

## CDIO delprojekt 3

---



**Udarbejdet af:**

**Stefan Hyltoft – s144872**

**Christian Laursen – s141355**

**Emdadollah Habibollah – s144882**

Gruppe nr: 52

Afleveringsfrist: *søndag den 30/11 2014 Kl. 20:00*

Antal sider: 38

2

## Contents

Introduktion.....	4
Kundens vision.....	5
Kravspecificering.....	5
Use Cases.....	6
Use case beskrivelse: Spil brætspil .....	6
Use case beskrivelse: Felter.....	7
Territory felt .....	7
Tax felt .....	8
Refuge felt .....	8
Labor camp felt.....	9
Fleet felt.....	9
Use case beskrivelse: Test .....	10
Use case diagram.....	11
Domænenmodel analyse.....	12
BCE-model .....	13
System sekvens diagram (blackbox system) .....	14
Design sekvensdiagram .....	15
Design klassesdiagram .....	17
Forklaringer .....	17
Arv.....	17
Abstract .....	19
Polymorfi .....	21
Test .....	21
Test cases.....	21
Test case 1: Test om spiller kan dø.....	21
Test case 2: Test om spiller kan eje et territory .....	21
Test case 3: Test at penge kan overføres mellem spillere.....	22
Test case 4: Test om en spiller kan vinde .....	22
Test case 5: Test om spillet slutter når en spiller har vundet.....	22
Screenshot af test kørsel .....	23
Dokumentation for GRASP .....	24
Controller.....	24

Creator .....	24
High Cohesion .....	25
Information Expert .....	26
Low Coupling .....	27
Konfigurationsstyring .....	27
Produktionsplatform .....	27
Import projekt i Eclipse.....	28
Konklusion .....	36
Bilag .....	37

## Introduktion

*"CDIO er et koncept for udvikling og kvalitetssikring af ingeniøruddannelse. I sin grundsubstans tager konceptet udgangspunkt i den professionelle ingeniørs virkelighed, og bogstaverne CDIO karakteriserer den løbebane – eller livscyklus – som ingeniørens problemløsning gennemgår. CDIO er forkortelsen af de engelske ord: "Conceive, design, implement, operate". Begrebsafklaring (conceive) er første fase, hvor problemet belyses og forstås, eller ideer skabes og behov afdækkes. Dernæst følger design-fasen, hvor der udtænkes en løsning til problemet. Under implementeringen realiseres løsningen på et demonstrationsniveau, og der skabes eller bygges et produkt eller en proces. Tilslut findes drift (operate), som repræsenterer den fase, hvor problemets løsning løbende bliver anvendt i praksis."*<sup>1</sup>

Denne rapport er udarbejdet af IT-økonomi- og softwareteknologi studerende på DTU i forbindelse med 3. del af vores CDIO eksamensprojekt på 1. semester.

Projektet går ud på at vi skulle forestille os, at vi var et spilfirma, som havde fået til opgave at udvikle et spil. Spillet er et matador-lignende brætspil som spilles på en Windows PC. Spilfirmaet har i de to tidligere projekter udviklet simplere udgaver af spillet. Først et spil som kun indeholdt to spillere, som slog med to terninger og samlede point i forhold til terningernes øjne. Derefter skulle det udbygges med tolv felter og en pengebeholdning. Felterne svarede til det antal øjne man slog med terningerne, og hvert felt øgede eller formindskede spillerens pengebeholdning.

I dette tredje CDIO projekt har spilfirmaet nu fået stillet til opgave at udbygge spillet yderligere. Der ønskes at man kan være 2-6 spillere, som hver starter med en pengebeholdning på 30.000. Der er også en

---

<sup>1</sup> Citeret fra DTU's CDIO handlingsplan 2007-2008 da CDIO principperne blev indført på DTU's diplomuddannelser [http://ctt.sitecore.dtu.dk/upload/dtu%20kommunikation/publikationer/dtu\\_cdio\\_final2%202.pdf](http://ctt.sitecore.dtu.dk/upload/dtu%20kommunikation/publikationer/dtu_cdio_final2%202.pdf)

spillerplade, således at når en spiller lander på et felt, så rykker han videre derfra på næste slag. Felterne består af fem forskellige typer, som har forskellige funktioner.

Rapporten er en oversigt med forklaring af de forskellige design elementer i projektet. Den indeholder projektets formål i form af kundens vision af det spil, som firmaet skal udvikle, samt figurer og diagrammer med forklaringer, som er fremstillet i forbindelse med udviklingen af projektet. En del af figurerne og diagrammerne er fra analyse delen af projektet, altså den del hvor vi udtænkte eller *analyserede* hvordan vi skulle udvikle spillet. Andre figurer og diagrammer viser hvordan spillet blev realiseret rent teknisk. Altså hvordan vores spillet er *designet* og hvordan det fungerer. Rapporten indeholder også et afsnit omkring hvordan vi har testet vores program for at sikre os, at spillet blev udviklet korrekt. Rapporten slutter af med en guide til hvordan spillet kompiles og startes, samt systemkrav og en konklusion på projektets forløb.

## Kundens vision

Fra vores oplæg til CDIO del 3:

*"Nu har vi terninger og spillere på plads, men felterne mangler stadig en del arbejde. I dette tredje spil ønsker vi derfor at forrige del bliver udbygget med forskellige typer af felter, samt en decideret spilleplade. Spillerne skal altså kunne lande på et felt og så fortsætte derfra på næste salg. Man går i ring på brættet. Der skal nu være 2-6 spillere.*

*Man starter med 30.000.*

*Spillet slutter når alle, på nær én spiller er bankerot.*

*I bilag 1 kan i se en oversigt over de felter vi ønsker, samt en beskrivelse af de forskellige typer."*<sup>2</sup>

Der ønskes altså, at man kan være 2-6 spillere, som hver starter med en pengebeholdning på 30.000. Der skal også udvikles en spillerplade, således at når en spiller lander på et felt, så rykker han videre derfra på næste slag. Felterne laves om, så de består af fem forskellige typer, som hver har forskellige funktioner. Felterne er *Territory*, *Refuge*, *Tax*, *Labor camp*, og *Fleet*. Hvad felterne betyder, hvor mange der skal være af hvert felt og hvor meget de koster at købe og leje er beskrevet i bilag 1.

## Kravspecificering

Gennem kundens vision, og spørgsmål til kunden, har vi udarbejdet følgende krav til programmet:

### Funktionelle krav:

---

<sup>2</sup> Citeret fra kundens vision i oplægget til CDIO 3.

- Spil mellem 2-6 personer.
- Skal kunne spilles på maskiner i DTU's databarer.
- 2 terninger - hver terning er en almindelig terning med øjne fra 1-6.
- Spillerne starter med en pengebeholdning på 30.000.
- Fungerer som et brætspil med 5 typer af felter.
- Hver type af felt har en særlig funktion.
- Når man lander på et felt skal spillet udskrive en tekst med feedback omkring hvad der er sket, eller hvilke muligheder spilleren har nu.
- Spillet er slut, når alle på nær én spiller er bankerot.
- Spillere kan ikke få negativ balance, da de således er bankerot og har tabt.
- En toString metode der udskriver alle felterne i GameBoard arrayet.

#### Ikke-funktionelle krav:

- Overhold GRASP principper (Creator, Controller, høj binding, information expert, lav kobling)
- En klasse Spiller, der indeholder en spillers attributter og funktioner.
- En klasse Konto, der beskriver Spillerens pengebeholdning.
- En klasse GameBoard der kan indeholde alle felterne i et array.
- Passende konstruktører.
- Passende get og set metoder.
- Passende toString metoder.
- Spilleren og hans pengebeholdning skal kunne bruges i andre spil.
- Skal versioneres i GIT, så kunden kan følge udviklingen.
- Ikke bemærkelsesværdige forsinkelser (max 33ms).
- Spillet skal let kunne oversættes til andre sprog.
- Det skal være let at skifte til andre terninger.

## Use Cases

Vi har i forbindelse med udviklingen af vores spil opstillet tre use cases. De har til formål at vise hvordan vores spil interagerer med brugeren i tre forskellige scenarier: *Spil brætspil*, *Felter*, og *Test*.

### Use case beskrivelse: Spil brætspil

Spil brætspil er vores centrale use case. Det er den som beskriver hvad der sker når man starter spillet op.

**Use case:** Throw dice

**ID:** Spil brætspil

**Beskrivelse:**

Spillerne kaster med 2 terninger og rykker brikken det antal felter frem, som øjnene viser. Forskellige ting kan forekomme afhængigt af hvilket felt man lander på. Spillet fortsætter indtil en af spillerne har vundet.

**Primære aktører:** Spillerne

**Betingelser forud:** Spillerne har startet spillet

**Main flow:**

- Systemet spørger hvor mange spillere der skal være med (2-6)
- Systemet spørger efter navn på spillerne.
  - **Hvis** Spilleren ikke skriver et navn, så får spilleren Player(1,2,osv.) som navn.
- Systemet vælger en tilfældig Startspiller
- **Mens** ingen af spillerne har vundet
  - Systemet fortæller hvilken Spillers tur det er
  - Systemet afventer input fra spiller("Press Enter to throw dices" f.eks.).
  - Systemet generer 2 tilfældige Terningekast.
  - Systemet viser i GUI hvad Spilleren har slået i Terningekastet.
  - Systemet rykker Spilleren hen til det felt som terningekastet repræsenterer. Alt efter hvilket felt man lander på, sker der noget forskelligt:
    - Lander på Territory felt – Se UC Territory.
    - Lander på Tax felt – Se UC Tax.
    - Lander på Refuge felt – Se UC Refuge.
    - Lander på Labor Camp felt – Se UC Labor Camp.
    - Lander på Fleet felt – Se UC Fleet.
- Udskriv hvilken Spiller har vundet

**Use case beskrivelse: Felter**

Felter er use cases som beskriver hvad der sker når du lander på et felt i spillet. Vi har lavet en use case der beskriver hvad der sker når man lander på hvert enkelt felt, da det der sker, afhænger af hvilken type felt det er.

**Territory felt**

**Use case:** Lander på Territory felt

**ID:** Territory

**Beskrivelse:**

Spilleren lander på et Territory felt. Er feltet ikke købt af en anden spiller endnu, har man mulighed for at købe territoriet, hvis man har råd. Hvis feltet ejes af en anden spiller skal man betale en afgift til den spiller.

**Primære aktører:** Spillerne.

**Betingelser forud:** At spilleren har slået med terningerne og er landet på et Territory felt.

**Main flow:**

1. Systemet tjekker om feltet er frit eller om det ejes af en anden spiller.
  - a. Er det frit spørger systemet om spilleren vil købe feltet.

- i. Hvis han vælger at købe feltet, så trækkes feltets pris fra spillerens pengebeholdning.
  - ii. Hvis spilleren vælger ikke at købe feltet, eller ikke har råd, så slutter turen.
- b. Ejes feltet af en anden spiller betales der en afgift til den spiller og turen slutter.

### Tax felt

**Use case:** Lander på Tax felt

**ID:** Tax

**Beskrivelse:**

Spilleren lander på et Tax felt og skal betale skat.

**Primære aktører:** Spillerne.

**Betingelser forud:** At spilleren har slået med terningerne og er landet på et Tax felt.

**Main flow:**

1. Systemet tjekker om feltet giver mulighed for at vælge mellem et fast beløb eller 10% af spillerens formue.
  - a. Hvis feltet er "**17. Caravan**" og spilleren har over 4000 på konto, giver det mulighed for at vælge, så kommer muligheden frem og spilleren skal vælge.
    - i. Vælger spilleren det faste beløb (**4000**), så trækkes beløbet fra spillerens pengebeholdning og turen slutter.
    - ii. Ellers betaler spilleren 10% af sin pengebeholdning og turen slutter.
  - b. Hvis feltet er "**16. Goldmine**" er det kun 2000 som skal betales, så trækkes beløbet fra spillerens pengebeholdning og turen slutter.

### Refuge felt

**Use case:** Lander på Refuge felt

**ID:** Refuge

**Beskrivelse:**

Spilleren lander på et Refuge felt og får udbetalt en bonus.

**Primære aktører:** Spillerne.

**Betingelser forud:** At spilleren har slået med terningerne og er landet på et Refuge felt.



**Main flow:**

1. Hvis feltet er **12. Walled city** får spilleren 5000 i pengebonus
2. Hvis feltet er **13. Monastery Refuge Receive 500** får spilleren 500 i pengebonus

**Labor camp felt**

**Use case:** Lander på Labor Camp felt

**ID:** Labor Camp

**Beskrivelse:**

Spilleren lander på et Labor Camp. Er feltet ikke købt af en anden spiller endnu, har man mulighed for at købe feltet, hvis man har råd. Hvis feltet ejes af en anden spiller skal man betale en afgift til den spiller.

**Primære aktører:** Spillerne.

**Betingelser forud:** At spilleren har slået med terningerne og er landet på et Labor Camp felt.

**Main flow:**

1. Systemet tjekker om feltet er frit eller om det ejes af en anden spiller.
  - a. Er det frit spørger systemet om spilleren vil købe feltet.
    - i. Hvis han vælger at købe feltet, så trækkes feltets pris fra spillerens pengebeholdning.
    - ii. Hvis spilleren vælger ikke at købe feltet, eller ikke har råd, så slutter turen.
  - b. Ejers feltet af en anden spiller betales der en afgift til den spiller. Det gøres ved at der slås med terningerne og antallet af øjne ganges med 100, og antallet af Labor Camps fra samme ejer. Herefter slutter turen.

**Fleet felt**

**Use case:** Lander på Fleet felt

**ID:** Fleet

**Beskrivelse:**

Spilleren lander på et Fleet felt. Er feltet ikke købt af en anden spiller endnu, har man mulighed for at købe feltet, hvis man har råd. Hvis feltet ejes af en anden spiller skal man betale en afgift til den spiller.

**Primære aktører:** Spillerne.

**Betingelser forud:** At spilleren har slået med terningerne og er landet på et Fleet felt.

**Main flow:**

1. Systemet tjekker om feltet er frit eller om det ejes af en anden spiller.
  - a. Er det frit spørger systemet om spilleren vil købe feltet.
    - i. Hvis han vælger at købe feltet, så trækkes feltets pris (**4000**) fra spillerens pengebeholdning.
    - ii. Hvis spilleren vælger ikke at købe feltet, eller ikke har råd, så slutter turen.
  - b. Ejers feltet af en anden spiller betales der en afgift til den spiller. Afgiften afhænger af hvor mange Fleet felter ejeren har i alt. Beløbene er som følger:
    - 1 Fleet: 500
    - 2 Fleets: 1000
    - 3 Fleets: 2000
    - 4 Fleets: 4000

### Use case beskrivelse: Test

Test use casen beskriver brugen af vores test klasse, som tester nogle af spillets forskellige funktioner.

**Use case:** Test brætspil

**ID:** Test

**Beskrivelse:**

En samling af test

**Primære aktører:** Testere

**Betingelser forud:** Testere har klassen med terninger, og klassen med ansvar for transaktioner til Konto.

**Main flow:**

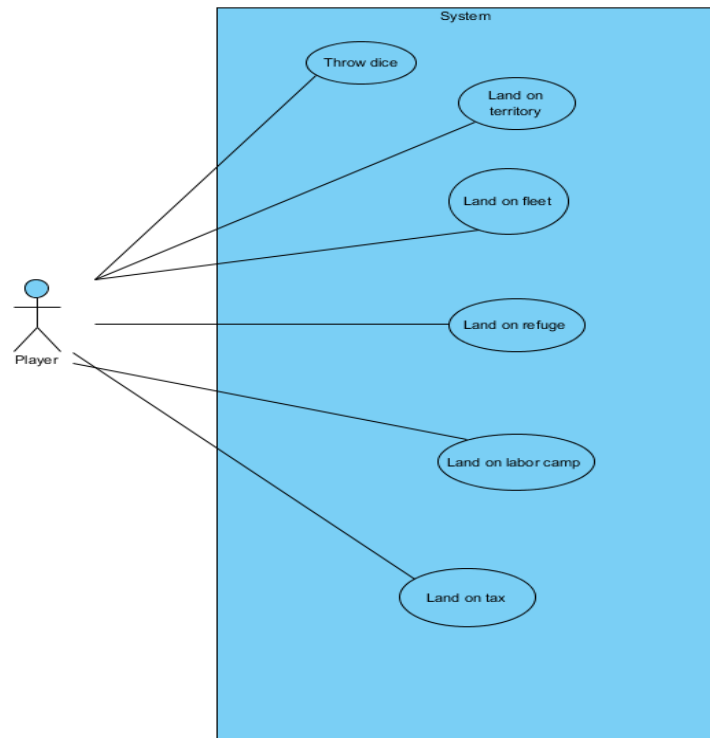
1. Tester om alle spillere får en tur til at kaste terninger
2. Test om spillere kan vinde
3. Test om spillere kan eje et territory
4. Test om spillere mister tur dør når de dør
5. Test overførsel af penge mellem spillere
6. Test om spillet stopper når én spiller har vundet

## Use case diagram

Vi har lavet to use case diagrammer som viser hvem der bruger de forskellige use cases.

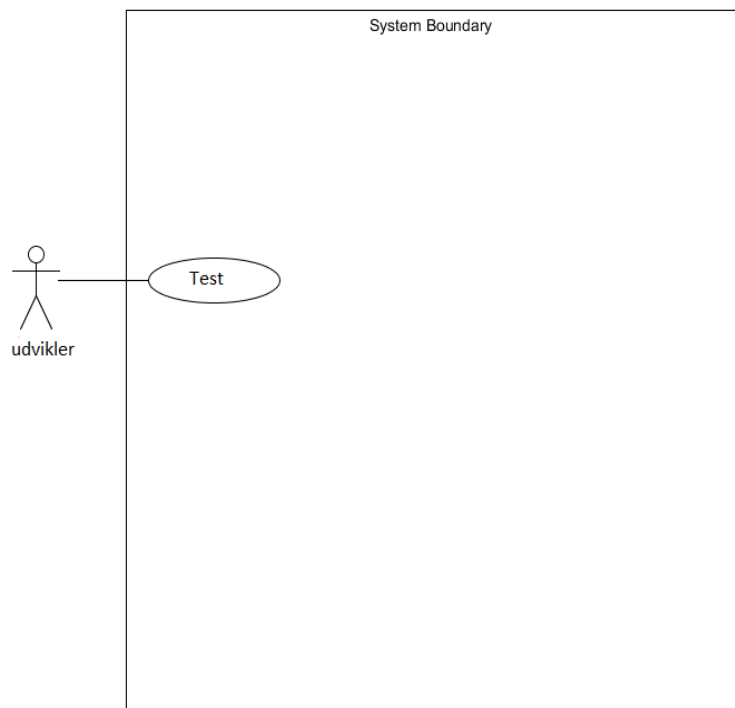
Use case diagram 1 viser at en spiller kan benytte de seks use cases:

*Throw dice*, og de use cases som beskriver hvad der sker når man lander på de forskellige typer af felter.



Use case diagram 1

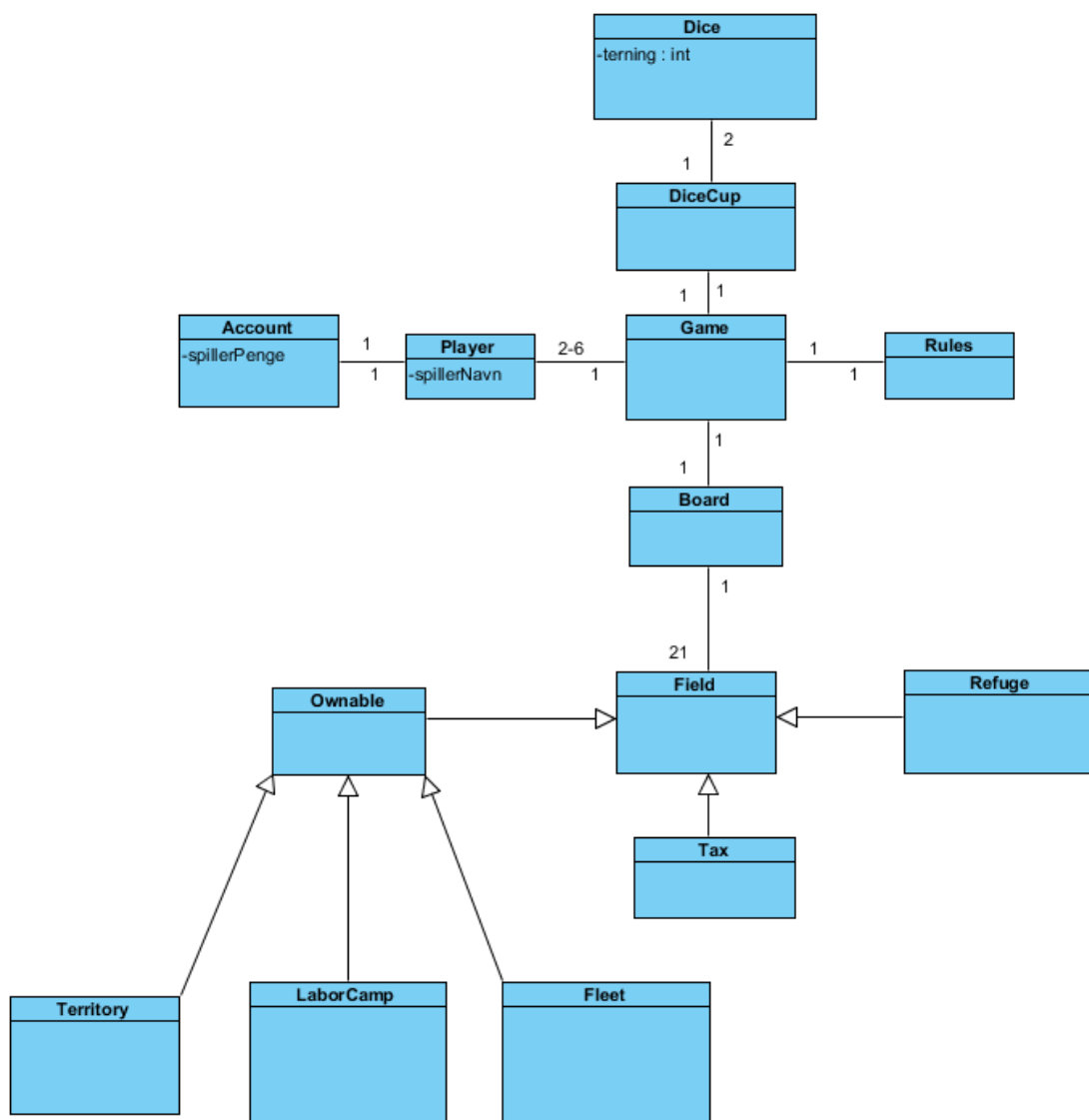
Use case diagram 2 viser blot at en udvikler kan teste vores spil.



Use case diagram 2

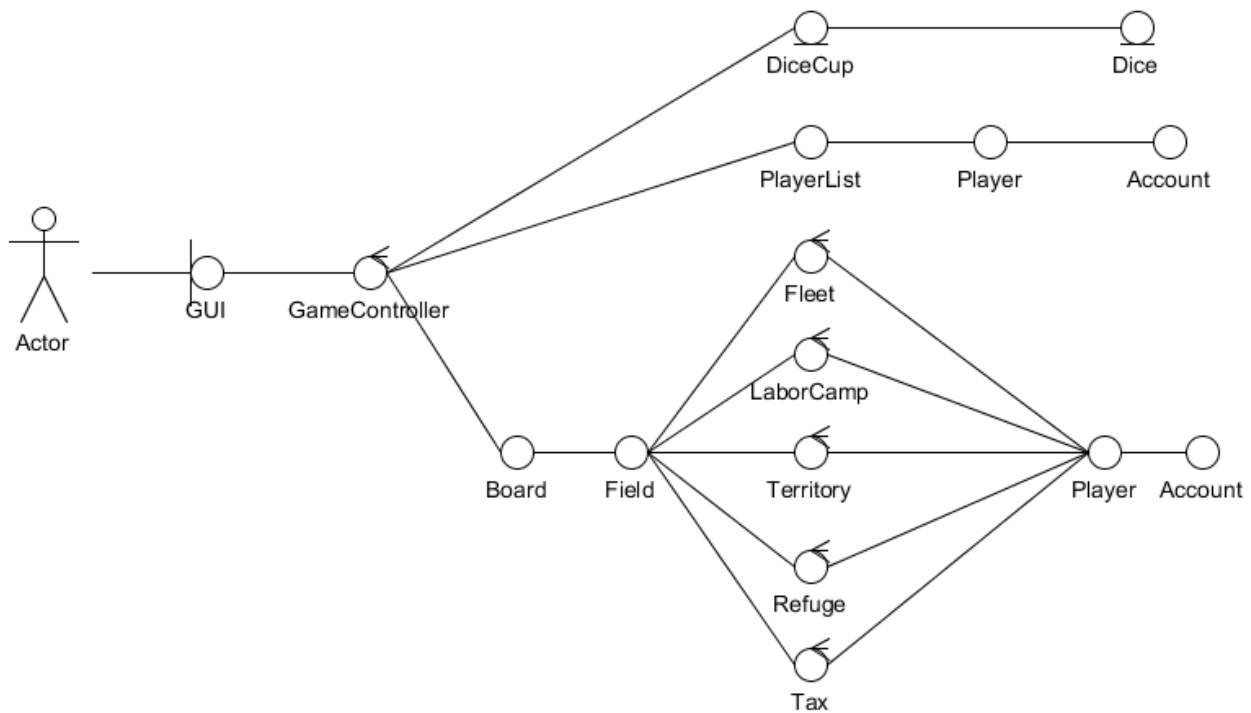
## Domænemodel analyse

Domænemodellen er modelleret efter objekter i den rigtige verden. Idéen er at vi har en spilmanager, *Game*, som holder styr på at spillet forløber som planlagt. Spilmanageren holder styr på antallet af spillere, raflebægeret med terninger, spillepladen og spillets regler. Spilmanageren registrerer hvor mange spillere der skal være med i spillet og hvor mange penge en spiller har på sin *Account*. Spilmanageren holder også styr på terningerne og hvilken spillers tur det er. Terningerne er i et raflebæger, således at begge terninger bliver slået samtidig. Spillepladen består af felter af fem forskellige typer: *Tax*, *Refuge*, *Territory*, *Fleet* og *Labor camp*. De tre sidstnævnte felter kan købes af spillerne. Spilmanageren fungerer samtidig også som en slags dommer, da det er ham der holder styr på spillets regler.



## BCE-model

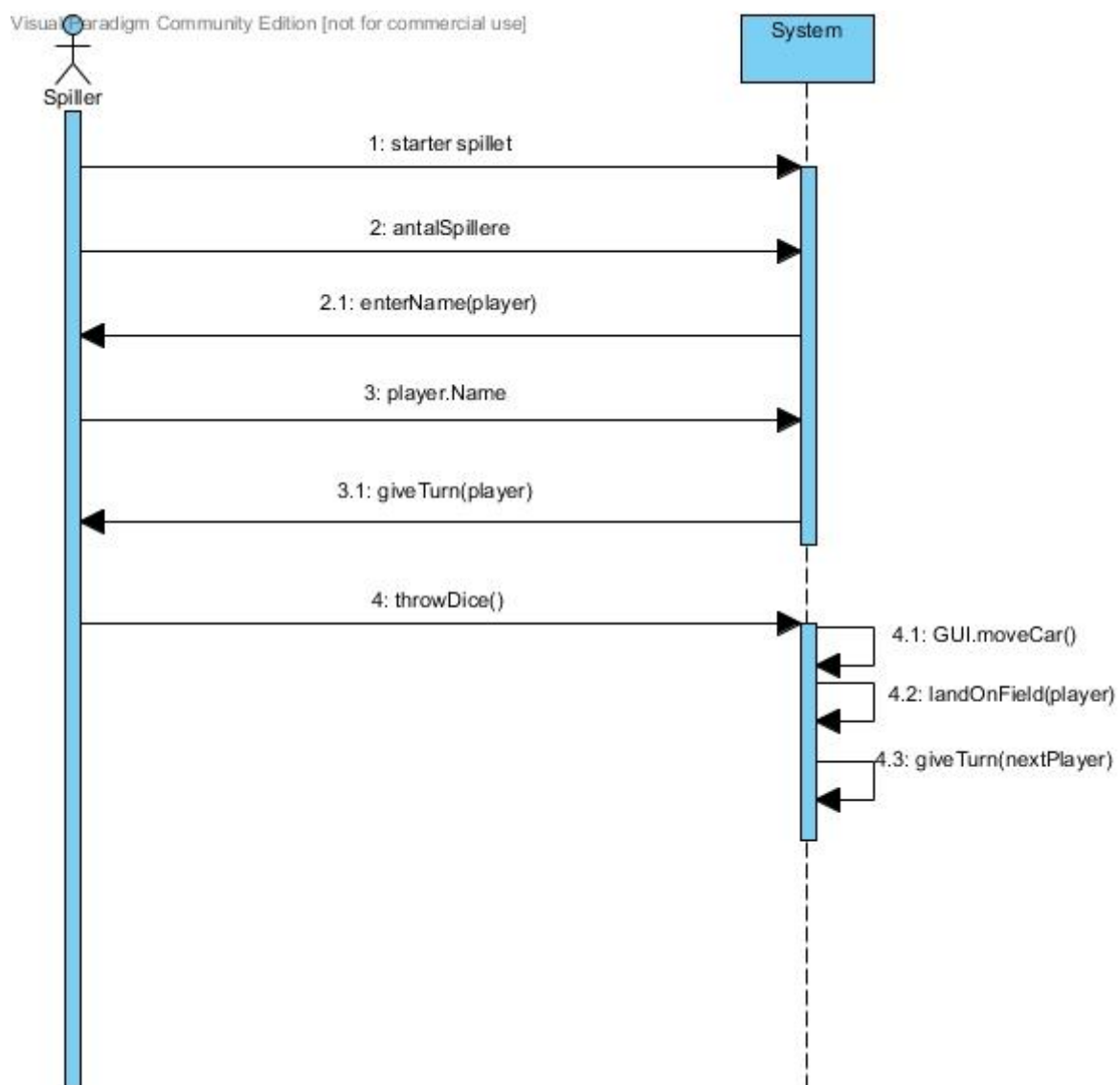
BCE-model 1 er baseret på vores use cases. I modellen kan det ses, at en aktør eller en spiller tilgår vores spil gennem GUI'en, som er vores boundary objekt. GUI'en styres af vores control objekt, *GameController*, som tilgår entity objekterne *Dice cup*, *PlayerList* og *Board*. *Boardet* tilgår vores *Field* entity, som indeholder fem control objekter. De fem control objekter styrer processen som er beskrevet i vores use cases for hver enkelt type af felt.



BCE-model

## System sekvens diagram (blackbox system)

System sekvens diagrammet (SSD) er baseret på vores throwDices use case. Diagrammet viser at en bruger, i det her tilfælde en spiller, starter spillet og vælger hvor mange spillere der skal være med. Systemet beder derefter brugeren om at skrive navnet på de deltagende spillere. Når navnene er indtastet finder systemet en tilfældig startspiller, hvorefter den spiller så kan kaste terningerne. Når terningerne er kastet, så rykker GUI'en den pågældende spillers bil det antal felter som øjnene viser på terningerne. Afhængigt af hvilken type felt der landes på sker der en handling svarende til vores use cases for de forskellige typer af felter. Til sidst gives turen videre til næste spiller.

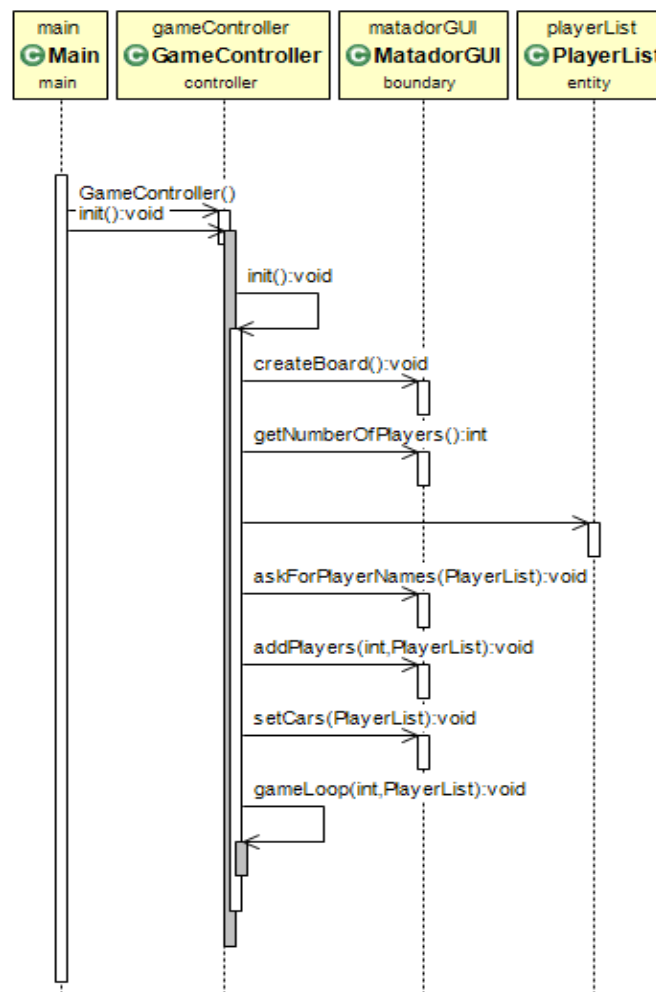


SSD (Black Box)

## Design sekvensdiagram

Vi har delt vores design sekvensdiagram op i en initialiserings del og en GameLoop del for at gøre det mere overskueligt.

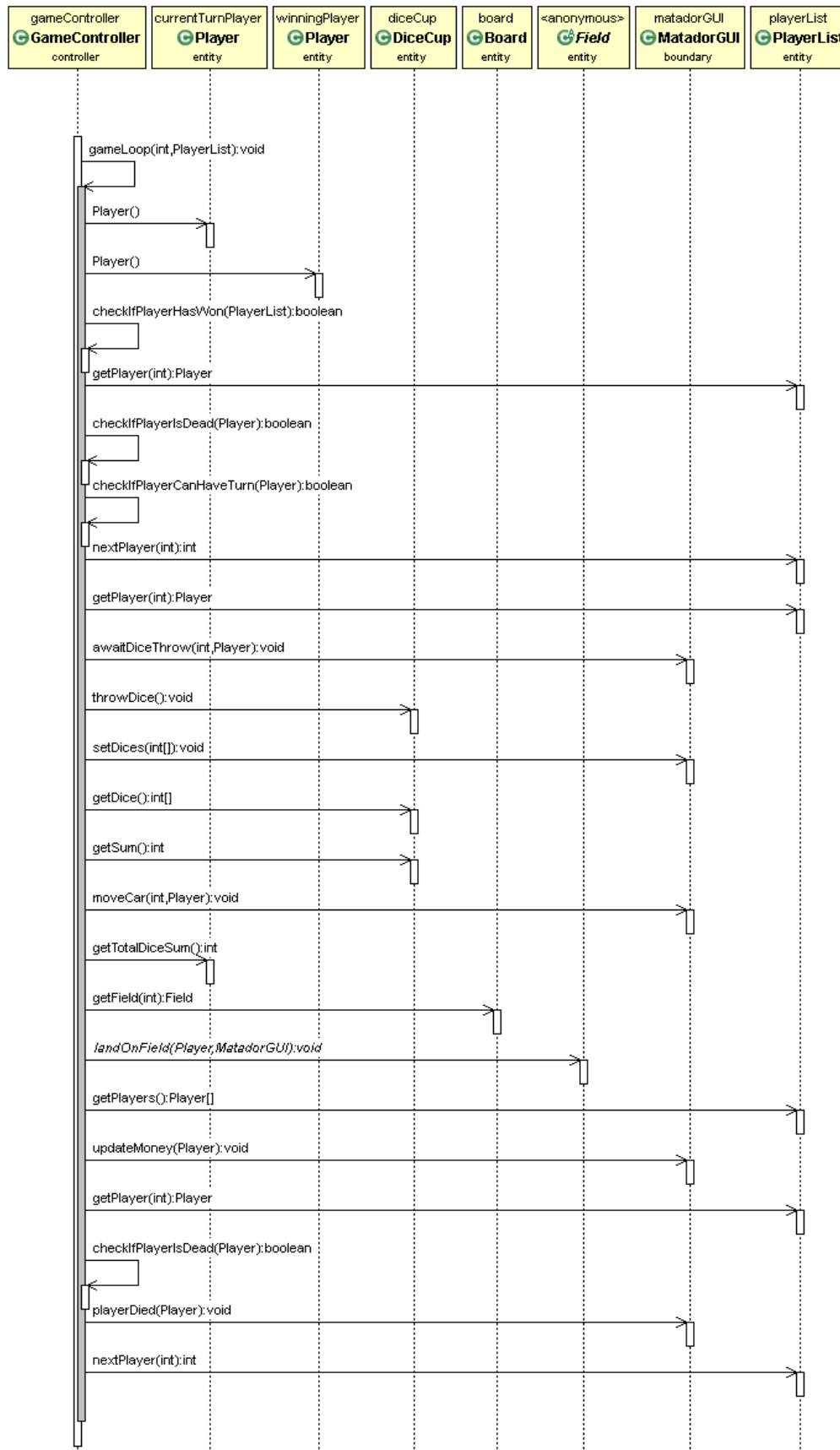
Design Sekvensdiagram: initialisering viser opstartsfasen af spillet. Når spillet startes bliver boardet oprettet, og man får antallet af spillere og deres navne. Spillerne tilføjes GUI'en med hver deres bil på pladen, hvorefter GameLoopet starter.



Design Sekvensdiagram: Initialisering

Design Sekvensdiagram: GameLoop viser sekvenserne i vores GameLoop. Først bestemmes det hvilken spiller der er den næste i PlayerListen og om spilleren har vundet, om spilleren er død eller om spilleren kan få en tur. Hvis spilleren kan få en tur, så afventer GUI'en at spilleren kaster terningerne. Når terningerne bliver kastet så ændrer terningerne på GUI'en sig svarende til det terningerne har slået i DiceCup klassen. Terningernes sum findes og spillerens bil rykkes i GUI'en. Der bliver så kaldt en LandOnField metode fra det felt spilleren er landet på<sup>3</sup>. Spillerens pengebeholdning bliver så opdateret og sendt til GUI'en og der checkes om spilleren er død. Hvis han døde sendes der en besked om at spilleren er død, ellers er det næste spillers tur. GameLoopet fortsætter til en spiller har vundet.

<sup>3</sup> Sekvenserne for hver type af felt er forskellige, så for overskuelighedens skyld er de ikke med i diagrammet.

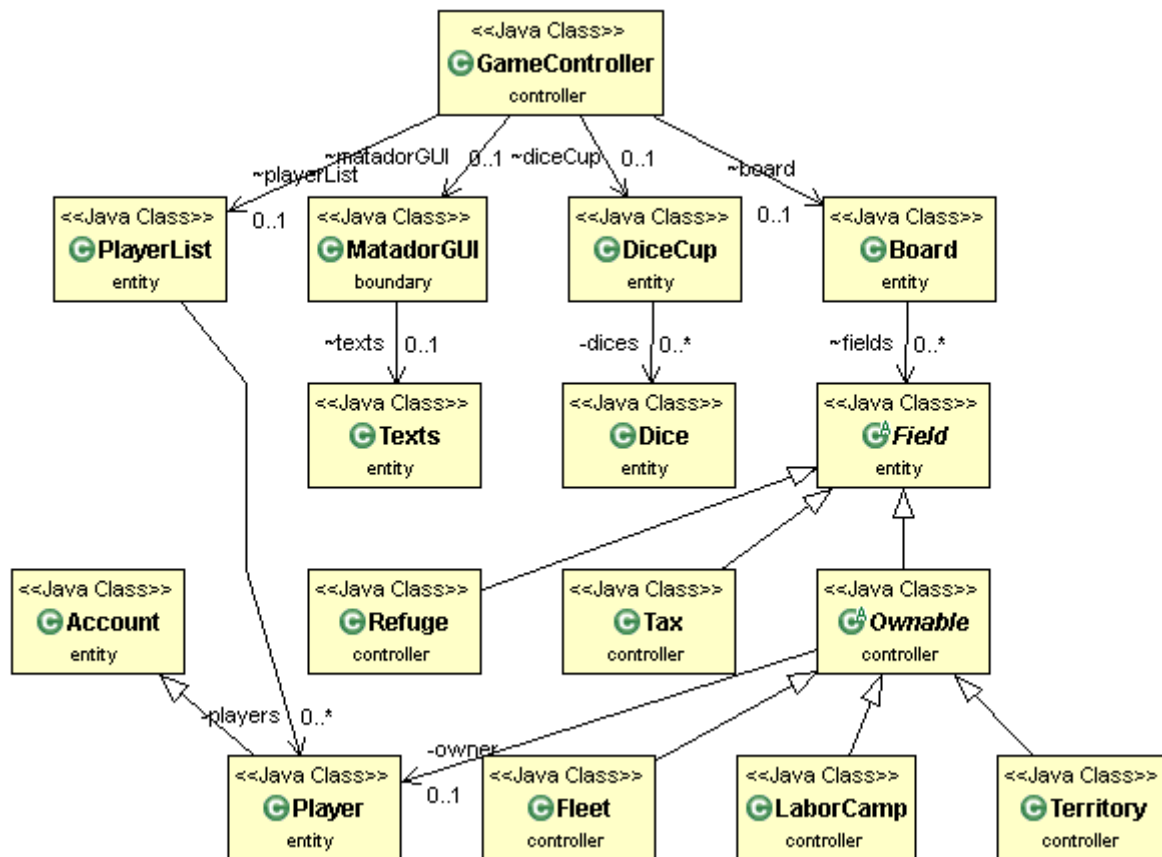


Design Sekvensdiagram: GameLoop



## Design klassediagram

Vi har delt vores design klassediagram op i to udgaver: En simpel og en udvidet. Den simple udgave viser blot klasserne og deres associationer til hinanden med multiplicitet, mens den udvidede også viser klassernes attributter, konstruktører og metoder.



Design Klassediagram: simpelt

## Forklaringer

### Arv

I objektorienteret programmering er "arv", når en klasse er baseret på en anden klasse. Det er en slags genbrug af software, hvor man udnytter lighederne mellem forskellige klasser, som vi ønsker at lave. Ligesom man kan udlede et objekt fra en klasse, kan man også udlede klasser fra klasser vha. arv. Det skaber et hieraki af klasser hvor attributterne og metoderne i én klasse bliver arvet af de udledte klasser. Herunder er eksempel omkring figurer med forskellige dimensioner.

Figur klassen er super/basis klassen i arve hierakiet. Det er ud fra denne klasse at de andre objekter "stammer fra". Basisklassen indeholder figur objekter med en farve.

```

1
2 public class Figur {
3     //
4     private String farve;
5     // Super}
6     public Figur(String farve) {
7         // } klassen i vores arve hieraki
8         this.farve = farve;
9         // Basis}
10    }
11    //
12    public String toString() {
13        // Overvridet toString() i class objekterne.
14        return "Farve: " + farve;
15    }
16    //
17 }

```

Figur klassen

Cirkel klassen er udledt af basisklassen. Efter navnet på den udledte klasse skriver man *extends* og så navnet på den klasse den skal arve fra. I det her tilfælde er det basisklassen Figur. Cirkel klassen arver fra figur klassen ved at den har en farve, men nu også en radius. Man "henter" farven fra figur klassen ved at skrive:

*Super(farve)*. Super kalder til klassen man arver fra.

*Super.toString()* henter en String fra Figur klassen. Her indeholder den kun farven.

```

1
2 public class Cirkel extends Figur {
3
4     private int r;
5
6     public Cirkel(String farve, int r) {
7         super(farve);
8         this.r = r;
9     }
10
11    public String toString(){
12        return super.toString() + " Radius " + r;
13    }
14 }
15

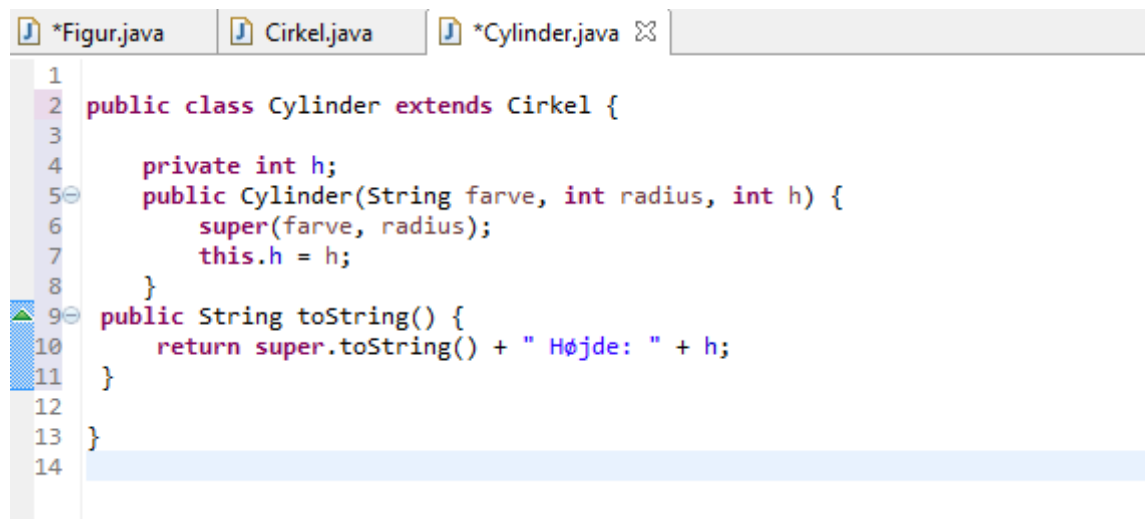
```

Cirkel klassen

Cylinder klassen *extender* nu Cirklen, ved at den udover at have en radius, også har en højde. Nu henter man både farven og radius fra Cirkel klassen ved at skrive:

*Super(farve, radius)*;

*Super.toString()* henter en string fra Cirkel klassen som indeholder farve og radius.



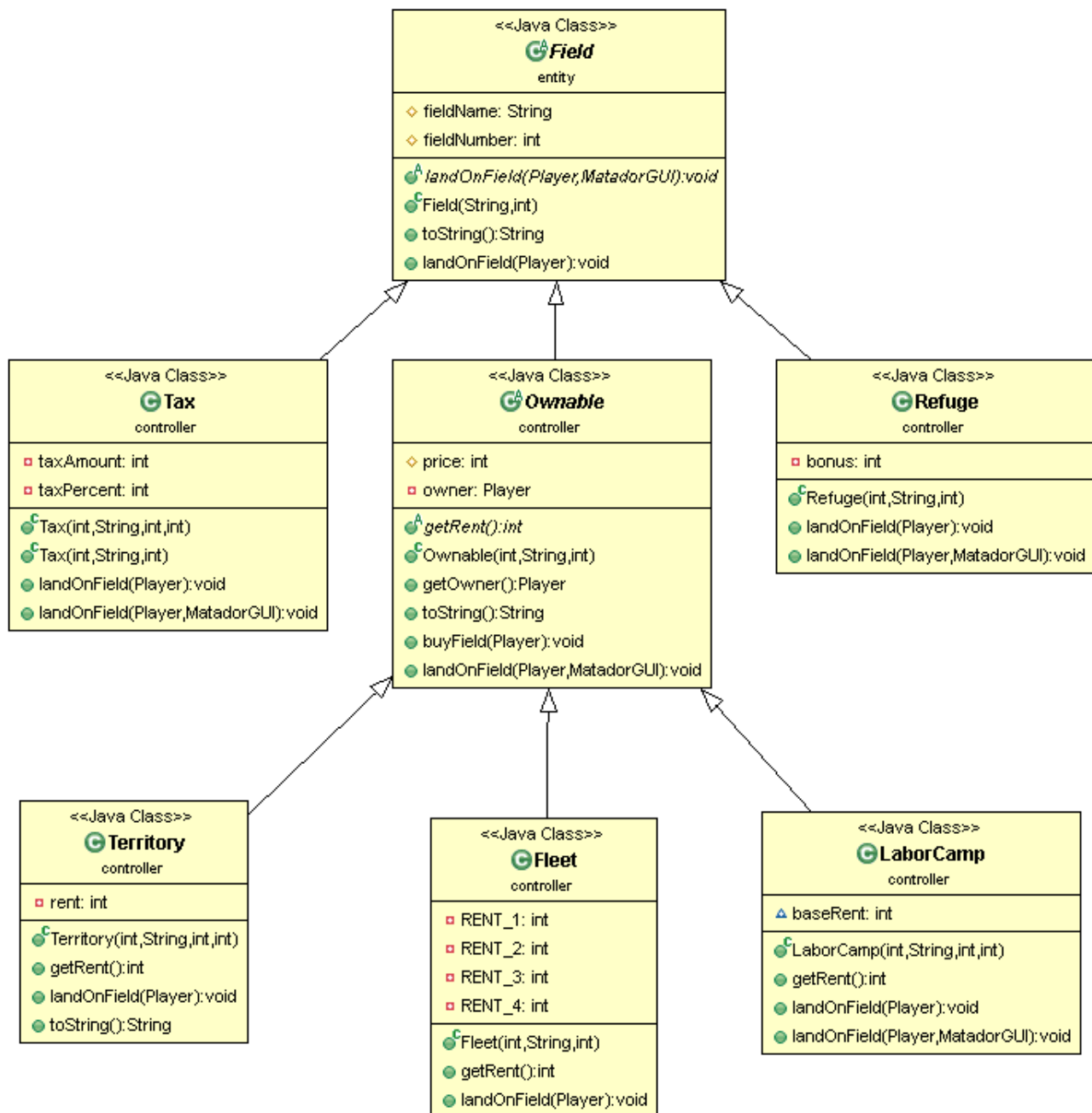
```
1
2 public class Cylinder extends Cirkel {
3
4     private int h;
5     public Cylinder(String farve, int radius, int h) {
6         super(farve, radius);
7         this.h = h;
8     }
9     public String toString() {
10         return super.toString() + " Højde: " + h;
11     }
12 }
13
14
```

Cylinder klassen

Fordelen ved arv er, at man kan genbruge metoder og attributter fra andre klasser, hvis klasserne kommer til at minde om hinanden. I eksemplet overfor vidste vi at Cylinder skulle indeholde attributter som radius og farve, så vi udledte den fra cirkel klassen. Dette sparer programmøren en hel del tid, og samtidig gør det projektet meget mere dynamisk.

## Abstract

I vores projekt har vi brugt *abstract* klasser, som udelukkende bruges som skabelon til andre klasser. En abstract klasse kan altså ikke instantieres til et objekt.



I vores spil er klasserne med kursiv skrift abstrakte, dvs. *Ownable* og *Field*. Det letter arbejdet for programmøren, fordi man ikke behøver at definere en metoder eller attributter fra super klasserne, som beskrevet i afsnittet omkring arv.

Når man bruger en abstrakt klasse som super klasse, undgår man at der bliver instansieret et objekt ud fra superklassen. Man kan lave en klasse som kun bliver brugt til nedarvning. I vores tilfælde vidste vi at der ville blive attributter som *price* og *owner* i de tre klasser *Territory*, *Fleet*, og *LaborCamp*. I stedet for at lave attributter og metoder separat i hver af disse 3 klasser (og risikere at skulle lave alle 3 om igen, hvis designet ændres), lavede vi blot en abstrakt klasse *Ownable*, som disse 3 klasser kan arve fra. *Ownable* klassen bliver altså kun brugt til arv, men vi har ikke lyst til at skulle lave nogen objekter ud fra den på noget tidspunkt. Der er *abstract* passende at bruge.

## Polymorfi

Polymorfi er endnu en videreudvikling af arv begrebet fra object orienteret programmering.

I vores klasse diagram kan vi se at der er en `landOnField()` metode, som alle felterne arver.

`landOnField` metoden bliver arvet videre fra den abstrakte `Field` klasse, men den kan ændre sig fra de forskellige typer af felter. Ifølge vores kravspecifikation skal der ske noget forskelligt alt efter, om man lander på `LaborCamp/Territory/Fleet/Tax`, og vha. polymorfi kan vi gøre dette.

## Test

Under udviklingen af programmet, er der prøvet at lave et *Test driven development*, dvs. at starte med at lave en test, for derefter at lave koden hertil. Det vil sige at man laver testen først, som fejler og derefter skriver kode, der gør at testen består.

De farligste test er egentlig dem der består i første træk, da man egentlig ikke kan være sikker på om man har rettet nogen fejl! Man kan risikere at en Junit test består, kun fordi man ikke bruger nogen `assertEquals` udtryk.

## Test cases

### Test case 1: Test om spiller kan dø

#### Precondition

1. Sæt spillerens balance til 0
2. Verificer at spillerens balance er 0
3. Verificer at spilleren ikke er død

#### Test

1. Send spiller objektet til spil controller tjekker og sætter spillers liv

#### Postcondition

1. Verificer at spiller nu er død

### Test case 2: Test om spiller kan eje et territory

#### Precondition

1. Sæt spillerens balance til 5000
2. Verificer at spillerbalancen er 5000

#### Test

1. Køb et felt med territory `buyField` metoden

#### Postcondition

1. Verificer at spilleren nu er ejer af pågældende territory objekt

### Test case 3: Test at penge kan overføres mellem spillere

#### Precondition

1. Sæt spiller A og spiller B's balance til 10000
2. Verificer at begge spilleres balance er 10000

#### Test

1. Overfør 5000 fra spiller A til spiller B

#### Postcondition

1. Verificer at spiller A nu har 5000 mindre, og spiller B 5000 mere

### Test case 4: Test om en spiller kan vinde

#### Precondition

1. Lav en spiller liste med et antal spillere
2. Sæt alle spillere til have en balance på 0, bortset fra én spiller med en balance på 1 (grænseværdi)
3. Verificer at alle spillers balance er som forventet, inkl én spiller med balance på 1
4. Få spil controlleren til at tjekke om alle spillere er døde

#### Test

1. Controlleren tjekker om en spiller har vundet

#### Postcondition

1. Controller afgiver at en spiller har vundet

### Test case 5: Test om spillet slutter når en spiller har vundet

#### Precondition

1. Opret en spiller liste med et variabelt antal spillere, og kun én vinder
2. Verificer at spillernes balance og deres livstatus er som forventet, ligesom i test case 4.

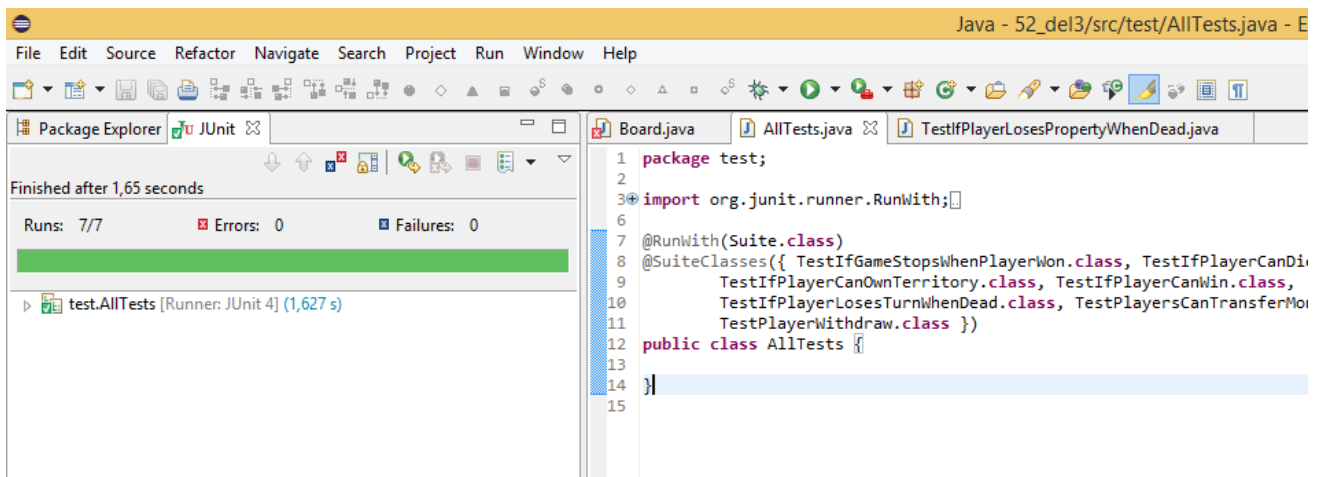
#### Test

1. Start spil loopet med spiller listen

#### Postcondition

1. Verificer at gameEnded variablen er sat til true, inde i controlleren.

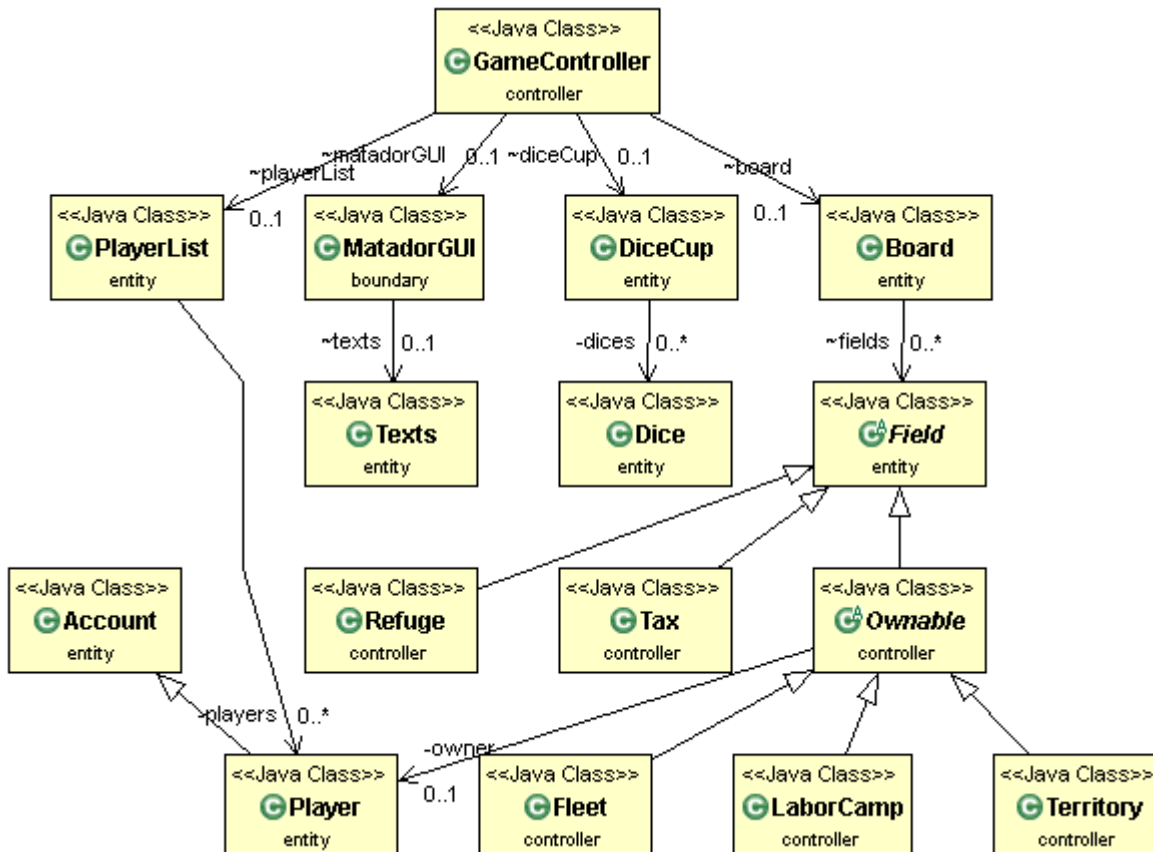
## Screenshot af test kørsel



## Dokumentation for GRASP

### Controller

Vi har 5 forskellige controllere, til 5 forskellige use cases. GameController uddelegerer opgaver for use cases som LandOnFleet og LandOnLaborCamp videre til de respektive controllers. GameControlleren håndterer samtidig også ThrowDices use casen.



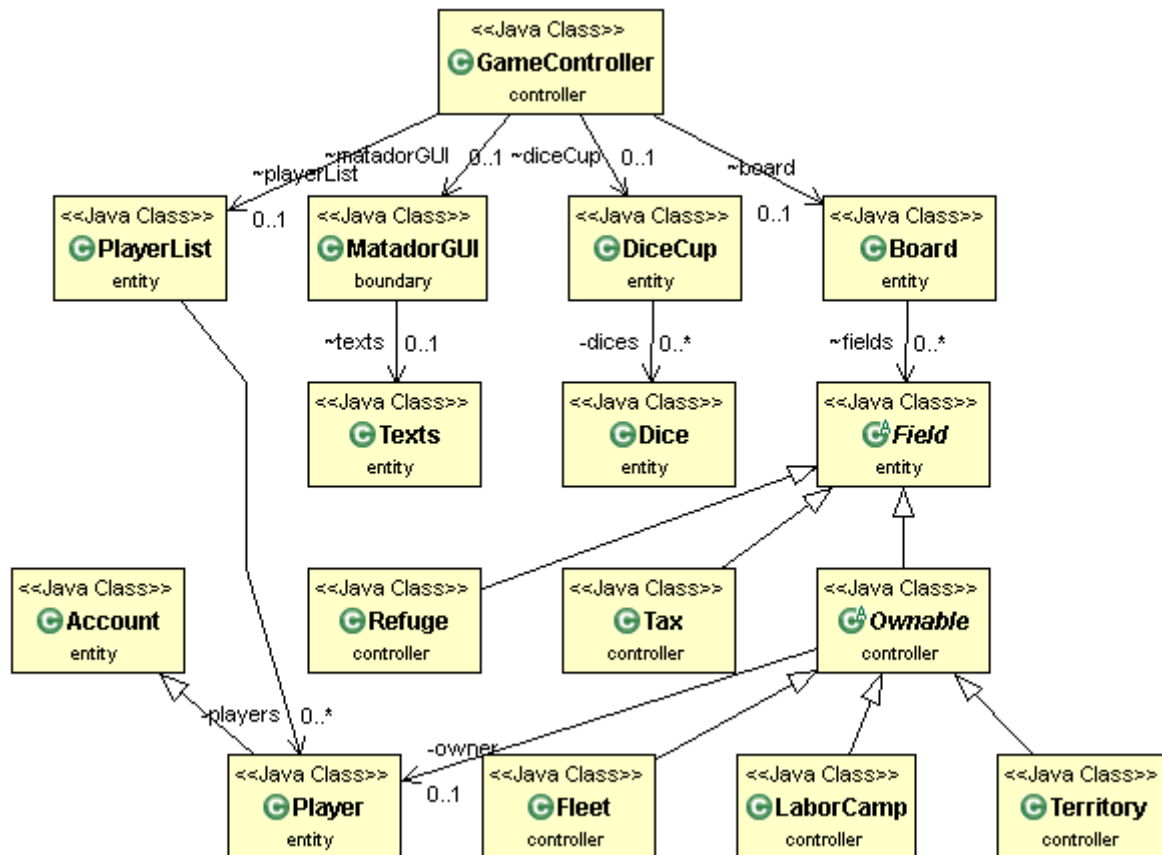
### Creator

Som et eksempel er *Board* creator af *Field* objekter, da Board indeholder/består af en masse field objekter. Board indeholder også den initialiserings data der skal bruges for field objekterne. Vores GameController er så også creator af Board klassen, da det er her klassen hovedsagligt bliver brugt. Det er altså den klasse der bruger objektet mest, som bliver designet som "Creator" af objektet.



## High Cohesion

Vi har forsøgt at have så høj cohesion som muligt ved at dele de forskellige ansvar op for klasserne.



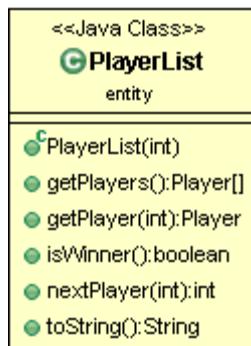
Der bliver ikke lavet direkte GUI kald i GameControllern, men der bliver kaldt metoder i MatadorGUI, som så håndterer det egentlige GUI arbejde. Der bliver heller ikke generet og kastet terninger inde i GameController, det bliver gjort med kald til DiceCup klassen.

## Information Expert

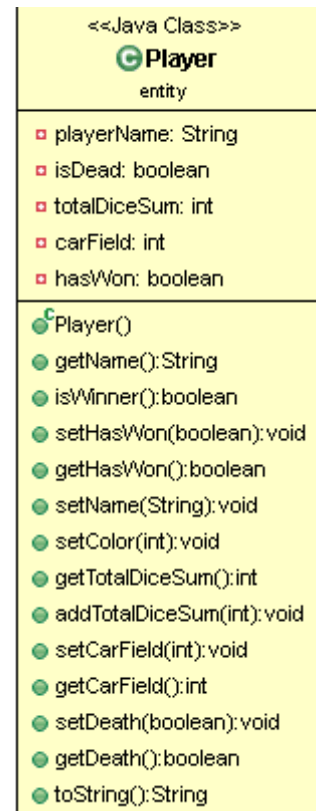
Information Expert princippet går ud på at man tildeler ansvar til den klasse, som har informationer relevante til det ansvar.

Vores Player klassen er information expert på hvert spiller objekt.

Da klassen indeholder information om en spillers navn, status og deres brik position, er det også her ansvaret for get og set metoder bliver lagt. Der bliver også gemt data for sidste terningekast, som bliver brugt i forbindelse med at sende spillerens brik i cirkler på brættet.

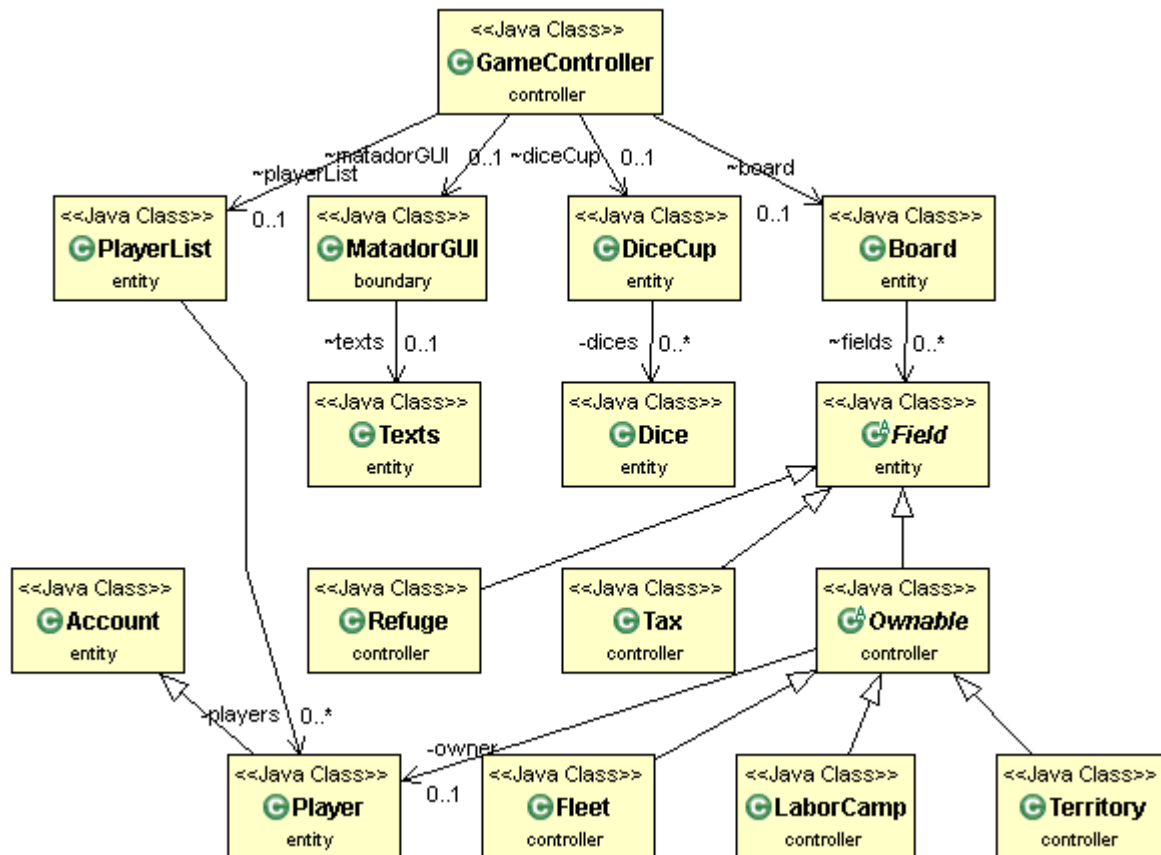


Da PlayerList indeholder 2-6 spiller objekter, følger man derfor Information Expert princippet, og placerer funktionen for at finde en vinder herinde.



## Low Coupling

Vi prøver at holde couplingen så lav som muligt i programmet.



For eksempel kender GameController klassen ikke til Player/Dice/Account/Texts klasserne, den har ikke brug for det. GameController kender til PlayerList klassen, og PlayerList klassen tilgår Player klassen.

Dette er for at mindske arbejdet hvis man ændrer i klasser. Ændringer i Dice klassen påvirker kun DiceCup klassen, og ændringer i Texts klassen påvirker kun MatadorGUI.

## Konfigurationsstyring

### Produktionsplatform

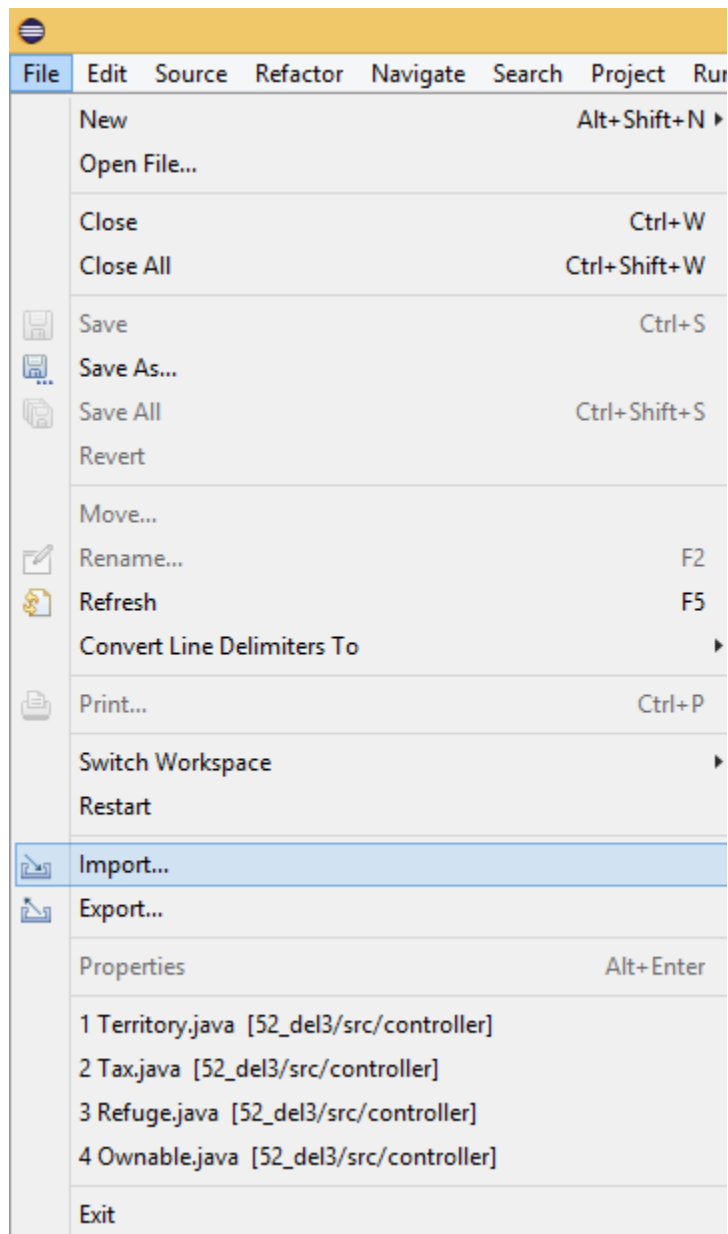
Windows 8.1.

Eclipse: Version: Luna Release (4.4.1)

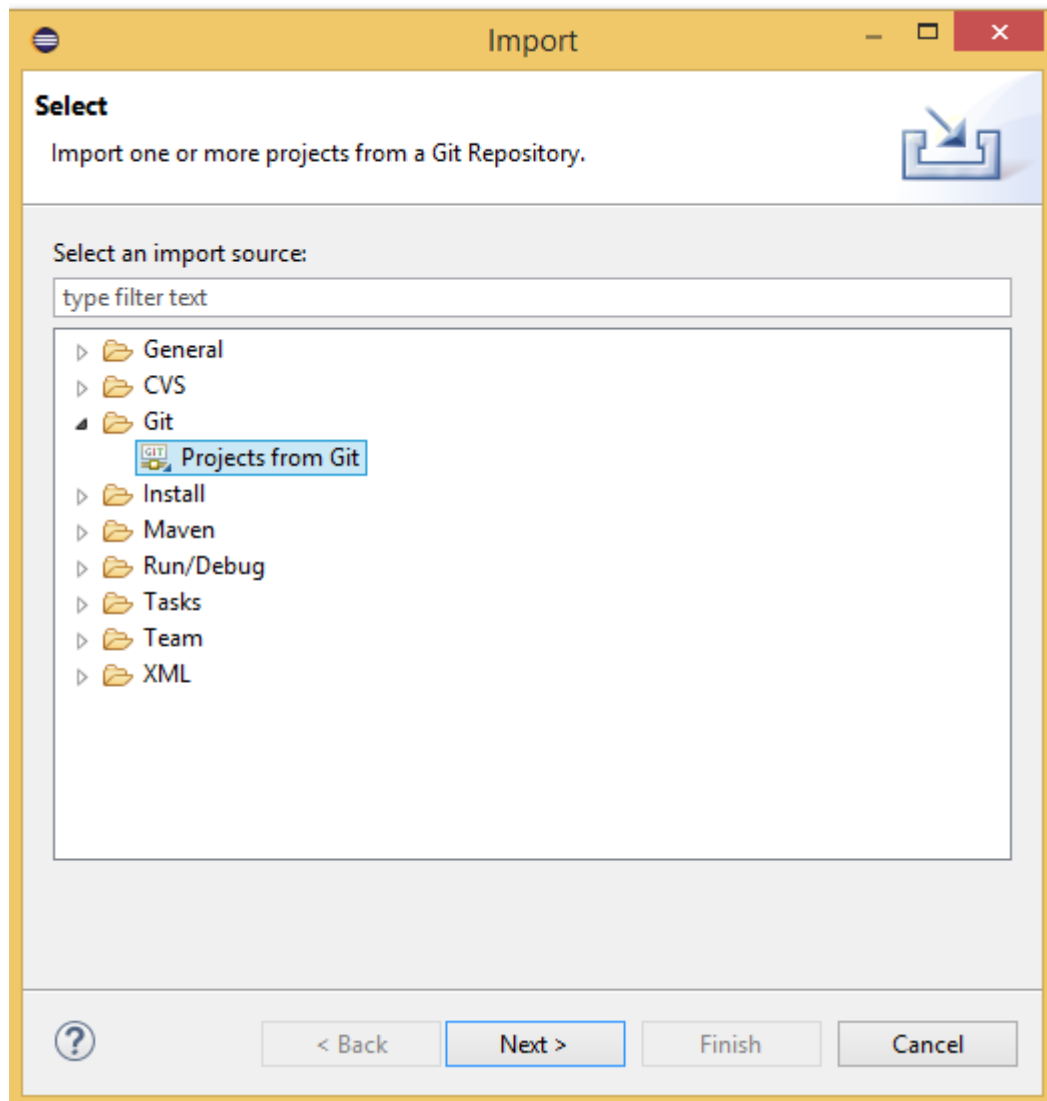
Java: JRE 1.8.0\_25 og JDK 1.8.0\_25

## Import projekt i Eclipse

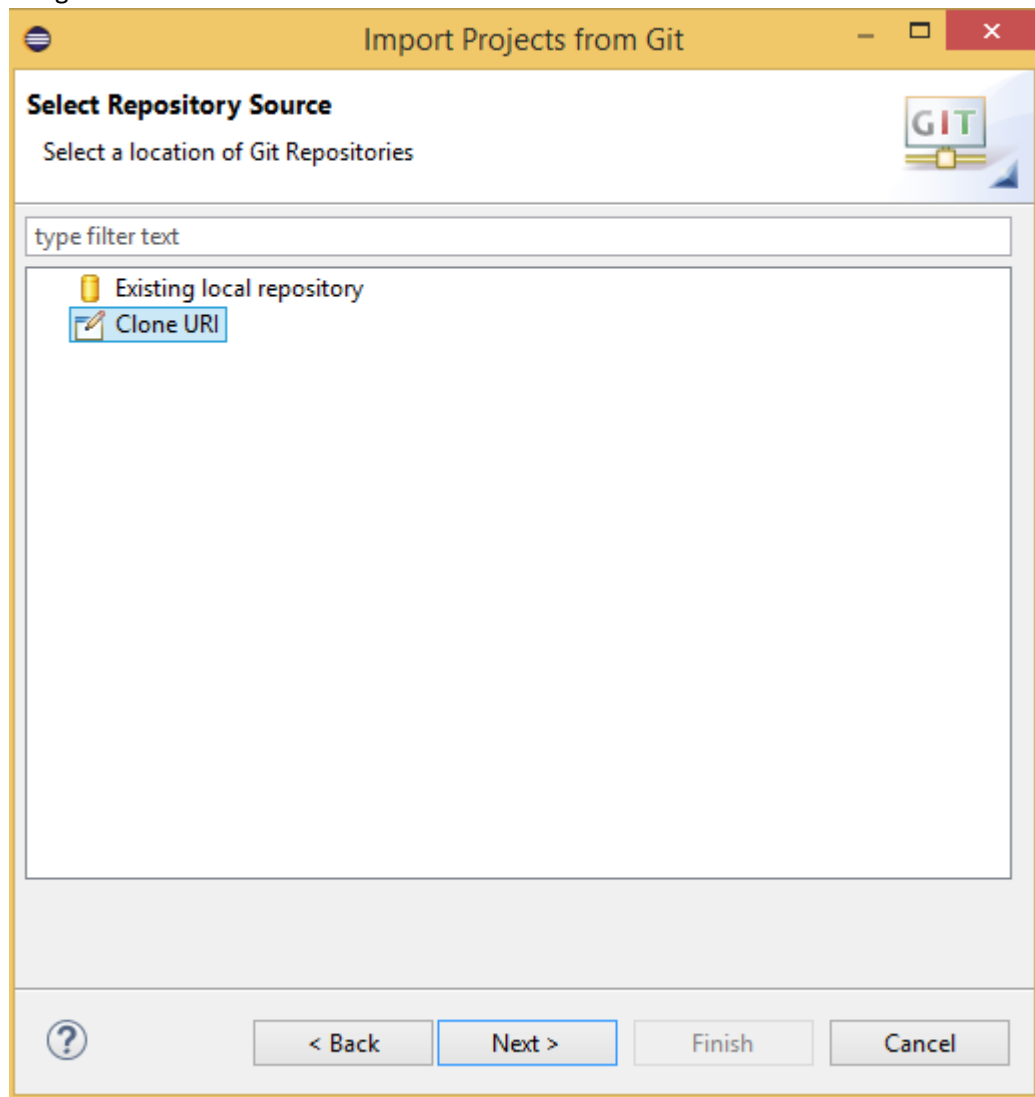
1. Vær sikker på at du har Eclipse og det krævede Java installeret.
2. Vælg Import under File i Eclipse



3. Vælg Git, og vælg så "Projects from Git"



## 4. Vælg "Clone URI"



5. Kopier adressen `https://github.com/Gruppe52/52_del3.git` i URI feltet. Eclipse udfylder resten af felterne. Username og password behøves ikke at blive udfyldt.

The screenshot shows the 'Import Projects from Git' dialog box in Eclipse. The title bar is yellow with the text 'Import Projects from Git'. The main area is titled 'Source Git Repository' and contains the instruction 'Enter the location of the source repository.' Below this, there are three sections: 'Location', 'Connection', and 'Authentication'. In the 'Location' section, the 'URI' field is filled with 'https://github.com/Gruppe52/52\_del3.git', the 'Host' field is 'github.com', and the 'Repository path' field is '/Gruppe52/52\_del3.git'. There is a 'Local File...' button next to the URI field. In the 'Connection' section, the 'Protocol' is set to 'https' and the 'Port' field is empty. In the 'Authentication' section, the 'User' and 'Password' fields are empty, and the 'Store in Secure Store' checkbox is unchecked. At the bottom, there are four buttons: '< Back', 'Next >', 'Finish', and 'Cancel'. The 'Next >' button is highlighted with a blue border.

**Source Git Repository**  
Enter the location of the source repository.

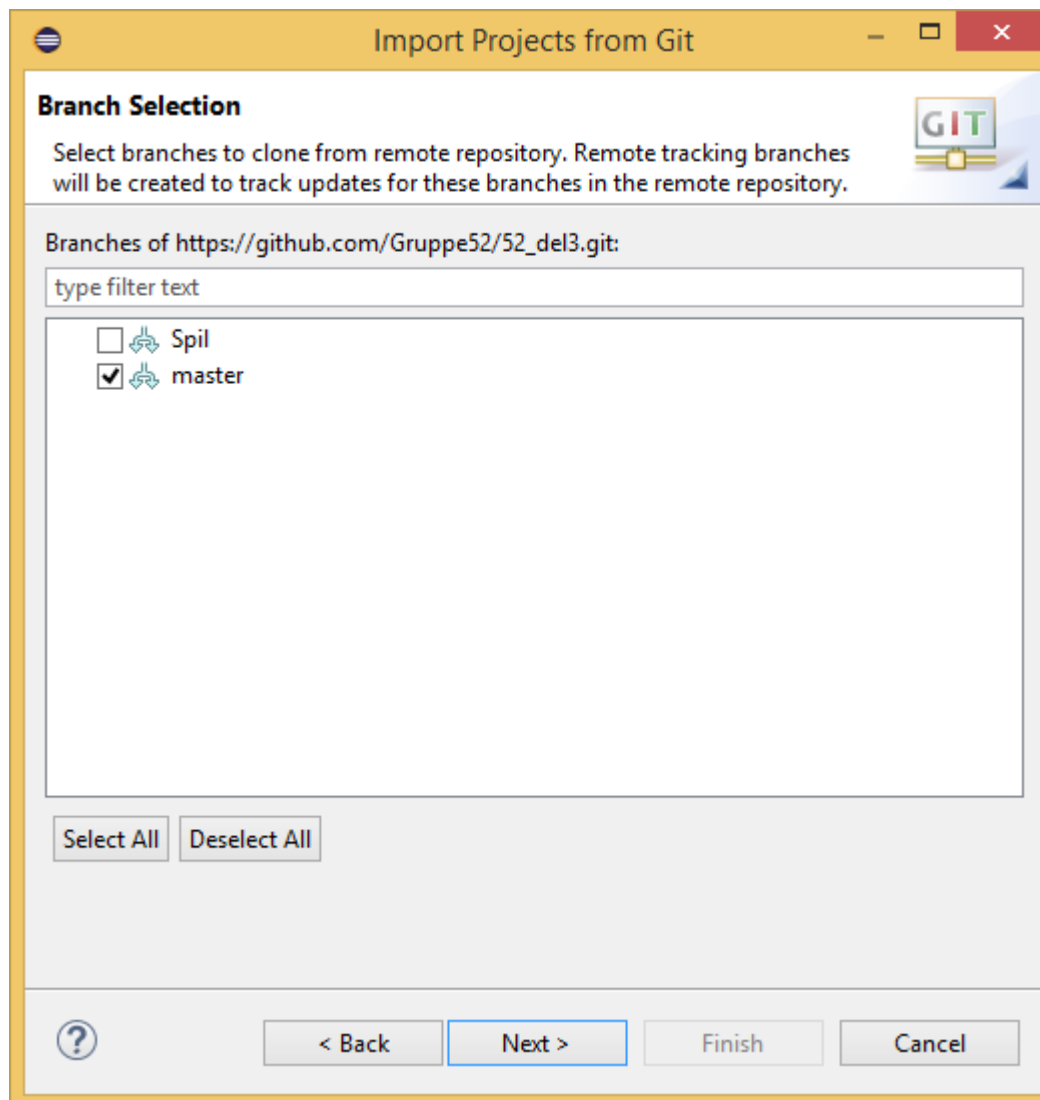
**Location**  
URI:  Local File...  
Host:   
Repository path:

**Connection**  
Protocol:    
Port:

**Authentication**  
User:   
Password:   
Store in Secure Store ☐

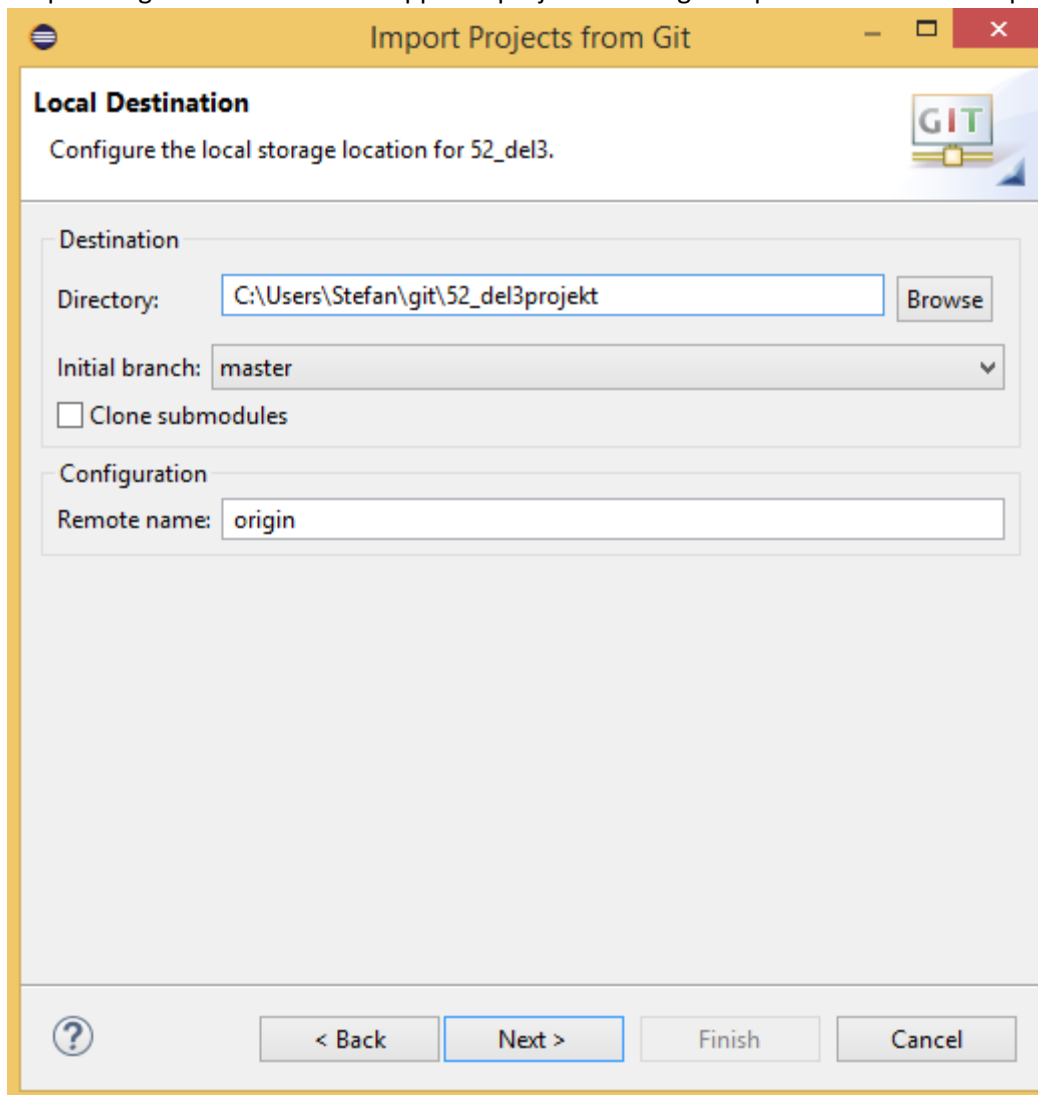
? < Back Next > Finish Cancel

6. Fravælg Spil branchen, det er en eksperimental branch. Hav kun master branchen valgt som på billedet.

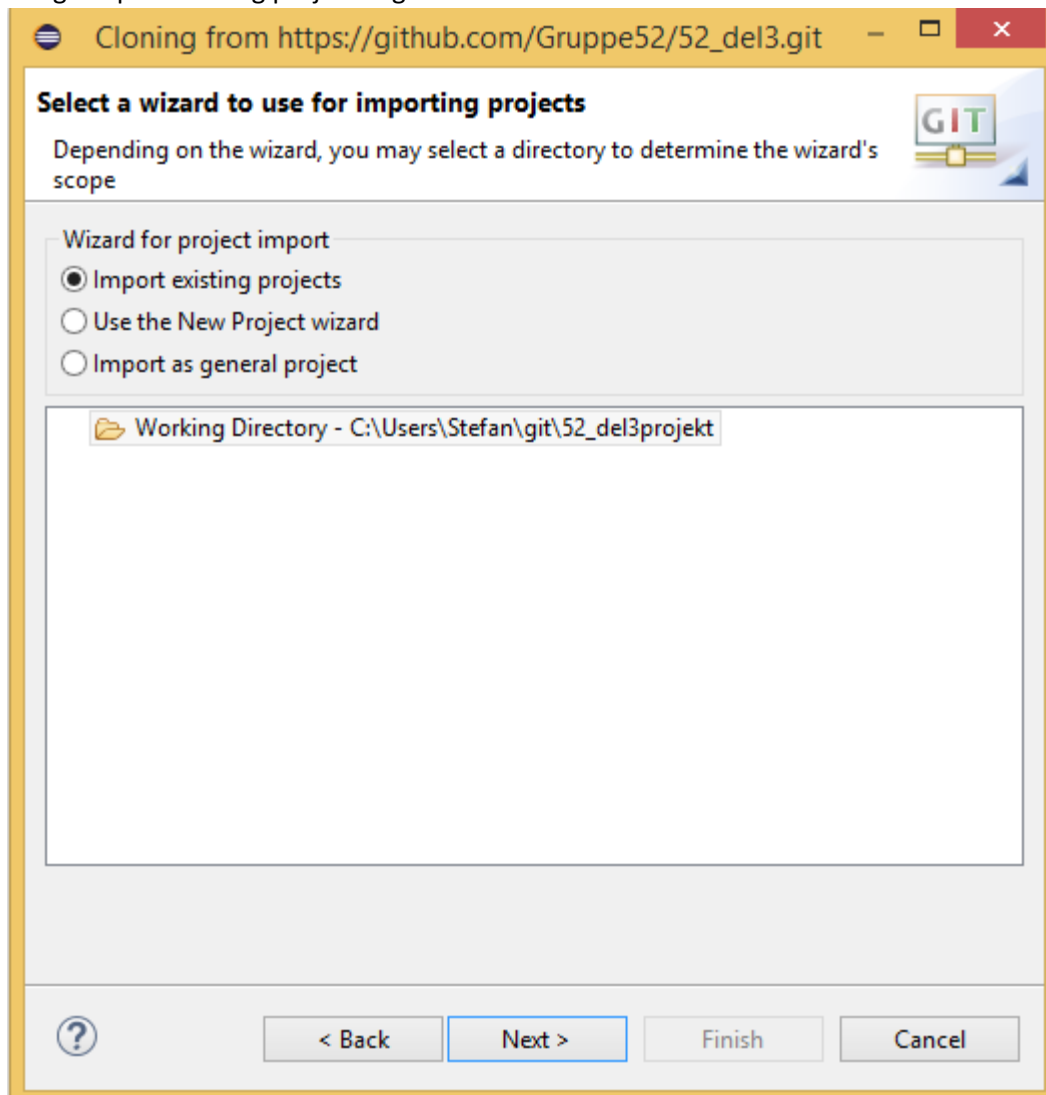




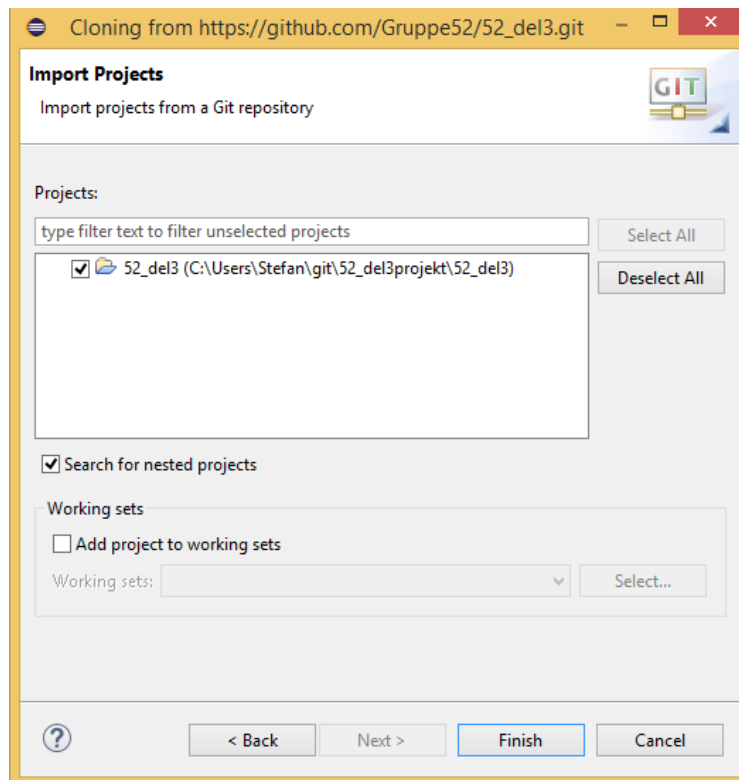
7. Eclipse vælger en destinationsmappe for projektet for dig. Klik på "Next" hvis det er passende.



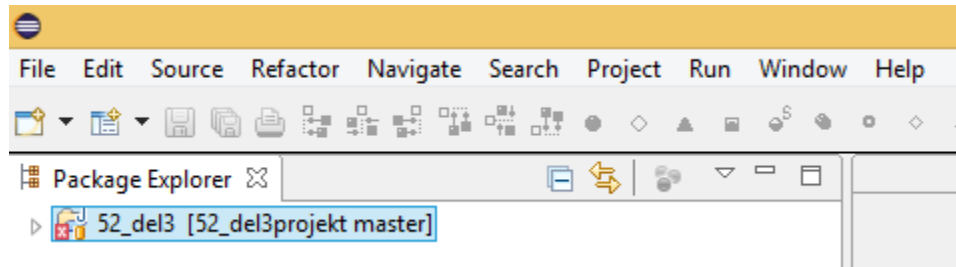
8. Vælg "Import existing project" og klik next.



## 9. Klik Finish



## 10. Projektet skulle nu gerne være fremme i "Package Explorer" i Eclipse.



## Konklusion

Gruppen startede med at være 6 gruppemedlemmer, men efter projekt nr. 2 er halvdelen af gruppemedlemmer faldet fra.

Det har stort set kun været ét gruppemedlem, som har stået for programmeringen i projekterne, inklusive dette sidste eksamensprojekt. Det har desværre ikke været muligt at få et samarbejde igang i kodedelen, og i dette sidste 3. projekt, har det altså været for stort et arbejde for én enkelt person.

Spillet har en hel del mangler og fejl, og gruppemedlemmet der har kodet har været nødsaget til at stoppe arbejdet, for at kunne arbejde videre på andre dele af projektet.

Der har generelt været en aktiv diskussion mellem gruppemedlemmerne i design-fasen af programmet, men ved programmeringsdelen er det kun ét medlem der deltager.

Selve basen for spillet er lagt, men der mangler en masse funktioner fra felt typerne. Der blevet lagt en hurtig skitse af de 22 felter fra oplægget i GUI'en. Spillerne får mulighed for at købe felter og blive ejere af dem, og lejen for felterne bliver også overført mellem spillerne. Spillernes brikker rykker også rundt i cirkler på brættet. Der mangler dog stadig en hel del arbejde, bl.a. virker det ikke helt efter hensigten, når en spiller dør. LaborCamp og Fleet felterne er heller ikke blevet udbygget.

Dog set i lyset af, at det kun har været en studerendes arbejde i kodedelen, er det gået godt. Projektet har bare været for stort for én person.

Gruppen har dog haft en god diskussion og samarbejde kørende under designfasen af programmerne, dvs. under UML delen.

Der har også været et godt samarbejde mellem 2 af gruppens medlemmer på rapportdelen for projektet.

## Bilag

### Bilag 1

#### 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

