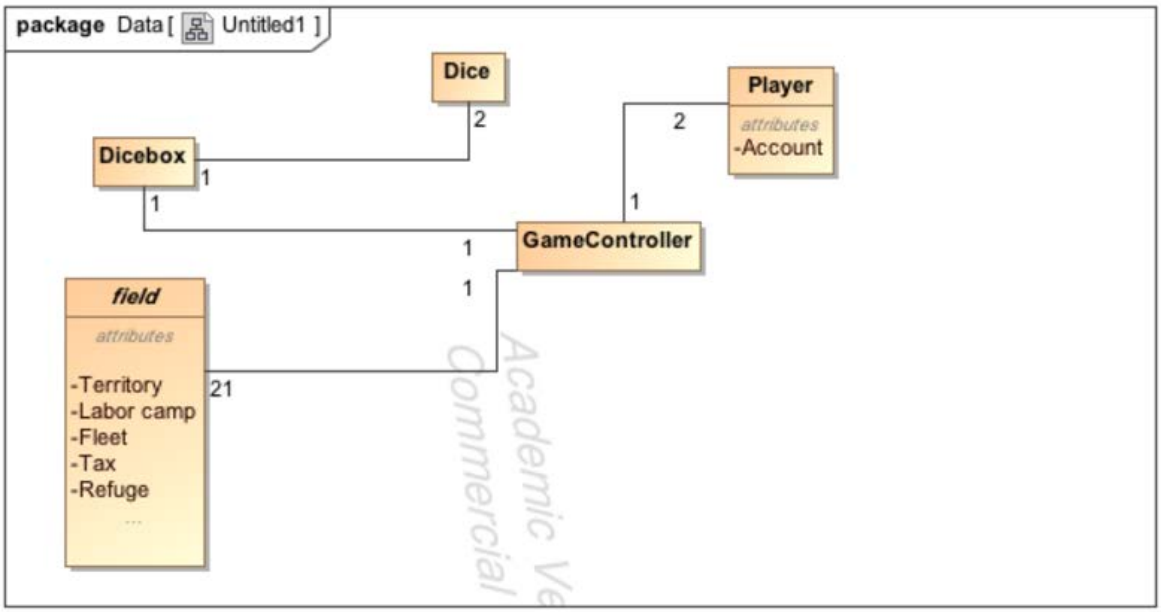
ID: 3
Use case: Lande på felt
Kort beskrivelse: Spillerne lander på et felt der enten kan købes, eller at der kan forekomme betaling, hvis det er ejet af en anden spiller
Pre-condition: Spillerne lander på et felt
Main flow:
3.1 Spiller x lander på et felt
3.1.1 Feltet er ikke ejet
3.1.1.1 Spilleren beslutter sig for at købe feltet – se use case 'køber felt'
3.1.2 Feltet er ejet
3.1.2.1 Se use case 'betal skat/leje'
3.1.3 Feltet er en fleet felt
3.1.3.1 Se use case 'betal skat/leje'
3.1.4 Feltet er en labor camp
3.1.4.1 Se use case 'betal skal/leje

3.1.5 Feltet er en tax felt
3.1.5.1 Betal x beløb eller x %
3.1.5.2 Se use case 'betal skat/leje'
Primære aktører: Aktive spiller
Post-condition: spilleren er færdig med at lande på felt

ID: 4
Use case: Køber felt
Kort beskrivelse: Hver spiller har mulighed for at købe et felt, hvis de lander på det, og det ikke er ejet af nogle af de andre spillere
Pre-condition: Spiller lander på et felt der ikke ejes af nogle af de andre spillere og vil købe det
Main flow:
4.1 Spiller betaler feltets pris
4.2 Spiller bliver ejer af feltet
Primære aktører: Aktive spiller
Post-condition: Spilleren ejer feltet

ID: 5
Use case: Betale skat/leje
Kort beskrivelse: Hver spiller skal betale leje til den pågældende spiller feltet er ejet af, eller skat hvis det er en betalingsfelt
Pre-condition: Spiller lander på et felt der er ejet af en anden spiller, eller som er et skatfelt
Main flow:
5.1a Spiller x betaler leje til den spiller der ejer feltet
5.1b Spiller x betaler skat til systemet, idét det er et skatfelt
Primære aktører: Aktive spiller
Alternativt flow:
5.1.1 Spilleren har ikke nok penge og går konkurs
Post-condition: spilleren har betalt leje

Domænemodel



Figur 2 Domæne model

En domænemodel er en objekt orienteret analyse i UML, som viser et programs fysiske opførsel. Programmets opførelse viser de forskellige operationer et objekt kan opfylde. En domænemodel opbygges uden de faglige termer i Java sprog, men beskriver blot systemets fysiske dele i de forskellige klasser.

En domænemodel ligner meget et klassediagram dog uden attributter med typer og metoder. I vores ovenstående domænemodel har vi Dice, Dicebox, Account, Player, Field, Tax, Refuge, territory, Labor camp, Fleet og GameController som styrer klasserne og GUI der visualiserer vores spil.

De forskellige associationer viser forholdet mellem objekterne. Forholdet og opførslen i vores tilfælde er et Dicebox, som associeres til Dice og Gamecontroller, og samtidig har player klassen association til Account og Gamecontroller. Så har vi Field som har association til GameController og er samtidig Superklasse til Tax og Refuge, Territory, Labor camp og Fleet. GameController har association til GUI.

Design

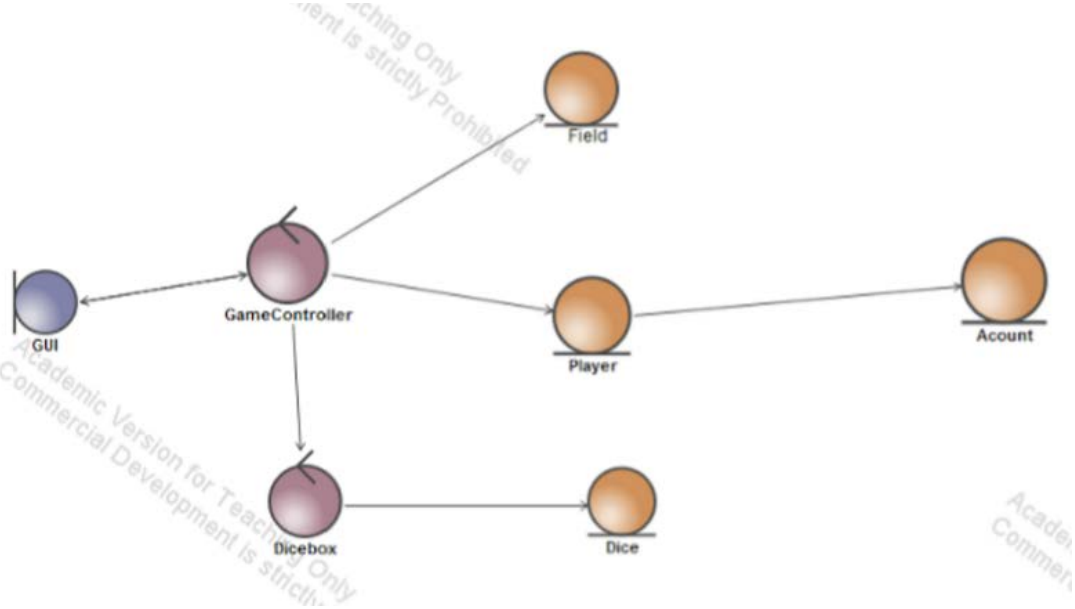
BCE-model

Vores BCE-model illustrerer et terningspil, som har 2-6 deltagere. Det er opbygget af 3 forskellige slags klasser, som er følgende:

Boundary er en klasse der håndterer interaktion og kommunikation med systemets omgivelser.

Controller er en klasse der indkapsler use casens specifikke opførsel, samt fordeler ansvaret på systemets andre objekter og fastlægger sekvens i handlinger.

Entity er en klasse der modellerer det systemet handler om, denne har også hukommelse.



Figur 3 BCE Mdel

Figur 3illustrer vores BCE model, her er Dicebox en controller, det skyldes at den ikke indeholder data med får uddelegeret ansvaret og håndtere data fra Dice.

Vi har arbejdet videre på cdio2 bce model, se bilag 1.1

Vi har brugt følgende 7 klasser, som er navngivet, GameController, Dicebox, Dice, Player, Account, Field, og GUI.

Field har 5 klasse nedarvet, som er følgende, Territory, Refuge Tax, Labor camp, Fleet.

Følgende proces har en deltagende aktør per gang.

Det første skridt, er at spillet bliver sat i gang ved at kører run kommandoen og via GUI bliver givet input fra aktøren og man får mulighed for at vælge antal spiller mellem 2-6 mulig deltagende spiller og herefter kan man vælge navne til de deltagende spiller.

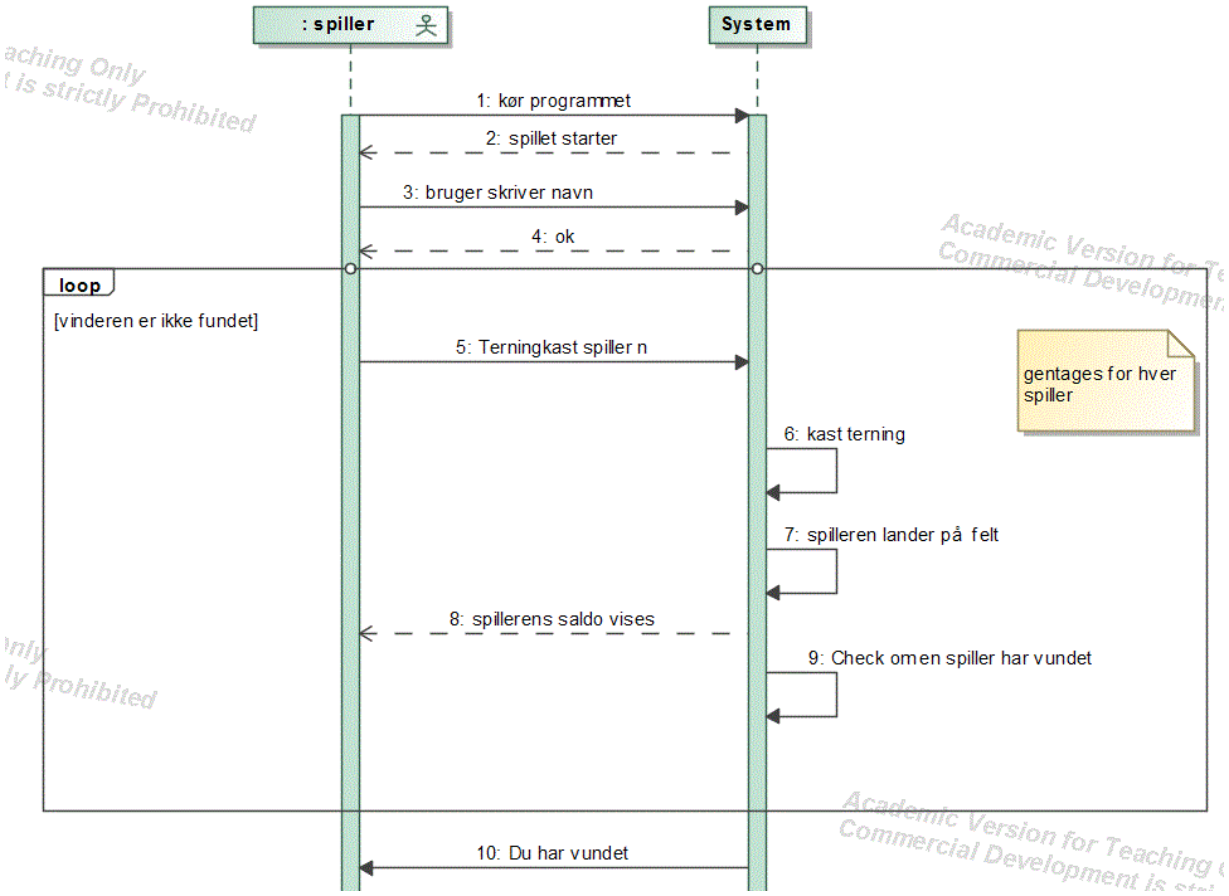
Efterfølgende bliver der givet et input fra den første aktør og dette bliver kommanderet videre til vores controller som er GameController, her bliver der der uddelegeret ansvaret i systemet.

GameController sender en kommando til Dicebox om at kaste med 2 terninger og dette bruger Dicebox, ved at kalde funktionen via. Dice entity.

Outputtet fra bliver kaldt ud af GameController og bliver kommanderet til Player så den får association med den aktør der spiller spillet, og dette output bliver så lageret i Account klassen.

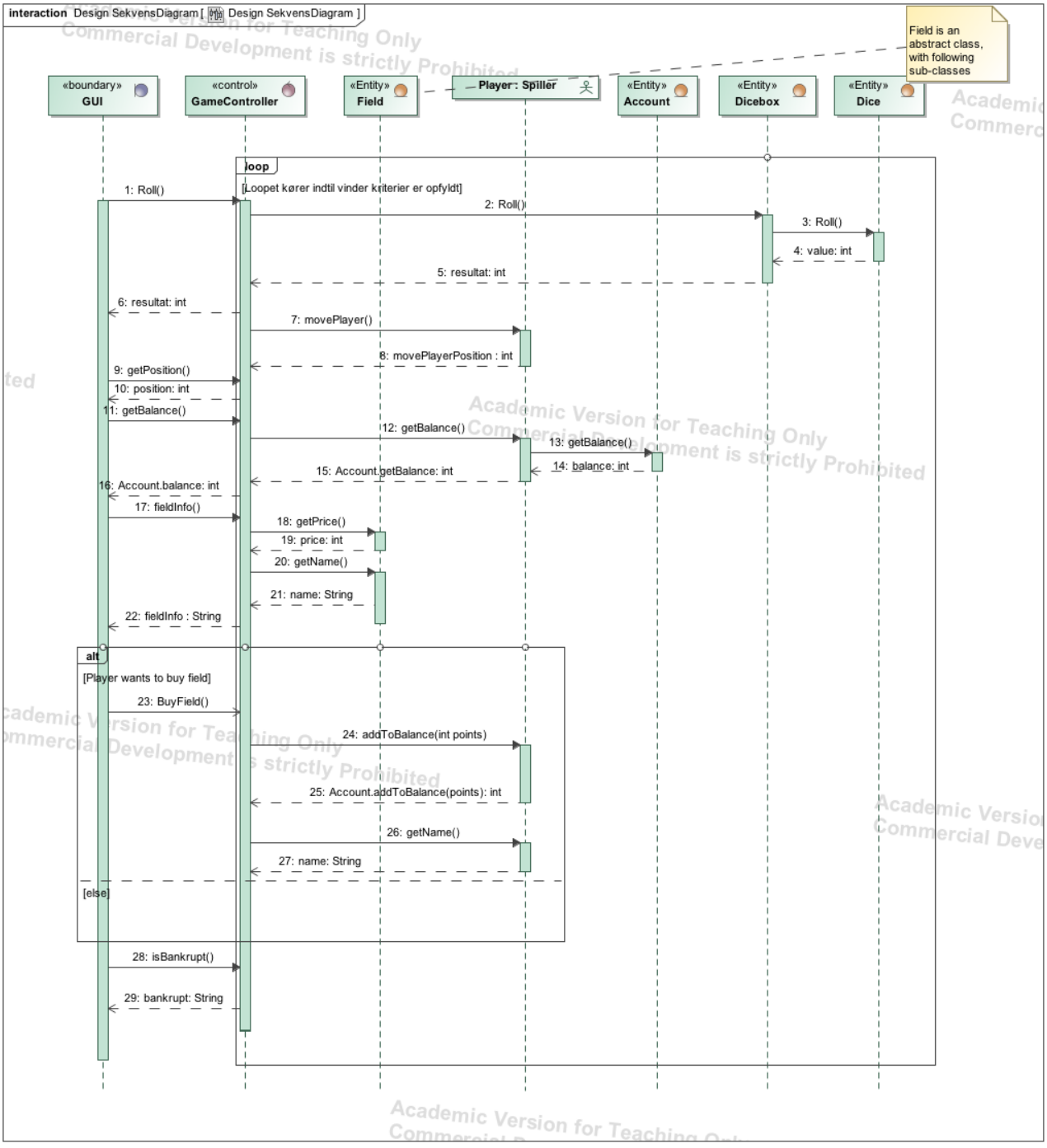
GameController kalder outputtet fra Account og dette bliver kommanderet til Field.

System-sekvensdiagram



I vores systems sekvensdiagram bliver kommunikationen mellem spilleren og systemet beskrevet. Som illustreret på ovenstående sekvens diagram, vises der at man skal starte med at køre programmet, så starter programmet hvor man derefter indtaster navne af antal spiller som deltager i spillet, så starter spillet som kører i et loop, indtil kriteriet for at vinden er opfyldt. Efterfølgende afsluttes loopet og systemet angiver hvilken spiller der har vundet spillet.

Design-sekvensdiagram



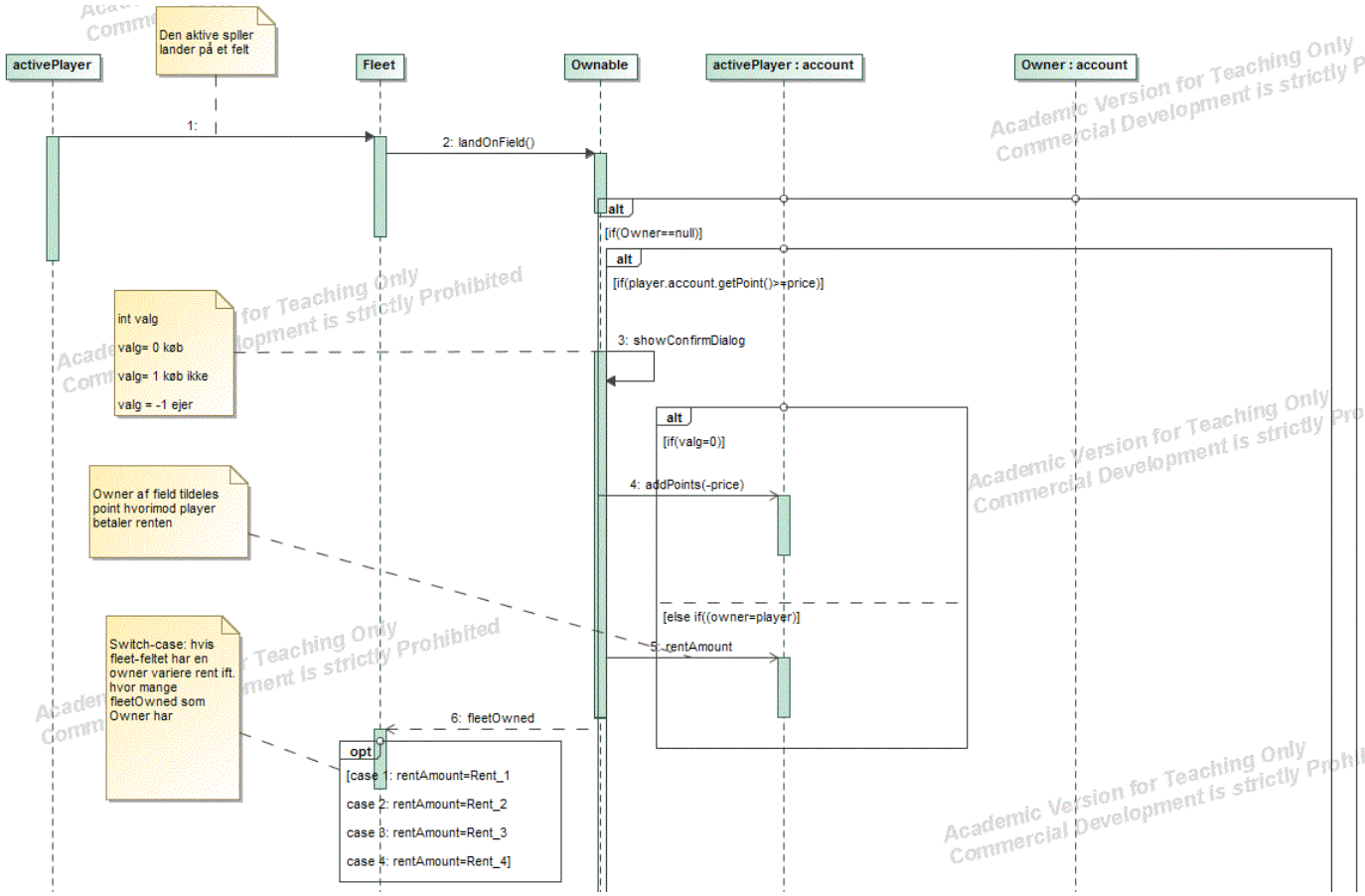
I overstående design-sekvensdiagram bliver kommunikationen mellem klasserne beskrevet samt GUI. Vi har i dette program en GameController, som styrer hele spillet samt henter de brugbare metoder fra de andre klasser, og dernæst svarer klasserne tilbage til GameControlleren med de vigtige informationer.

Controlleren afgiver input til GUI. Terningerne(Dice) bliver rullet i raflebægeret(Dicebox), og GameControlleren får en besked om udfaldet, som derefter videresendes til GUI. Samtidig giver controlleren besked til spilleren og feltklassen, og derved bliver brikkerne rykket det antal øjne

terningen viser. Alt efter hvilket felt brikkerne lander på, vil det få en positiv eller negativ effekt på spillernes konti, idét feltet kan være købt af en anden spiller, som man så skal betale leje af eller blive beskattet.

Dette proces køres i et loop, indtil kriteriet for at vinde er opfyldt. Kriteriet er så indtil alle på nær én spiller går bankerot.

Design-sekvensdiagram-landOnFleet



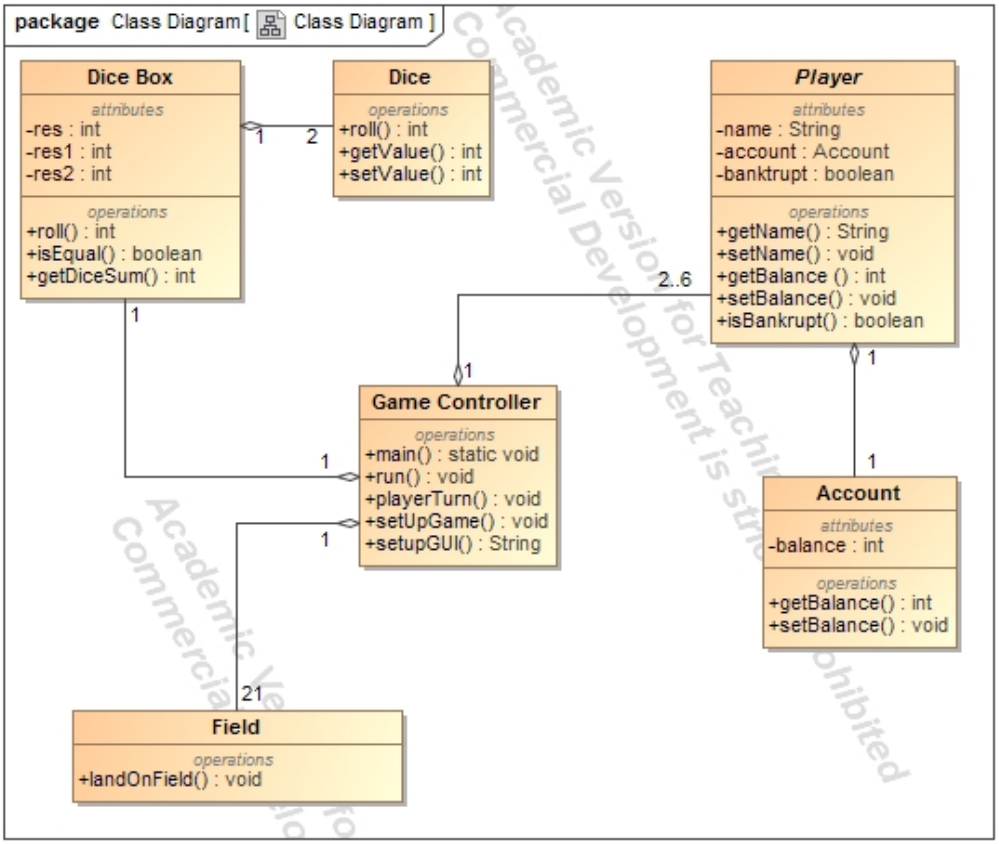
Design sekvensdiagrammet landOwnFleet Som illustreret på ovenstående diagram, viser hvad der sker når en spiller lander på Fleet: når man lander på fleet feltet bliver der kaldt en metode fra superklassen Ownerable,som hedder landOnField.

Klassediagram

For at gøre det mere overskueligt har vi valgt at dele klassediagrammet op i to. Disse beskriver de klasser vi gør brug af under implementeringen samt metoder og attributter. Pilene imellem klasserne beskriver deres associationer. Hertil har vi valgt at gøre brug af low coupling, hvilket vil sige, at klasserne ikke skal være alt for afhængige af hinanden. Dette hænger sammen med 'high cohesion', der sørger for at klasser er fokuserede og styrbare.

Klassediagram 1

Dette klassediagram står for selve gameplayet, hvortil spillerne, terninger og spilbræt/felterne bliver lavet.

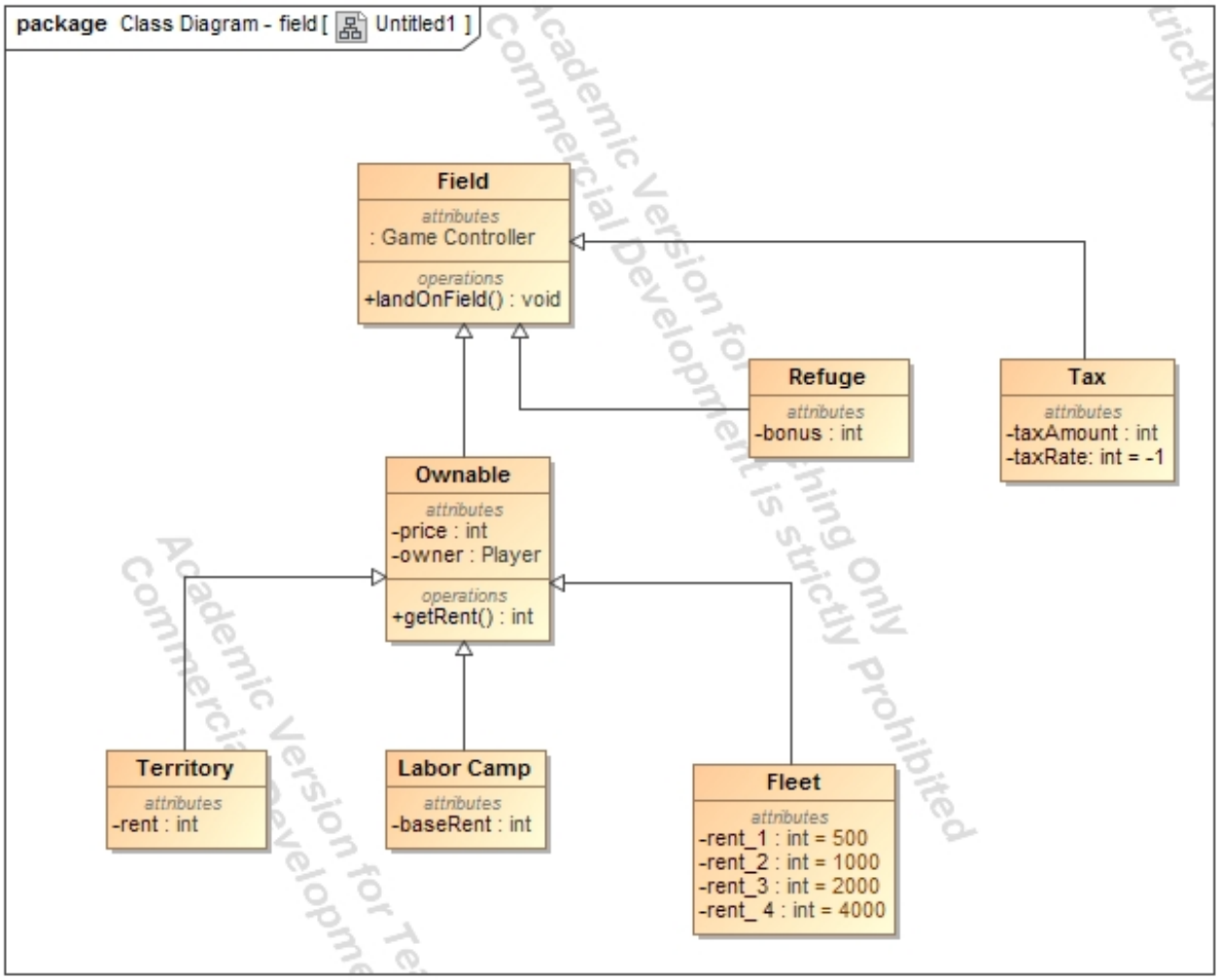


Desuden er der angivet multipliciterter, som beskriver forholdet mellem klasserne – altså har ét raflebæger to terninger osv.

Klassediagram 2

Dette klassediagram beskriver feltklassen, som nedarver forskellige klasser. Der findes forskellige former for felter, som har forskellige funktioner. Hertil er der brug for klasser, der kan beskrive dette. Field og Ownable er begge abstrakte klasser, hvilket betyder, at de er superklasser. Dette gør, at man ikke kan udføre operationer i disse klasser, hvilket derfor skal ske i de klasser, der earver derfra.

I UML ses en abstrakt klasse ved at metoden står i kursiv.



Implementering

GRASP

I den her afsnit vil vi gerne forklare hvordan vores klasser arbejder sammen ved at lave en GRASP dokumentation. I bogen Applying UML and Patterns skrevet af Craig Larman beskrives 9 mønstre som indgår i GRASP. Hver af dem hjælper til at løse nogle af de problemer i objektorienteret analyse og som forekommer næsten i ethvert projekt for softwareudvikling. Således GRASP er et veldokumenteret, standardiseret og tidstestede principper for objektorienteret analyse og ikke et forsøg på at bringe noget helt nyt.

Eksemple på disse 9 mønstre som anvendes i GRASP er: Information expert, Creator; Controller, Low coupling, High cohesion, Polymorphism, Pure Fabrication, Indirection og Protected Variations.

I vores program bliver der anvendt 4 af disse mønstre. Det er Controller, Creator, Information Expert, Low Cobling og så vil vi argumentere lidt for High Cohesion selvom vi ikke har udnyttede den i vores program.

Controller:
En controller er en non-userinterface klasse der styrer udførelsen af en usecase og uddelegerer opgaver rundt i programmet. I vores tilfælde er det hovedcentralen i programmet som henter metoder fra andre objekter og styrer spillets gang. Vores controller hedder GameController. Den kalder på f.eks. Dicebox når der skal rafles og Player når der skal oprettes en konto osv.

Princippet er, at der så vidt som muligt skal være én controller til at styre programmet, hvilket er imødekommet i vores spil. Kort sagt ’controller klassen’ styrer hele spillet.

Information expert:
En klasse er Information expert for den viden den har. Et eksempel på det i vores spil, vil være at terning klassen har viden om det antal øjne en terning har. Raflebaegeret holder så styr på den samlede antal øjne, som de to terninger har.

High Cohesion:
High Cobling er når alle klasserne har forbindelse til hinanden og går ud på at splitte programmet op i klasser og subsystemer, hvor der skal være samhørighed samtidig med, at koblingen er lav. Vi har samlet vores metoder og attributter, der hvor det giver mening.

Low Cobling:
Low Cobling er når alle klasser har mindre forbindelse til hinanden og er i modsætning til High Cobling. Dvs. Der er en lettere afhængighed mellem klasserne, og deres forbindelse til hinanden går igennem controlleren. Den bedste metode er at kode i forhold til Low Cobling, og det vi har gjort i vores tilfælde er at raflebaegeret klassen har forbindelse til terning klassen, hvor så raflebaegeret også har forbindelse controlleren. Det samme gøre sig gældende for at spiller klassen har forbindelse til konto klassen, hvor så spiller klassen kun har forbindelse til controller klassen.

Vi udbygger spillet fra CDIO 2. De vigtigste ændringer er, at der bliver tilføjet felter samt det faktum, at der nu skal være mulighed for at være 2-6 spillere efter eget valg. Desuden skal der tilføjes metoder til de forskellige felter, hvortil vi gør brug af nedarvning fra 'Fields.'

Vi har ikke lavet et fuldt funktionelt spil. Dog har vi fået kreeret et spil, hvor man kan være op til seks personer.

Fejl:

- Crasher efter x antal runder, selvom der ikke er fundet en vinder.
- Spillere kan købe det samme felt frem for at nr. 2, der lander på feltet betaler leje.
- Det er ikke initialiseret, at spillerne taber, hvis de går bankerot (ved endnu ikke, hvad der sker, hvis de går i minus.) – konkl. De betaler ikke rent, og derfor er det svært for spillerne at gå i minus
- Bilerne fjerner sig ikke fra felterne, når de først har været. Derfor er der mange biler på hele brættet.
- Spillet crasher pludseligt
- Spillet er fucked

Hvad er nedarvning

I objektorienteret design- og programmering betyder nedarvning også kaldet på engelsk inheritance at en sub-klasse eller child-class nedarver identiske funktionaliteter fra en øvre anden super-klasse super-class, hvorpå der kan laves nye funktioner for den specifikke sub-klasse uden at påvirke den øvre klasse. UML-notationen er en solid streg fra sub-klassen med en lukket, tom pil pegende på super-klassen. I vores program giver dette sig til udtryk i vores klassehierarki, hvor Field-klassen og Ownable er en (abstrakt) og subklasse af Field er henholdsvis Territory, Tax, Refuge, Labor Camp og Fleet.

Nedarvning

I dette projekt arbejder vi med nedarvninger af klasser,
Vi har en en klasse kaldt Field og denne har 5 under klasser, Territory, Tax, Labor Camp, Tax og Refuge.
I dette projekt har vi brugt LandonField metode, som her bliver nedarvet fra Field klassen.

Sådan som det er struktureret er, at Field klassen har en definition af metoden og bliver overskrevet, alt efter hvilket felter på man lander på.

```
        public abstract void landOnField(Player player ); {
}
```

Vi kan se hvordan overskrivning fungere, ved at se på Territory klassen, denne indeholder instrukser og kommandoer som håndtere hvad der skal ske når man lander på en given felt.

```
public void landOnField(Player player ) {
                        if (this.getOwner() == null) {
                                int price = this.getPrice();
                                player.addToBalance(-price);
                                this.setOwner(player);
                        } else if (this.getOwner() != player) {
                                int rent = this.getRent();
                                player.addToBalance(-rent);
                                this.getOwner().addToBalance(rent);
                        }
```

Test

I følgende afsnit vil der ses den mængde JUnits tests som er bleven kørt.
Disse tester metoder i programmet, eller større sammenhænge.
Vores udføres hovedsageligt som Whitebox tests, men da vi kender kildekoden, vil vi tilstræbe os at lave mere strukturerede tests, også kendt som whitebox tests.

Vi har i vores projekt testet de fem LandOnFields der er blevet krævet i opgavebeskrivelsen.
Det er normalt at man tester grænser i sine tests.
Dette har vi undladt i landOnFields testen, da vores klasser er prædefineret,
og vi henter disse fra GameController klassen.
Vi har derfor valgt at teste funktionaliteten af disse klasser.
Vi har inkluderet alt test kode i bilag 1.3 , men screenshots som projektbeskrivelsen kræver er inkluderet efter Test cases afsnittet.

Blackbox tests er tests der fokuserer på input og output.

Testen tester forskellige funktioner uden at kigge på hvordan programmet er opbygget, og uden at man kigger på kildekoden.

Whitebox test, er strukturerede tests, hvor man bl.a. undersøger sine forgreninger,

statements og løkker igennem, og ser om der er kode der ikke bliver eksekveret,
og om funktionerne fungerer som de skal.

Preconditions, er hvad skal have sket før selve testen kan køres.
Det vil sige hvis variabler eller objekter skal være initialiseret, eller sat til bestemte værdier, eller visse metoder er kaldt. Testen består af et tjek for, om noget er sandt eller falsk.

Dette kunne eksempelvis være, om visse værdier er ens.

Postconditions er hvilke resultater man forventer, og hvad der skal til før at testen er sand

Konklusion

Vi har desværre ikke fået implementeret et fuldt funktionelt spil, og derfor vil vi beskrive de fejl, vi oplever når vi kører det. For det første er det muligt at spille spillet alene.

Fordi vi ikke har kunne få funktionen GUI.removeCar til at arbejde sammen med playerTurn-metoden, bliver bilerne stående hver gang de har besøgt et felt.

Spillet crasher efter nogle runder, fordi vi har lavet en forkert implementering af, hvordan bilerne skal hoppe i rundt på brættet. Vi kunne ikke finde en måde, hvorpå det fungerede, men vi véd at implementeringen crasher

Ud fra vores arbejdsmetode, har vi denne gang valgt at fokusere forward og af reverse engineering ved bla. at lave analyse, design samt implementering.

Ud fra de testcases vi har lavet og Junit test, kan vi se at ved at arbejde med dem, kunne vi faktisk udrette fejl og mangler ved kildekoden, eksempelvis logiske fejl. Denne metode er meget nyttig kendes ved Test Driven Development.

Litteraturliste

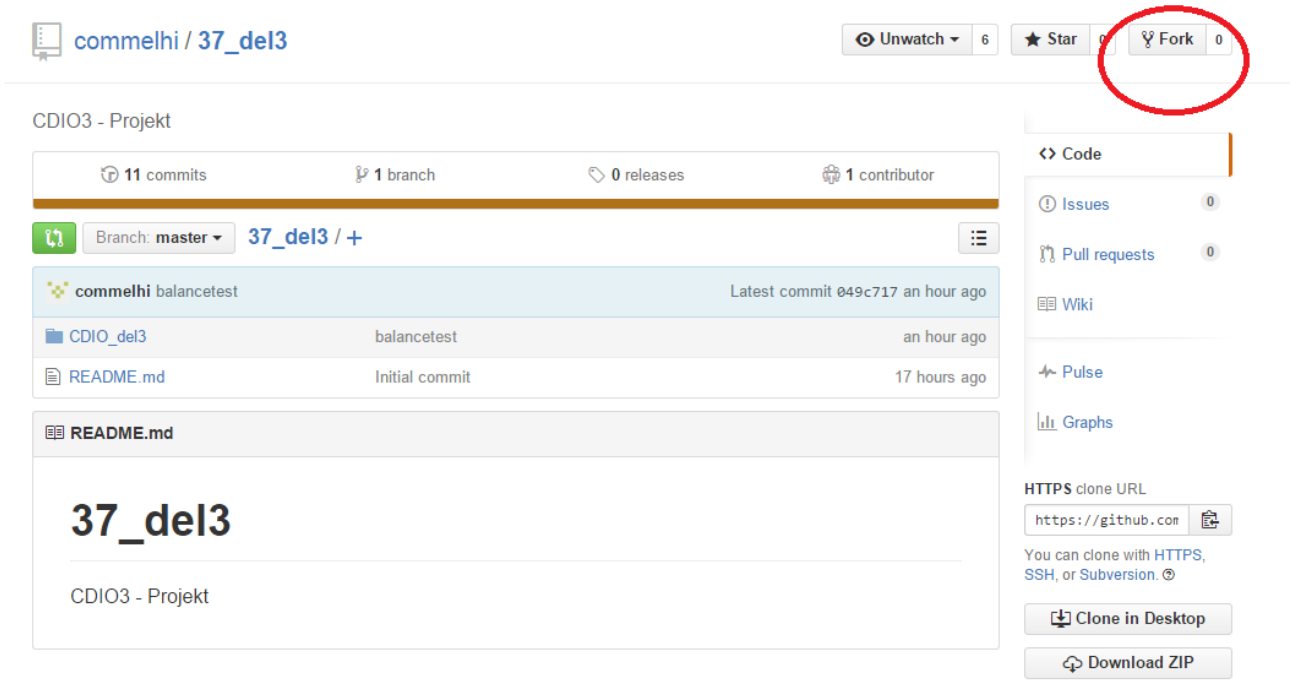
CDIO2 projekt – Se bilag 1.1

http://javabog.dk/

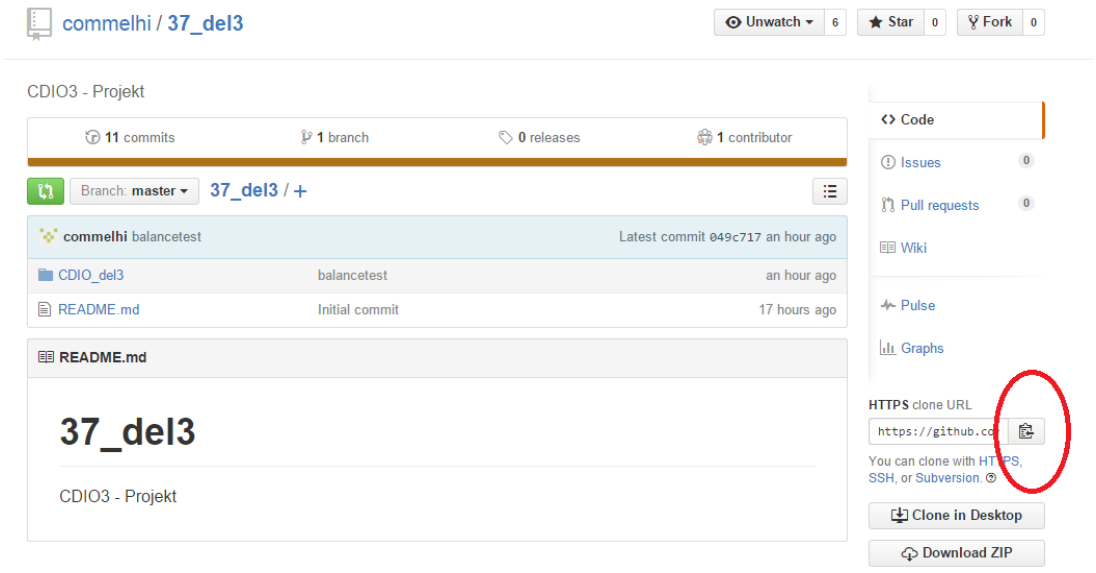
Git Vejledning til at hente Java projekt fra Github.com

1. Først skal vores repository hentes fra Github vha. linket og forkes.

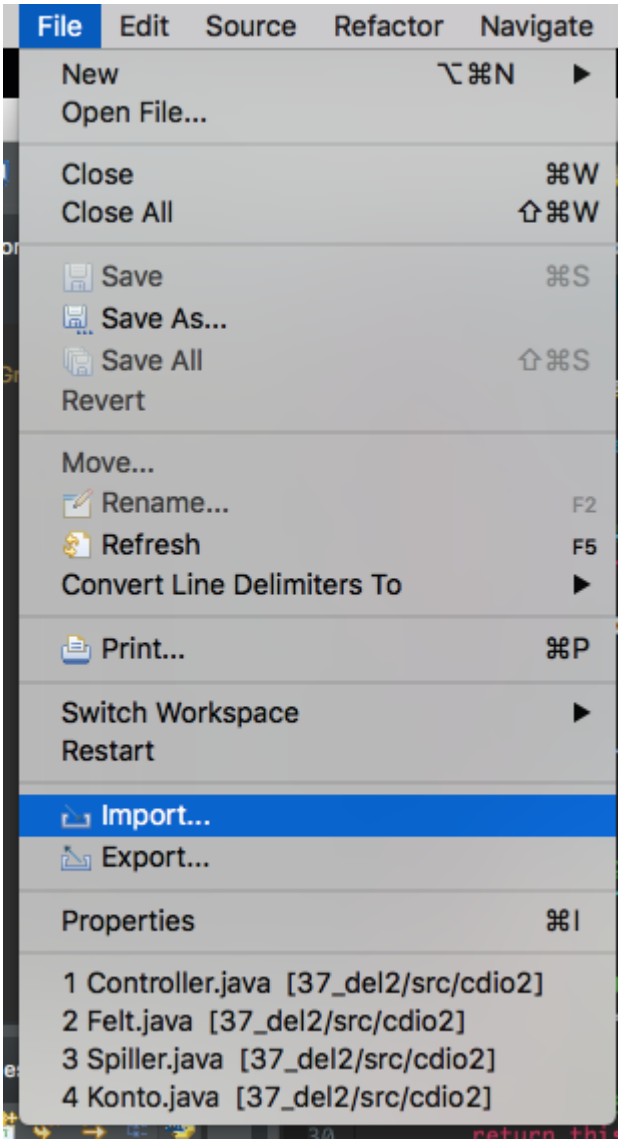
   https://github.com/commelhi/37_del3.git



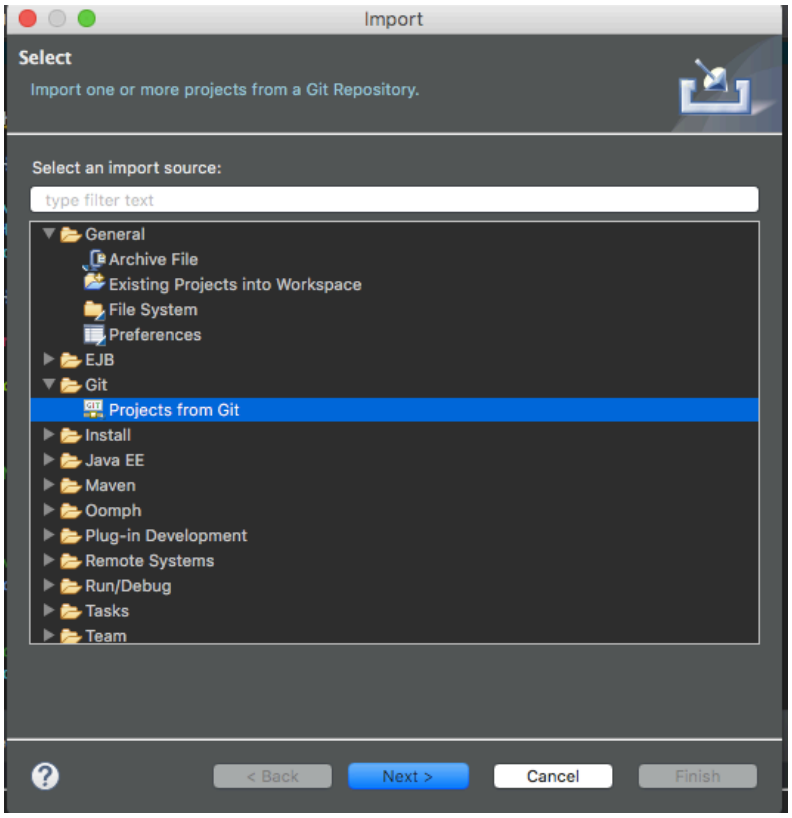
2. Herefter skal linket til vores repository kopieres



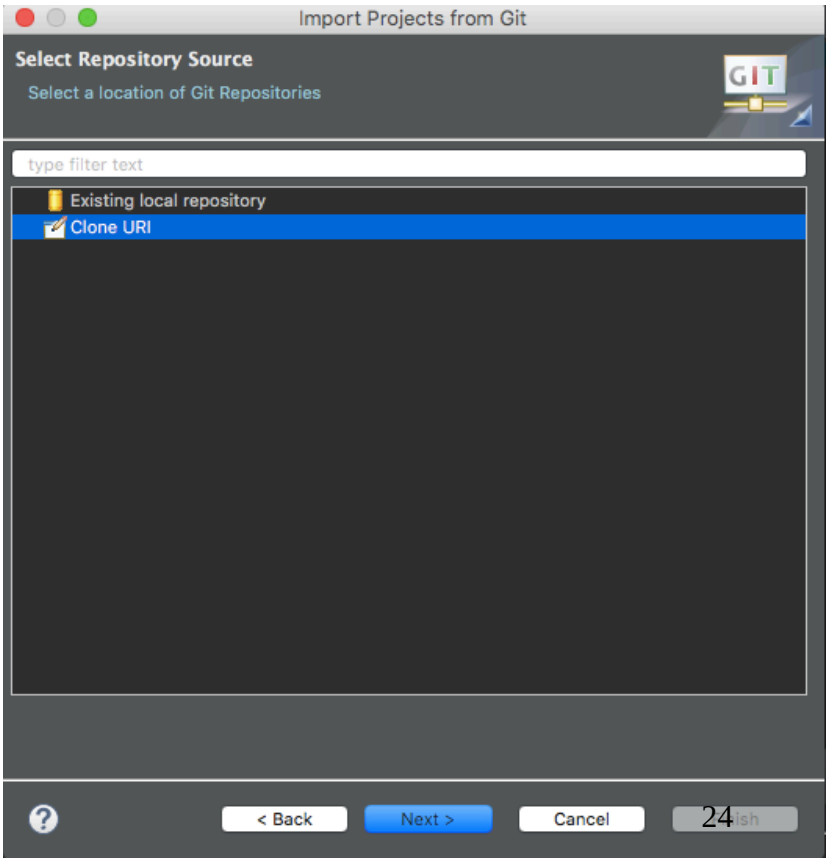
3. Nu skal projektet importeres i Eclipse



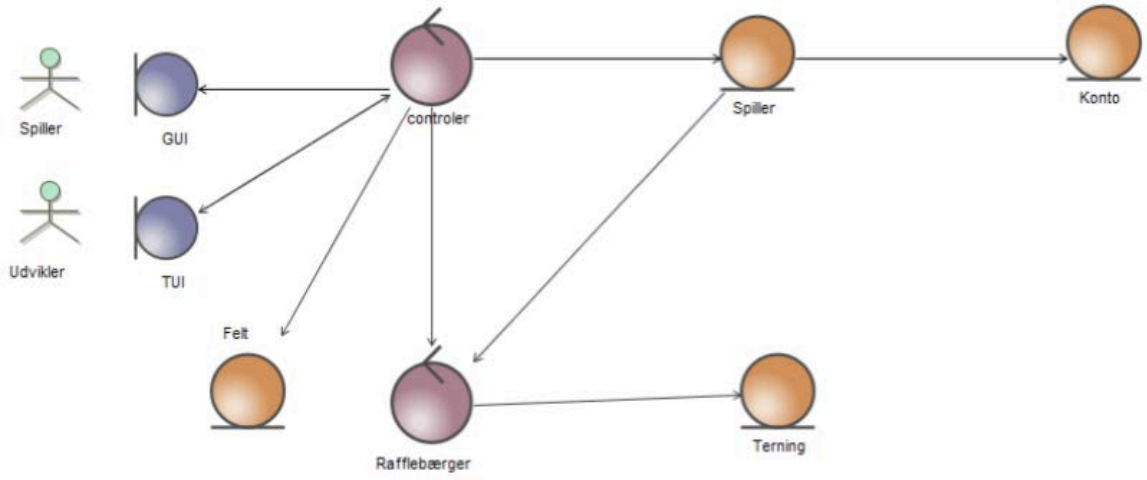
4. Vælg Git > Projects from Git



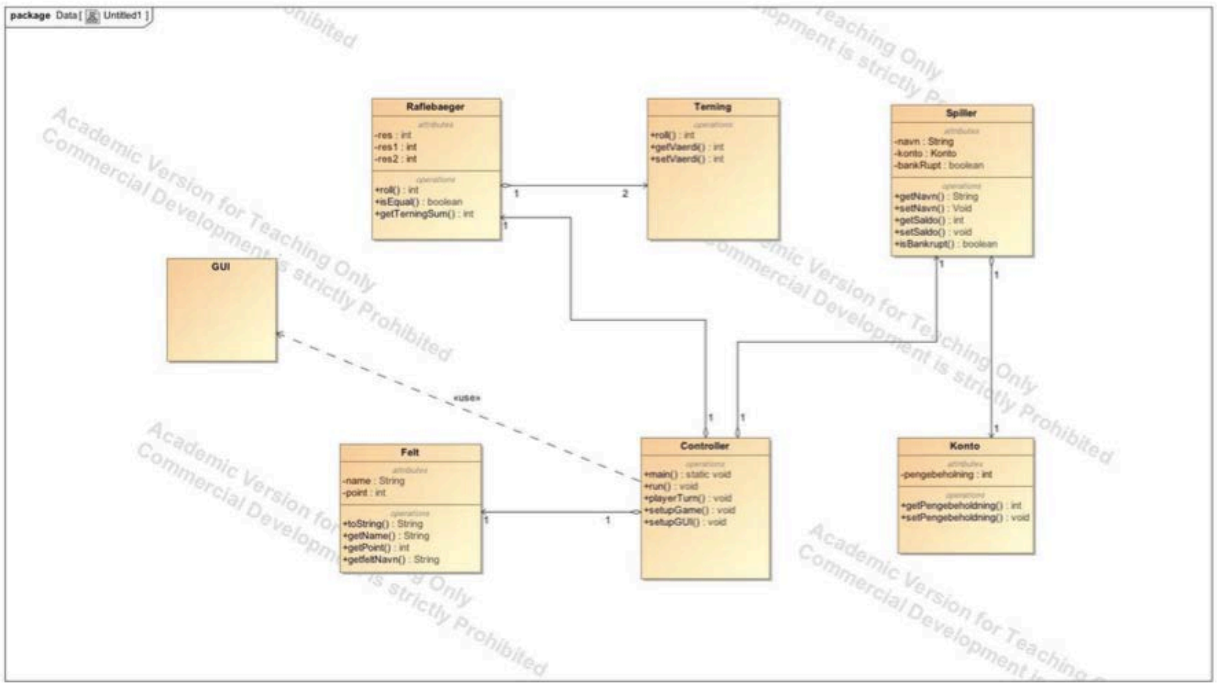
5. Vælg clone URL og tryk næste



Bilag 1.1



Figur 4 BCE model cdio2



Figur 5 Klassediagram cdio2

Bilag 1.2

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

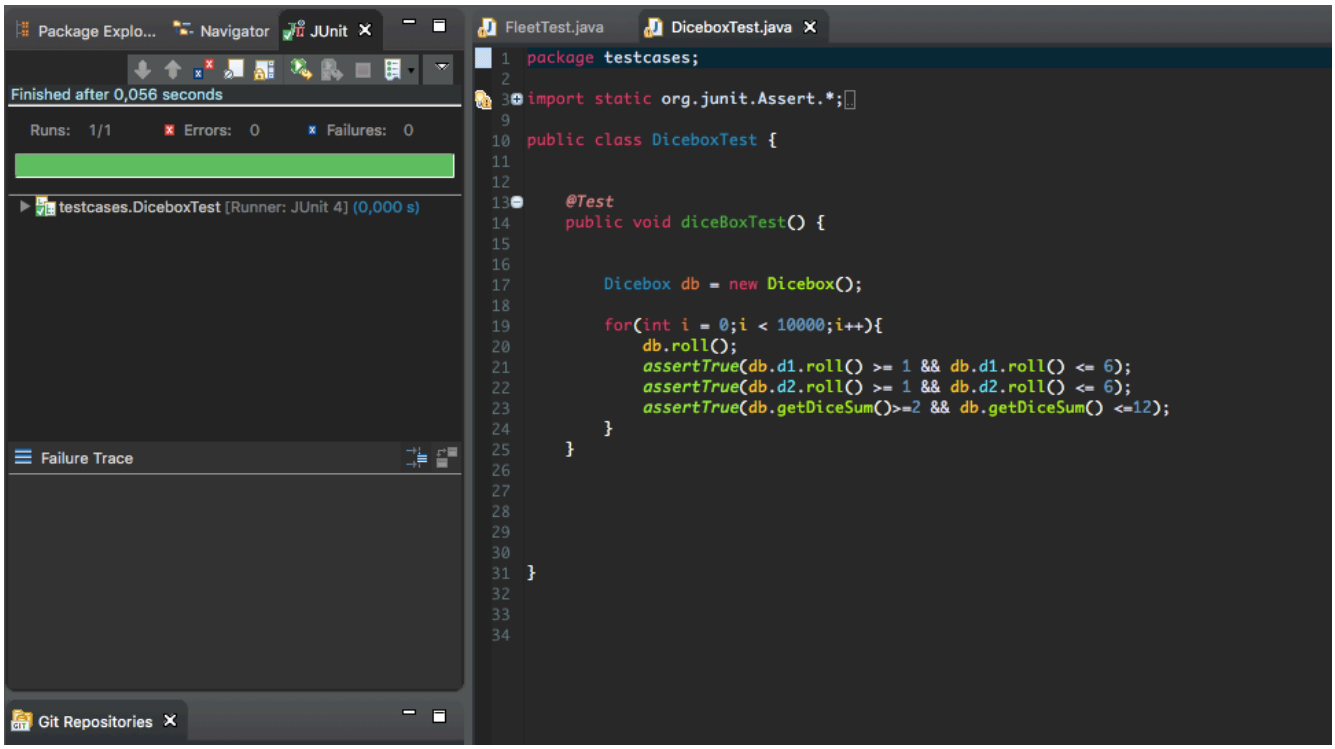
Typer af felter:

- **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

Bilag 1.3

Testkode

Test af Dicebox (se kildekode i eclipse –> test -> testcases)



Figur 6 test af Dicebox klassen

Test kode

Test af Fleet

```
package test;

import static org.junit.Assert.*;
import org.junit.Assert;
import org.junit.Before;
import org.junit.Test;
import entity.Field;
import entity.Player;
import entity.Territory;
import entity.Fleet;


public class FleetTest {

		private Player spiller1;

		private Player spiller2;


		private Fleet fleet1;

		private Fleet fleet2;


		@Before

		// setUp metoden bliver kaldt FøR hver test metode

		public void setUp(){

				fleet1 = new Fleet("Fleet1, pris: 16000", 16000);

				fleet2 = new Fleet("Fleet2, pris: 13000", 13000);



				spiller1 = new Player("Obama", 0);
```

```
                spiller1.addToBalance(10000);

                spiller2 = new Player("Clinton", 1);
                spiller2.addToBalance(15000);
        }
        // En spiller lander på et frit felt
        @Test
        public void testNewOwner() {
                int expected, actual;

                // en simpel test for at sikre at startbeløbet er rigtigt
                int saldo = spiller1.getBalance();
                Assert.assertEquals(40000, saldo);


                //spiller 1 lander på Fleet, den koster 16.000
                fleet1.landOnField(spiller1);

                boolean bought = false;

                //er feltet ledigt? hvis ja, køb det
                if(spiller1.getInformation() == 0) {
                        bought = spiller1.buyField(fleet1);
                }
```

```
		// fik han købt feltet? hvis ja, test om prisen er fratrukket saldoen
		if(bought){
			Assert.assertEquals(1, spiller1.getInformation());
	expected = 40000 - 16000;
			actual = spiller1.getBalance();
			Assert.assertEquals(expected, actual);
		}else{
			//ellers burde beløbet ikke være fratrukket
			Assert.assertEquals(40000, spiller1.getBalance());
		}
}

@Test
public void testAlreadyOwned() {
		int expected, actual;

		// en simpel test for at sikre at startbel�bet er rigtigt
		int saldo = spiller1.getBalance();
		Assert.assertEquals(40000, saldo);

		// en simpel test for at sikre at startbel�bet er rigtigt
		saldo = spiller2.getBalance();
		Assert.assertEquals(45000, saldo);
```

```
//spiller 1 lander p[?] Fleet, den koster 16.000
fleet1.landOnField(spiller1);

boolean bought = false;

//er feltet ledigt? hvis ja, køb det
if(spiller1.getInformation() == 0) {
        bought = spiller1.buyField(fleet1);
}

// fik han købt feltet? hvis ja, test om prisen er fratrukket saldoen
if(bought){
        expected = 40000 - 16000;
        actual = spiller1.getBalance();
        Assert.assertEquals(expected, actual);
}else{
        //ellers burde beløbet ikke være fratrukket
        Assert.assertEquals(40000, spiller1.getBalance());
}

if(bought){

        fleet1.landOnField(spiller2);
        boolean rented = false;
```

```
                                        // hvis feltet ejes allerede, og vi kan leje det
                                        if(spiller1.getInformation() == -1){
                                                rented = spiller2.rentField(fleet1);
                                        }
                                        //hvis det lykkedes at leje feltet, test om lejepris er
fratrukket saldo
                                        if(rented){
                                                expected = 45000 - 500;
                                                actual = spiller2.getBalance();
                                                Assert.assertEquals(expected, actual);
                                        }
                        }
                }

                @Test
                public void testMyOwnFleet() {
                        int expected, actual;

                        // en simpel test for at sikre at startbeløbet er rigtigt
                        int saldo = spiller1.getBalance();
                        Assert.assertEquals(40000, saldo);

                        // land på et frit felt
                        fleet2.landOnField(spiller1);
                        boolean bought = false;
```

```
//er feltet ledigt? hvis ja, køb det
if(spiller1.getInformation() == 0) {
		bought = spiller1.buyField(fleet2);
}

// fik han købt feltet? hvis ja, test om prisen er fratrukket saldoen
if(bought){
		System.out.println(spiller1.getInformation());



		expected = 40000 - 13000;
		actual = spiller1.getBalance();
		Assert.assertEquals(expected, actual);
}else{
		//ellers burde beløbet ikke være fratrukket
		Assert.assertEquals(40000, spiller1.getBalance());
}
System.out.println(spiller1.getInformation());

// hvis man lander på eget felt igen, skal info være 1
if(bought){
		Assert.assertEquals(1, spiller1.getInformation());
}
```

```
                        // hvis man lander på eget felt igen, skal der ikke trækkes noget
                        if(bought){
                                fleet2.landOnField(spiller1);

                                //test om der er trukket beløb fra konto (FALSE)
                                expected = 40000 - 13000;
                                actual = spiller1.getBalance();
                                Assert.assertEquals(expected, actual);
                        }
                }
}
```

Test af LaborCamp

```
package test;

import static org.junit.Assert.*;
import org.junit.Assert;
import org.junit.Before;
import org.junit.Test;
import entity.Player;
import entity.LaborCamp;

public class LaborCampTest {

        private LaborCamp laborcamp1;

        private LaborCamp laborcamp2;

        private Player spiller1;
```

```
	private Player spiller2;


	@Before
	// setUp metoden bliver kaldt FøR hver test metode
	public void setUp(){
		laborcamp1 = new LaborCamp("laborcamp1, leje: 1000, pris: 6000", 1000, 6000);

		laborcamp2 = new LaborCamp("laborcamp2, leje: 5000, pris: 11000", 5000, 11000);

		spiller1 = new Player("Obama", 1);
		spiller1.addToBalance(10000);

		spiller2 = new Player("Clinton", 2);
		spiller2.addToBalance(15000);
	}
	// En spiller lander p? et frit felt
	@Test
	public void testNewOwner() {

		// en simpel test for at sikre at startbeløbet er rigtigt
		int saldo = spiller1.getBalance();
		Assert.assertEquals(40000, saldo);


		//spiller 1 lander på Territory1, den koster 6.000
```

```
                laborcamp1.landOnField(spiller1);

                boolean bought = false;
                // hvis getInformation returnere 0, så er feltet ledigt.
                if(spiller1.getInformation() == 0) {
                        bought = spiller1.buyField(laborcamp1);
                }
                //hvis han fik lov til at købe feltet, så test saldo - pris
                if(bought){
                        // udregn det forventede resultat
                        int expected = 40000 - 6000;

                        //få saldo ud af spillerens konto
                        int actual = spiller1.getBalance();
                        //test om de tal er ens
                        Assert.assertEquals(expected, actual);
                }else{
                        // hvis ikke han kunne købe feltet, så skal
saldoen være startbeløbet
                        saldo = spiller1.getBalance();
                        Assert.assertEquals(40000, saldo);
                }
        }
```

```
@Test
public void testAlreadyOwned() {
        int expected, actual;

        // en simpel test for at sikre at startbel�bet er rigtigt
        int saldo = spiller1.getBalance();
        Assert.assertEquals(40000, saldo);

        // en simpel test for at sikre at startbel�bet er rigtigt
        saldo = spiller2.getBalance();
        Assert.assertEquals(45000, saldo);

        //spiller 1 lander p� Fleet, den koster 16.000
        laborcamp1.landOnField(spiller1);

        boolean bought = false;

        //er feltet ledigt? hvis ja, køb det
        if(spiller1.getInformation() == 0) {
                bought = spiller1.buyField(laborcamp1);
        }

        // fik han købt feltet? hvis ja, test om prisen er fratrukket
saldoen
```

```
                if(bought){
                        expected = 40000 - 6000;
                        actual = spiller1.getBalance();
                        Assert.assertEquals(expected, actual);
                }else{
                        //ellers burde beløbet ikke være fratrukket
                        Assert.assertEquals(40000,
spiller1.getBalance());
                }

                // test om spiller 2 har det rigtige startbeløb
                saldo = spiller2.getBalance();
                Assert.assertEquals(45000, saldo);

                // hvis spiller 1 havde købt feltet, så må spiller 2 betale
husleje
                if(bought){

                        //lad spiller 2 lande på feltet
                        laborcamp1.landOnField(spiller2);
                        boolean rented = false;


                        // hvis feltet ejes allerede, og vi kan leje det
```

```
                                    if(spiller2.getInformation() == -1){
                                        rented =
spiller2.rentField(laborcamp1);
                                    }
                                    //hvis det lykkedes at leje feltet, test om
lejepris er fratrukket saldo
                                    if(rented){
                                        expected = 45000 - 1000;
                                        actual = spiller2.getBalance();
                                        Assert.assertEquals(expected,
actual);
                                    }
                            }
                    }
}
```

Test af Refuge

```
package test;

import org.junit.*;

import entity.Field;
import entity.Player;
import entity.Refuge;
```

```
public class RefugeTest {

        private Player player;

        private Field refuge2;

        private Field refuge0;

        private Field refugeNeg200;

        private int startBeloeb;

        @Before

        public void setUp() throws Exception {

                this.player = new Player("Mufasa", 1);
                startBeloeb = this.player.getBalance();

                this.refuge2 = new Refuge( "simba +200", 200);

                this.refuge0 = new Refuge("simba 0", 0);
```

```
        this.refugeNeg200 = new Refuge("simba 200", -200);
    }
    @After
    public void tearDown() throws Exception {
    }
    @Test
    public void testEntities() {
        //The fields are unaltered
        Assert.assertNotNull(this.player);
        Assert.assertNotNull(this.refuge2);
        Assert.assertNotNull(this.refuge0);
```

```
        Assert.assertNotNull(this.refugeNeg200);

        Assert.assertTrue(this.refuge2 instanceof Refuge);

        Assert.assertTrue(this.refuge0 instanceof Refuge);

        Assert.assertTrue(this.refugeNeg200 instanceof Refuge);

}

@Test
public void testLandOnField200() {

        int expected = startBeloeb;

        int actual = this.player.getBalance();

        Assert.assertEquals(expected, actual);

        //Perform the action to be tested

        this.refuge2.landOnField(this.player);

        expected = startBeloeb + 200;
```

```
        actual = this.player.getBalance();

        Assert.assertEquals(expected, actual);

}

@Test

public void testLandOnField200Twice() {

        int expected = startBeloeb;

        //Perform the action to be tested

        this.refuge2.landOnField(this.player);

        this.refuge2.landOnField(this.player);

        expected = startBeloeb + 200 + 200;

        int actual = this.player.getBalance();

        Assert.assertEquals(expected, actual);
```

```
        }
}
```

Test af Tax

```
package test;

import org.junit.*;

import entity.Field;
import entity.Player;
import entity.Tax;

// Vi definerer en klasse TaxTest, hvor vi tester felt typen TAX

public class TaxTest {
        private Tax tax_1;
        private Tax tax_2;
        private Tax tax_3;

        private int startBeloeb;

        private Player spiller;
```

```
@Before
// setUp metoden bliver kaldt FÃ˜R hver test metode
public void setUp(){

        //opret 3 typer skat (beløb og procent )
        tax_1 = new Tax("Betal 2000", 2000);
        tax_2 = new Tax("Betal 1000 eller 10%", 1000, 10);
        tax_3 = new Tax("Betal 4000", 4000);

        //opret en ny spiller med 30.000kr i konto
        spiller = new Player("Mouse", 1);
        startBeloeb = spiller.getBalance();
}
@After
// setUp metoden bliver kaldt FÃ˜R hver test metode
public void tearDown(){
        spiller = new Player("Mouse", 1);
}
//I denne metode testes om beløbet trÃ¦kkes fra spillerens konto
@Test
public void testBelob2000(){
        int saldo = spiller.getBalance();
```

```
        Assert.assertEquals(saldo, startBeloeb);

        tax_1.landOnField(spiller);
        tax_1.pay();

        saldo = spiller.getBalance();
        Assert.assertEquals(saldo, startBeloeb - 2000);

}

@Test
//I denne metode testes om der trÃ¦kkes 10% fra konto
public void test10Procent(){

        int tiProcent = (10 * spiller.getBalance()) / 100;

        tax_2.landOnField(spiller);
        tax_2.payPercent();

        //udregn 10% af pengebeholdning
        int saldo = startBeloeb - tiProcent;
```

```
                        // test om saldoen vi har regnet os frem til er det samme
som spiller har i kontoen
                        Assert.assertTrue(saldo == spiller.getBalance());
        }

        public void testBelob4000(){
                        tax_3.landOnField(spiller);
                        tax_3.pay();
                        int expected = startBeloeb - 4000;

                        Assert.assertEquals(expected, spiller.getBalance());
        }
}
```

Test af Territory

```
package test;

import static org.junit.Assert.*;

import org.junit.Assert;
import org.junit.Before;
import org.junit.Test;

import entity.Player;
```

```
import entity.Territory;

public class TerritoryTest {
	private Territory territory1;
	private Territory territory2;

	private Player spiller1;
	private Player spiller2;

	@Before
	// setUp metoden bliver kaldt F�R hver test metode
	public void setUp(){
		territory1 = new Territory("Territory1, leje: 1000, eje: 6000", 1000, 6000);

		territory2 = new Territory("Territory2, leje: 5000, eje: 11000", 5000, 11000);

		spiller1 = new Player("Obama", 0);
		spiller1.addToBalance(10000);

		spiller2 = new Player("Clinton", 1);
		spiller2.addToBalance(15000);
```

```
}


// En spiller lander på et frit felt
@Test
public void testNewOwner() {

        // en simpel test for at sikre at startbeløbet er rigtigt
        int saldo = spiller1.getBalance();
        Assert.assertEquals(40000, saldo);

        //spiller 1 lander på Territory1, den koster 6.000
        territory1.landOnField(spiller1);

        // hvis getInformation returnere 0, så er feltet ledigt.
        if(spiller1.getInformation() == 0) {
                spiller1.buyField(territory1);
        }

        // udregn det forventede resultat
        int expected = 34000;

        //f saldo ud af spillerens konto
```

```
                int actual = spiller1.getBalance();

                //test om de tal er ens
                Assert.assertEquals(expected, actual);
        }

        //To spillere lander på samme felter efter hinanden
        // den første betaler for at eje, den næste betaler husleje
        @Test
        public void testAlreadyOwned() {

                // en simpel test for at sikre at startbeløbet er rigtigt
//              int saldo = spiller1.getBalance();
//              Assert.assertEquals(40000, saldo);

                //spiller 1 lander p Territory1, den koster 6.000
                territory1.landOnField(spiller1);

                boolean bought = false;
                if(spiller1.getInformation() == 0) {
                        bought = spiller1.buyField(territory1);
                }
```

```
                if(bought){
                        Assert.assertTrue(spiller1.getInformation() ==
1);
                }

                int saldo = spiller2.getBalance();
                Assert.assertEquals(45000, saldo);

                territory1.landOnField(spiller2);

                saldo = 45000 - 1000;

                if(bought){
                        Assert.assertEquals(saldo,
spiller2.getBalance());
                }




//              // udregn det forventede resultat
//              int expected = 40000 - 6000;
//
//              //f saldo ud af spillerens konto
//              int actual = spiller1.getBalance();
```

```
//
//                      //test om de tal er ens
//                      Assert.assertEquals(expected, actual);
//
//                      // Nu lander spiller 2 p? samme felt.. dvs han skal betale
husleje p? 1.000
//                      territory1.landOnField(spiller2);
//                      // udregn det forventede resultat
//                      expected = 15000 - 1000;
//
//                      //f? saldo ud af spillerens konto
//                      actual = spiller2.getBalance();
//                      //test om de tal er ens
//                      Assert.assertEquals(expected, actual);
            }

            // Vi tester om huslejlen passer med det forventede tal, som vi har
            // defineret h?jere oppe i setUp metoden
            public void testGetRent(){
                        int expected = 5000;
                        int actual = territory2.getRent();
```

```
        Assert.assertTrue(expected == actual);
    }
```

Test cases 2: tester om de terning er ens (positivt)

Precondition: ingen

Laver en test hvor vi henter værdien af den første terning i intervalen 1 til 6, og så den anden terning tilsvarende.
Derefter sammenlignes summen af de to terninger.

Postcondition: Terning ét og terning to er lig getDiceSum() i intervallet 2 til 12

Testscript:

```
package testcases;

import static org.junit.Assert.*;

import org.junit.Test;

import entity.Dice;
import entity.Dicebox;

public class DiceboxTest {


    @Test
    public void diceBoxTest() {


        Dicebox db = new Dicebox();

        for(int i = 0;i < 10000;i++){
            db.roll();
            assertTrue(db.d1.roll() >= 1 && db.d1.roll() <= 6);
            assertTrue(db.d2.roll() >= 1 && db.d2.roll() <= 6);
            assertTrue(db.getDiceSum()>=2 && db.getDiceSum()
<=12);
        }
    }

}
```

Test case 1: test balance (Positivt)

Precondition:
Opret spiller1 Obama
læg 10000 til spiller1 Obama konto

Test:
1) indsæt 10000 til spiller1 konto
2) sammenlign 40000 med spiller1 konto balance

Postcondition:

Meddelelse: Sandt / korrekt

Test Case 2: test NewOwner (Positiv)
Precondition:

Opret spiller1 Obama
Læg 10000 til spiller1 konto
Opret Territory1 felt
Land på Territory1

Test:

Tjek om Territory1 er ledigt
Køb Territory1 til 6000
Beregn konto balance
Sammenlign den forventede balance 34000 med konto balance

Postcondition:

Sandt, købet er gået igennem.

Bilag 1.5

Kravspecifikation fra CDIO2 gruppe 37:

Funktionelle krav:

1. Det er et spil mellem to personer
2. Spillerne skal have en pengebeholdning
3. Der er felter fra 2-12, som påvirker spillernes konti
4. Der skal være en tekst til hvert felt som udskrives *se feltbeskrivelse*
5. Spillerne starter med en beholdning på 1000
6. Den spiller, der opnår 3000 først, vinder
7. Hvis en spiller mister alle sine point, taber den

Ikke-funktionelle krav:

1. Skal kunne køres på databarerne på DTU (Windows) uden synderlige forsinkelser
2. Spillet skal let kunne oversættes til andre sprog
3. Det skal være let at skifte til andre terninger
4. Pengebeholdningen skal fungere i andre spil