Projektnavn: CDIO del 3
Gruppe: 35
Afleveringsfrist: 25/11-2016
Projektopgave efterår 2016

# CDIO del 3


Forfattere I gruppe 35:



s165158 Fabricius, Justin | s165151 Holm, Nikolaj | s165448 Enevoldsen, Jakob



s165152 Andreasson, Janus I s165156 Frederiksen, Mikkel I s155140 Tang, Tobias


Denne rapport er afleveret via Campusnet
Denne rapport indeholder 41 sider ekslusiv forside og bilag


Danmark Tekniske Universitet
DTU Compute


02313 Indledende programmering
02314 Udviklingsmetoder til IT-systemer
02315 Versionssyting og testmetoder

Vejledere: Henrik Bechmann, Daniel Kolditz Rubin-Grøn & Mads Nyborg

# 1 Abstract

This paper examines the software development process of a board game with multiple fields the players will move across. During development Unified Process was used to create iterations of the different classes, and how they work with one another. Additionally a preset goal was to design the source code following the GRASP principle.
The application has been tested by functional and acceptance test. After the test had been performed, Group 35 concluded that the project of developing the game had been successful. However there's always room for improvement and refactoring. Thus concluded this paper.

# 2 Timeregnskab

**Samlede regnskab**

| Navne | timer |
|---|---|
| Mikkel | 36 |
| Justin | 57,5 |
| Nikolaj | 31,5 |
| Tobias | 25 |
| Janus | 27 |
| Jakob | 15 |
| i alt | 177 |

# Indholdsfortegnelse

# 3 Indledning

Som en del af CDIO-projekterne har vi, i gruppe 35, i den tredje del fået stillet den opgave at udbygge forrige del med forskellige typer af felter og en decideret spilleplade. Vi har prøvet på at følge de metoder, som der er blevet undervist i i kursus 02313 til design af programmet, det vil altså sige, at adskillige versioner/iterationer af kravspecifikationen er udtrukket af kundens visioner og derefter analyseret videre til sekvens- og klassediagrammer. Netop disse diagrammer har hjulpet os videre til at kunne skrive og designe koden til spillet.

Til at versionere vores kode har vi gennem arbejdet benyttet os af GitHub for at danne overblik og for eventuelt også altid at kunne gå en version tilbage hvis vi finder en fejl, som tager for lang tid at rette. Desuden er JavaDoc blevet brugt i koden for at kunne dokumentere programmets funktioner og JUnit benyttet til at teste programmets metoder.

# 4 Problemformulering

IOOuterActive har fået endnu en opgave. Denne gang skal vi fremstille, teste og dokumentere et brætspil som vi selv designer fra bunden. Målet for os er at skabe et fungerende program med intuitiv kode ud fra GRASP, som overholder opgavens krav. Derudover skal vi have ordentlig dokumentation på vores projekt, hvilket vi vil gøre med forståelige og passende diagrammer. Sidst vil vi grundigt gennemgå og teste vores kode for at sikre os at koden er så solid som muligt.

# 5 Metoder

## 5.1 Unified Process

Som udgangspunkt gør vi brug af metoden Unified Process gennem projektet. Vi følger nødvendigvis ikke de fire faser stringent, men vi har fokus på at arbejde iterativt og dermed foretage ændringer undervejs.

## 5.2 Unified Modelling Language (UML)

Vi benytter os af UML til at beskrive strukturen og interageren i det spilsystem vi skal udvikle. Hertil benyttes både strukturelle- og adfærdsdiagrammer.

## 5.3 Unit test

Vi foretager unit tests af forskellige dele af programmet, for at undersøge, hvorvidt de enkelte dele af programmet kører som vi forventer. Til at teste benytter vi JUnit, og de forskellige tests fremgår senere i rapporten.

## 5.4 FURPS+

Til at udforme kavspecifikationen har vi benyttet os af FURPS+ modellen. Det ekstra led '+' er en udbyggelse af den oprindelige FURPS model og opfatter bl.a. en implementations-begrænsning. Denne er tilføjet i vores kravspecifikation. Det skal også bemærkes, at kravspecifikationen ikke kun er baseret på kundens vision i del 3, men da vi bygger videre på visse dele af vores spilsystem fra del 1 og 2 er flere krav fra de foregående spil inddraget i denne kravspecifikation.

# 6 Analyse

## 6.1 Kravspecifikation

Kravspecifikationen har gennemgået flere iterationer gennem forløbet. Følgende version er den endelige og dermed også hvad vi har designet ud fra.

## Functionality requirements:

1.1 Spilsystemet skal designes for 2-6 spillere, som slår på skift.

1.2 Der skal kastes med 2 terninger.

1.2.1 Terningen skal have 6 flader med værdierne 1-6.

1.3 Terningernes adderede sum angiver hvor mange felter spilleren skal gå frem.

1.4 Der skal nu være 21 felter, der indeholder den information kunden har angivet i feltlisten.

1.5 Hvis spilleren lander på et 'Territory' felt, skal man have muligheden for at købe feltet hvis det ikke er ejet af nogen anden spiller.

1.5.1: Hvis feltet er ejet af en anden spiller, skal man betale leje til ejeren af feltet.

1.5.2: Lejen beregnes på baggrund af 'Feltlisten' (bilag 8) og 'Typer af felter' (bilag 8 )

1.6 Landes der på et 'Refuge' felt, får man udbetalt en bonus dokumenteret i 8.

1.7 Landes der på et 'Tax' felt, skal spilleren betale en afgift.

1.7.1: Lander spilleren på feltet Goldmine, skal spilleren have fratrukket 2000.

1.7.2: Lander spilleren på feltet Caravan, fratrækkes 4000 fra spillerens konto eller 10 % af spillerens formue efter eget valg.

1.8 Landes der på et 'Labor camp' felt skal spilleren have mulighed for at købe feltet.

1.8.1 Hvis feltet er ejet af en anden spiller, skal der betales en afgift til denne beskrevet i 8.

1.8.2 Hvis ejeren af feltet ejer begge Labor camps felter skal afgiften fordobles.

1.9 Landes der på et 'Fleet' felt skal spilleren have mulighed for at købe feltet.

1.9.1 Hvis feltet er ejet af en anden spiller, skal der betales en afgift til denne. Afgiften skal bestemmes ved antallet af fleets ejeren har: de forskellige beløb fremgår af bilag 8.

1.10 Spillerne skal starte med en pengebeholdning på 30.000.

1.11 Spillet skal slutte når alle på nær én spiller er gået bankerot.

1.11.1: Man skal gå bankerot når spillerens pengebeholdning går i 0.

1.11.2: Pengebeholdningen skal ikke kunne være negativ.

## Usability requirements:

2.1: Der skal udskrives en tekst i GUI, på baggrund af det felt man lander på.

2.2: Spillet skal have en grafisk brugerflade (GUI).

2.2.1: Spillet skal indeholde et sæt grafiske terninger.

2.3.2: Spillet skal indeholde en spilleplade med 21 felter.

2.4.3: Spillepladen skal indeholde de respektive feltnavne fra feltlisten.

2.5.3: Spillet skal indeholde en simpel manøvre til at ryste og slå med raflebægeret.

2.3: Spillet skal fortælle hvilken spillers tur det er.

## Reliability requirements:

3.1: Raflebægret skal være pålideligt. Det vil sige, at det skal kunne bruges mindst 1000 gange uden at crashe.

## Performance requirements:

4.1 Første kast skal kunne afvikles inden for 600 millisekunder.

4.1.1: De efterfølgende kast skal afvikles med en maksimal forsinkelse på 333 millisekunder.

## Supportability requirements:

5.1: Det skal være let at oversætte og ændre teksten som udskrives.

5.2: Det skal være let at udskifte spillets terninger, med en anden type terning.

## Implementation:

6.1 Spilsystemet skal virke på DTU's Windows-computere i Databaren i bygning 303A.

## 6.2 Udsagnsordsanalyse

Med udgangspunkt i kundens vision, laver vi en udsagnsordsanalyse. Ved at lave denne analyse får vi et overblik over, de handlinger som skal kunne udføres i spillet.

| |
|---|
| At slå med terningerne |
| At lande på et felt |
| At fortsætte fra opnået felt |
| At købe ejendom |
| At betale leje |
| At modtage leje |
| At gå bankerot |

*Tabel 1: Udsagnsordsanalyse*

## 6.3 Use Case modellering

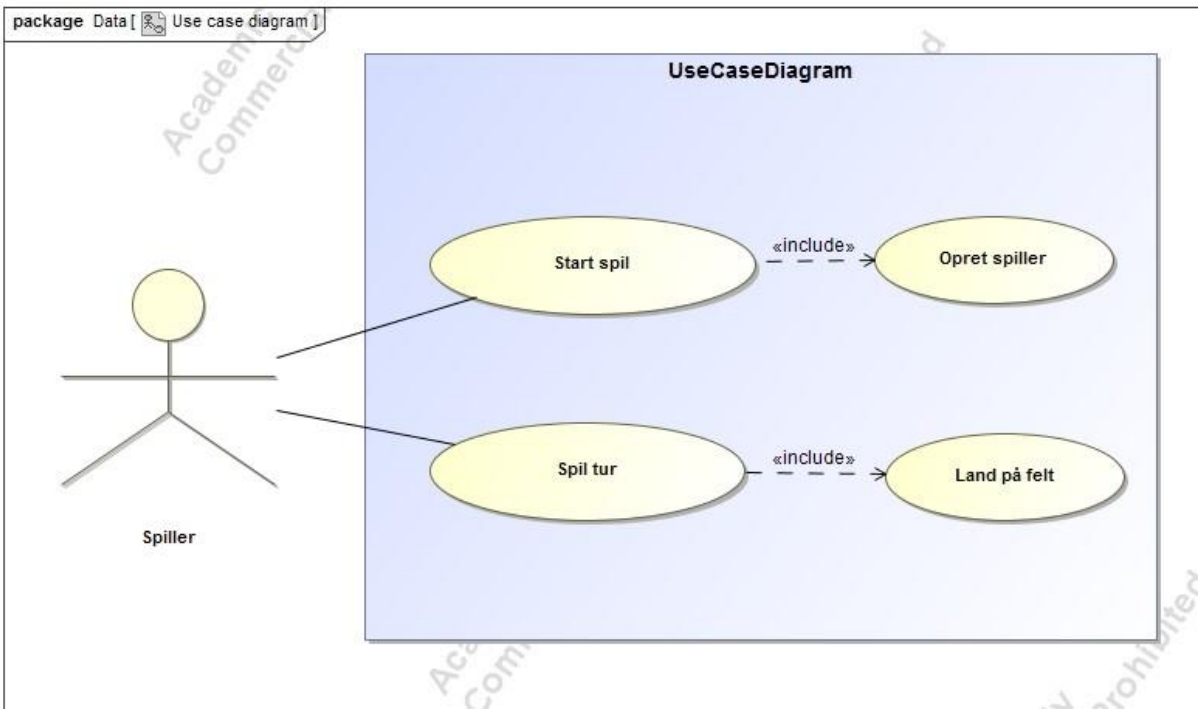
Med use cases fokuserer vi på, hvordan en aktør interagerer med systemet for at opfylde sine mål. For at finde de enkelte use cases er vi nødt til at vide lidt om systemet samt aktøren, der skal interagere med det.

*System boundary:*
Systemet er et brætspil, der består af 21 felter for 2-6 spillere. Spillerne slår med to terninger og flytter hen til et felt, hvor de skal udføre en "handling." bestemt af spillets spillogik.

*Aktør:*
Aktøren til dette system er en "spiller". Selvom spillet spilles af 2-6 spillere, er der ikke forskel på deres måde at interagere med systemet. Da alle spillere er ligestillet og kan det samme, kan vi nøjes med at fokusere på én aktør i form af en "spiller".

Spilleren er en person, som sammen med andre personer, kan spille brætspillet. Spilleren skal kunne starte spillet, rulle med terningerne, flytte til et felt, og udføre en handling, som det pågældende felter kræver. En del af det sker automatisk, så de to primære mål for aktøren er, at starte spillet og spille en tur.

## 6.4 Use case diagram




*Figur 1: Use case diagram*

Som det ses på use case diagrammet har vi den tidligere omtalte aktør, kaldet "Spiller." 'Spiller' har to muligheder for at få systemet til at udføre noget. Den ene use case 'Start spil' der også indeholder den inkluderende use case 'Opret spiller.'

Den anden use case er ‘Spil tur’. Dette er aktørens anden måde at interagere med systemet på. Det ses på diagrammet at ‘Land på felt’ også er en ‘include use case’ i forhold til den overordnede use case ‘Spil tur’. Det skyldes, at ‘Spil tur’ udfører ‘Land på felt’ når spilleren interagerer med systemet. Spilleren har altså mulighed for at gøre noget, når der landes på et felt.

Dette spillesystem er som udgangspunkt et meget lille og enkelt system. Aktøren har som nævnt ovenfor, kun 2 ting at foretage sig. Det er reelt ikke nødvendigt med flere use cases end de 2, men for at få en lille smule mere kød på, inkluderer vi de omtalte use cases. I større systemer vil der ofte være en del flere use cases.

## 6.5 Use case beskrivelser

Vi laver en use case beskrivelse for de enkelte use cases.

| |
|---|
| **Use case:** Start spil |
| **ID:** UC01 |
| **Brief description:** Spilleren skal angive sit navn og spillet startes. |
| **Primary actors:** Spiller |
| **Secondary actors:** N/A |
| **Preconditions: -** Adgang til databaren på DTU, eller anden maskine med programmet på.<br>**-** Man skal være mindst to spillere. |
| **Main flow:**<br>1. Systemet skriver en velkomstbesked.<br>2. Spiller trykker OK for at gå videre.<br>3. Spiller indtaster antal spillere og trykker på knappen OK.<br>4. Spiller opretter en spiller med pengebeholdning på 30.000. (Reference UC02)<br>5. Spillet startes. |
| **Postcondition**: Spillets startes, viser spilleren og hvilken spillers tur det er. |
| **Alternative flows:** N/A |

*Tabel 2. UseCase*

| |
|---|
| **Use case:** Opret spiller |
| **ID:** UC02 |
| **Brief description:** Gør det muligt at oprette en spiller |
| **Primary actors:** Spiller |

| |
|---|
| **Secondary actors:** N/A |
| **Preconditions:** Spillet er åbent |
| **Main flow:**<br>1. Spiller indtaster sit navn og trykker OK. |
| **Postcondition:** Spillernavn registreres og UC01 fortsættes |
| **Alternative flows:**<br>1. Indtaster brugeren 2 ens navne, beder systemet aktøren om at indtaste navnene igen. |

*Tabel 3: UseCase*

| |
|---|
| **Use case:** Spil tur |
| **ID:** UC03 |
| **Brief description:** Spilleren spiller en tur i spillet. |
| **Primary actors:** Spiller |
| **Secondary actors:** N/A |
| **Preconditions:** UC01 er kørt og spillet er startet. |
| **Main flow:**<br>1. Systemet skriver hvilken spiller, der skal slå med terningerne.<br>2. Spiller trykker på OK-feltet.<br>3. Systemet genererer to tilfældige tal mellem 1 og 6 for hver terning.<br>4. Summen af de to terninger vises, samt terningernes enkelte udfald på spillebrættet.<br>5. Spiller lander på et felt (Reference til UC04)<br>6. Turen afsluttes |
| **Postcondition:** Efter endt tur vises det, hvis tur det er. |
| **Alternative flows:**<br>1. Spilleren kan gå bankerot ved at lande på et felt.<br>a. Spilleren er ude af spillet, og turen går videre. |

Tabel 4: UseCase

| |
|---|
| **Use case:** Land på felt |
| **ID:** UC04 |
| **Brief description:** Spillerens brik rykkes til et nyt felt, og feltets ‘handling’ udføres. |
| **Primary actors:** Spiller |

**Secondary actors:** N/A

**Preconditions:** Spiller har trykket OK til spil tur.

**Main flow:**
Spiller lander på et af nedenstående felter, og nedenstående handling udføres:

1. Spiller lander på feltet "Territory"
    a. Er feltet ledigt, spørger systemet om feltet skal købes.
    b. Spilleren køber feltet ved at trykke på knappen 'JA'
    c. Beløbet trækkes fra spillerens balance.

    d. Er feltet i forvejen ejet, betales en leje fra spilleren, til den spiller, der ejer feltet.

2. Spiller lander på feltet "Refuge", og nedenstående handling udføres:
    a. Spilleren modtager et beløb til sin balance afhængig af hvilket Refuge-felt der landes på.

3. Spiller lander på feltet "Labor Camp", og nedenstående handling udføres:
    a. Er feltet ledigt, spørger systemet om feltet skal købes.
    b. Spilleren køber feltet ved at trykke på knappen 'JA'
    c. Beløbet trækkes fra spillerens balance.

    d. Er feltet i forvejen ejet, betales en leje fra spilleren, der lander på feltet, til spilleren, der ejer feltet.
    e. Lejen som spilleren skal betale, er terningernes sum for spillerens slag ganget med 100.

4. Spiller lander på feltet "Tax", og nedenstående handling udføres:
    a. Spilleren betaler en afgift til banken.
    b. Beløbet der betales er enten 4000 eller 10 % af pengebeholdningen

5. Spiller lander på feltet "Fleet", og nedenstående handling udføres:
    a. Er feltet ledigt, spørger systemet om feltet skal købes.
    b. Spilleren køber feltet ved at trykke på knappen 'JA'
    c. Beløbet trækkes fra spillerens balance.

    d. Er feltet i forvejen ejet, betales en leje til ejeren.
        - Har samme ejer ét Fleet-felt er lejen 500.
        - Har samme ejer to Fleet-felter er lejen 1000.
        - Har samme ejer tre Fleet-felter er lejen 2000.
        - Har samme ejer fire Fleet-felter er lejen 4000.

**Postcondition:** Har udført handlingen på et felt, og turen går videre.

**Alternative flows:**

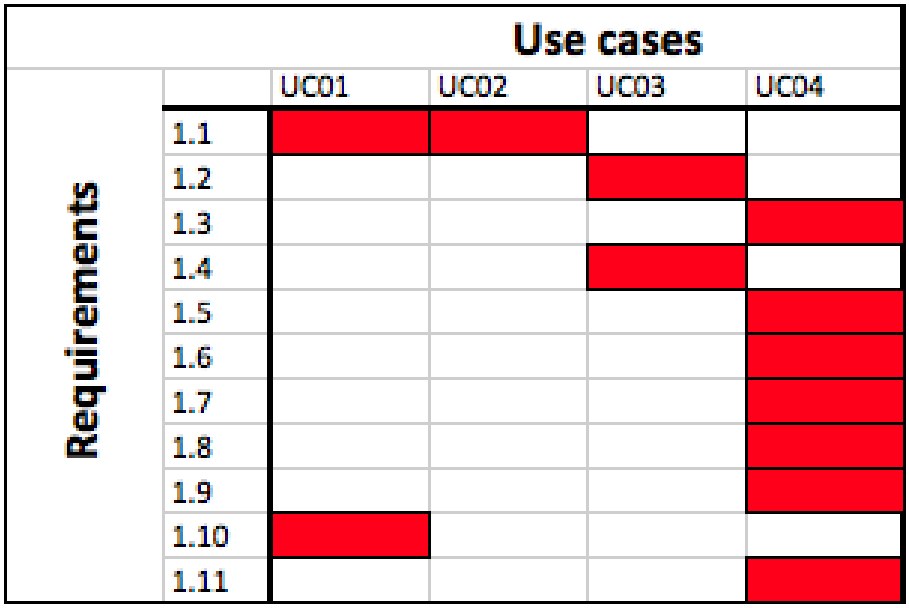
1. Spilleren kan vælge, ikke at købe feltet ved at trykke knappen 'NEJ'.
    a. Spilleren bliver stående på feltet uden at betale.

2. Spilleren kan gå bankerot ved at lande på et felt og modtage feltets konsekvens.
    b. Spilleren er ude af spillet, og turen går videre.

*Tabel 5: UseCase*

## 6.6 Requirement tracing

For at kombinere de præcise krav fra kravspecifikationen (hovedsageligt de funktionelle krav) med de mere formalistiske use cases, benyttes et requirement traceability matrix. Hovedreglen er, at hvert krav skal være dækket af minimum én use case, og ligeledes skal én use case dække minimum ét krav.

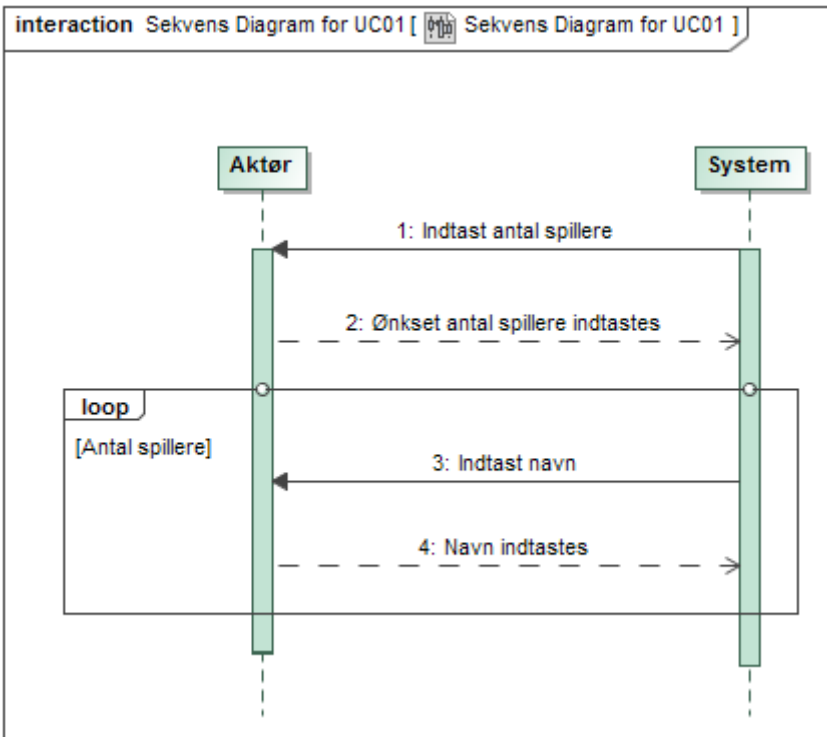



*Figur 2: Requirement tracing*

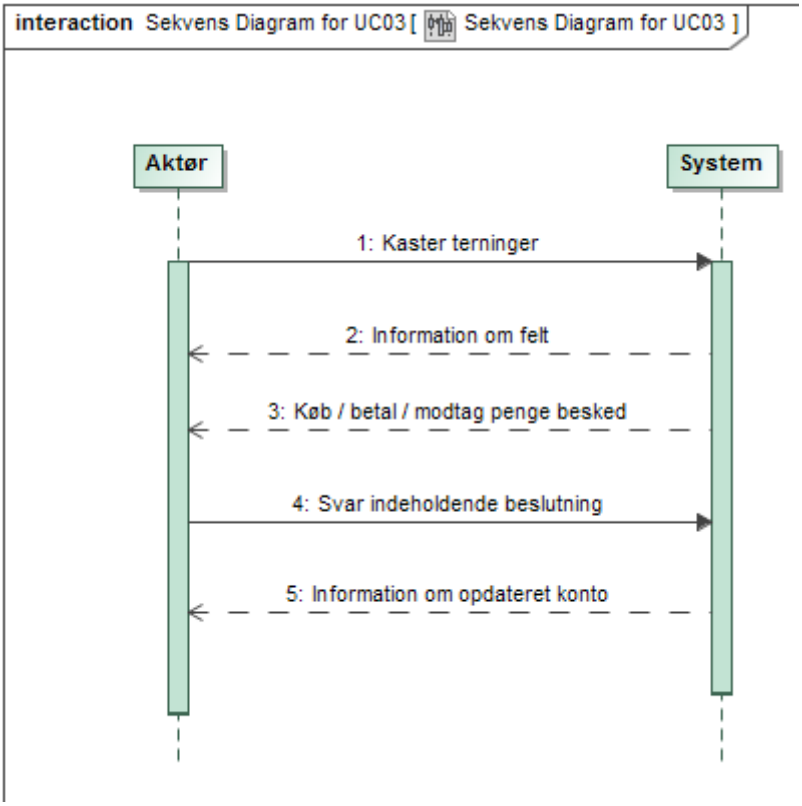
## 6.7 Systemsekvensdiagram

Vores system sekvens diagrammer viser, for en specifik handling i en use case, hvordan den eksterne aktør direkte interagerer med systemet og hvad deres handlinger generer. Tidslinjen følges fra top til bund og handlingerne bør følge rækkefølgen i diagrammet.

Figur 3 viser succes scenariet for et *Start Spil* scenarie. Dette diagram indikerer at systemet beder aktøren om at indtaste ønsket antal af spillere og navne på disse. Aktøren derimod svarer på systemets forespørgsler indtil at loopet, som har et boolean udtryk, skifter til false og spillet dermed kan gå i gang.




*Figur 3: Sekvens diagram for UC01*




*Figur 4: Sekvens diagram for UC03*

## 6.8 Navneordsanalyse

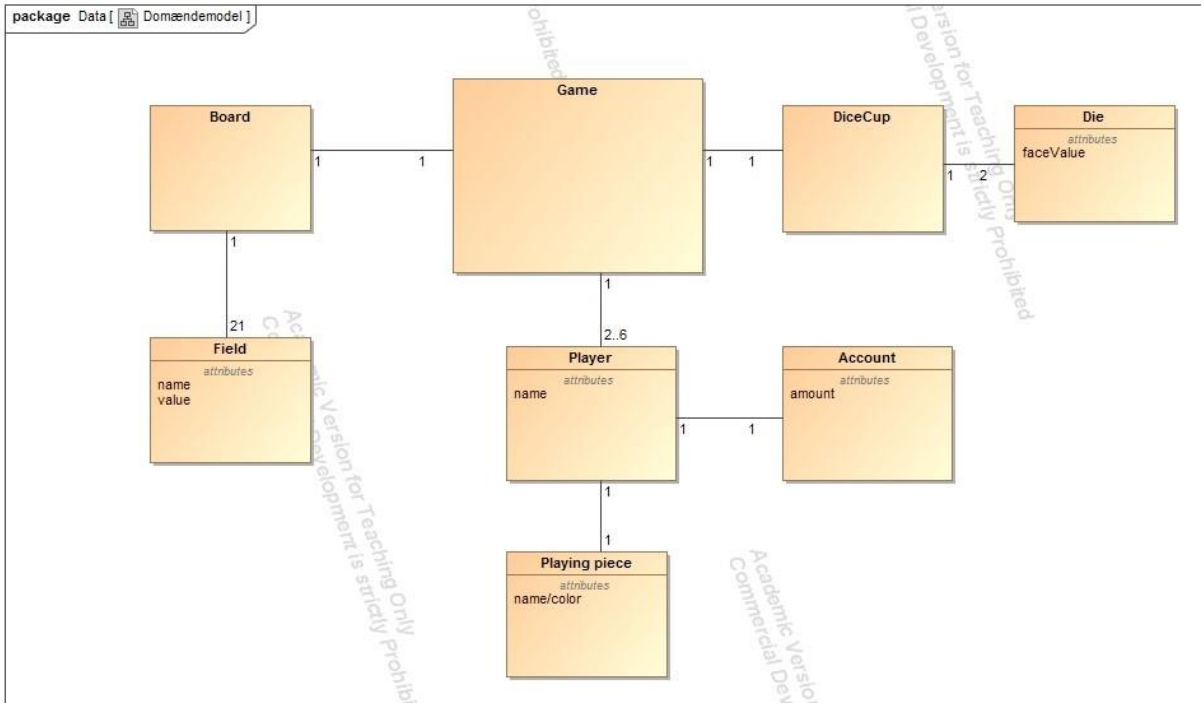
Med udgangspunkt i kundens vision, laver vi en navneordsanalyse. Ved at lave denne analyse får vi et overblik over, de elementer som er centrale for spillet. Navneordene giver os en ide om hvilke klasser vi skal bruge:

Terning
Spiller
Felt
Feltnummer/information
Spilleplade
Brik
Pengebeholdning/bankkonto
Effekt på konto
Spilliste
Raflebæger

*Tabel 6: Navneordsanalyse*

## 6.9 Domænemodel

På baggrund af navneordsanalysen har vi opstillet en domænemodel, som grundlæggende illustrerer et simpelt koncept og en virkelig forståelse af det spil vi skal udvikle. Det centrale ved domænemodellen er altså, at der er tale om såkaldte konceptuelle klasser, og ikke reelle software-klasser som afspejler koden. [1]




[1] Larman: Applying UML and Patterns 3rd

*Figur 5: Domænemodel*

På domænemodellen er tilføjet de mest nødvendige og logiske attributter, for at forstå spillet. For eksempel skal Player-klassen indeholde spillernes navne for at vide, hvilken spiller der interagerer med systemet. Derudover skal Field-klassen kende til et feltnavn og en eventuel feltværdi, ligesom klassen Playing piece er nødt til at kende en farve på en brik eller et navn, der knyttes sammen med et spillernavn.

Af domænemodellen fremgår også multipliciteten for de enkelte konceptuelle klasser.
Game har en 1-1 relation med Board, da selve spillet indeholder ét spillebræt. Fra Bord til Field ses en relation på 1-21, idet, det ene spillebræt indeholder 21 felter.

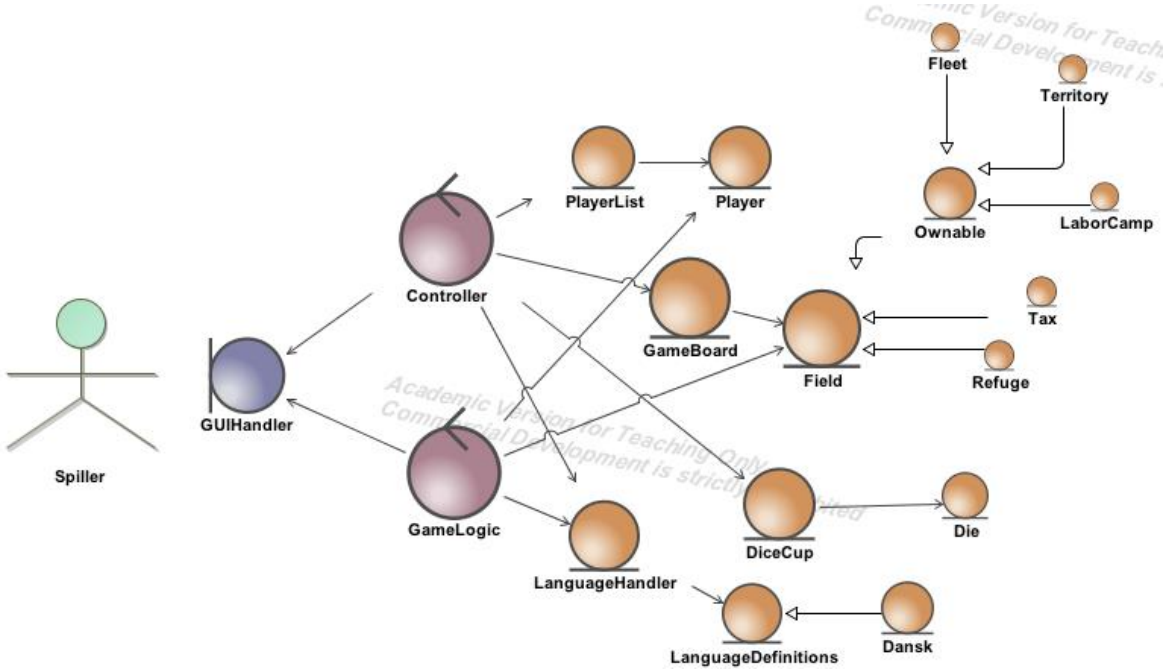
Hvis vi tager et kig på relationen mellem Player og Game ses det på multipliciteten, at den er 2..6 til 1, hvilket betyder at de 2 til 6 spillere har den samme relation overfor Game og dermed alle har ens attributter. Player klassen har ligeledes en relation til Account for at vide, hvor mange penge den enkelte spiller har. Player klassen har desuden en relation til Playing piece klassen, som holder styr på, hvor på spillebrættet spilleren står.

Spillet skal kunne spilles ved hjælp af terninger, og denne proces sker i Die klassen. Spillet spilles med 2 terninger, så relationen er 2-1 til DiceCup, som returnerer terningernes udfald og sum.

## 6.10 BCE model

Den her BCE model skitserer hvordan ansvaret er fordelt hos de enkelte klasser. I modellen har vi to controller klasser. Controller modtager informationer af entity klasserne og styrer use case realisering. Gamelogic kalder en metode, FieldRules, som har relation til Player, Field og Languauge - da den styrer konsekvenserne for at lande på et specifikt felt. Vi har valgt denne struktur, så ingen af entiteterne har nogle relationer til GUI'en.

Entity klasserne indeholder informationer om hvordan systemet forskellige dele af systemet fungerer og interagerer med hinanden. fx interagerer PlayerList med Player.
Entity klassen field er superklassen for vores felter. Felterne kan opdeles i to kategorier. Nogle felter kan købes og dermed ejes, og nogle felter medfører en gevinst eller en bøde. På samme måde er de delt op i vores BCE diagram, hvor man har mulighed for at eje 3 slags felter. Disse er nedarvet fra entity klassen Ownable. Dermed er de 2 sidste slags felter nedarvet fra superklassen Field. Nedarvningen er illustreret via pilene (generalization arrow). Alle entity klasser afleverer informationen til Controller, som formidler informationerne og data videre til vores boundary klasse GUIHandler. Her vil GUIHandler være aktørens primære user interface og aktørens måde at interagere med systemet på.

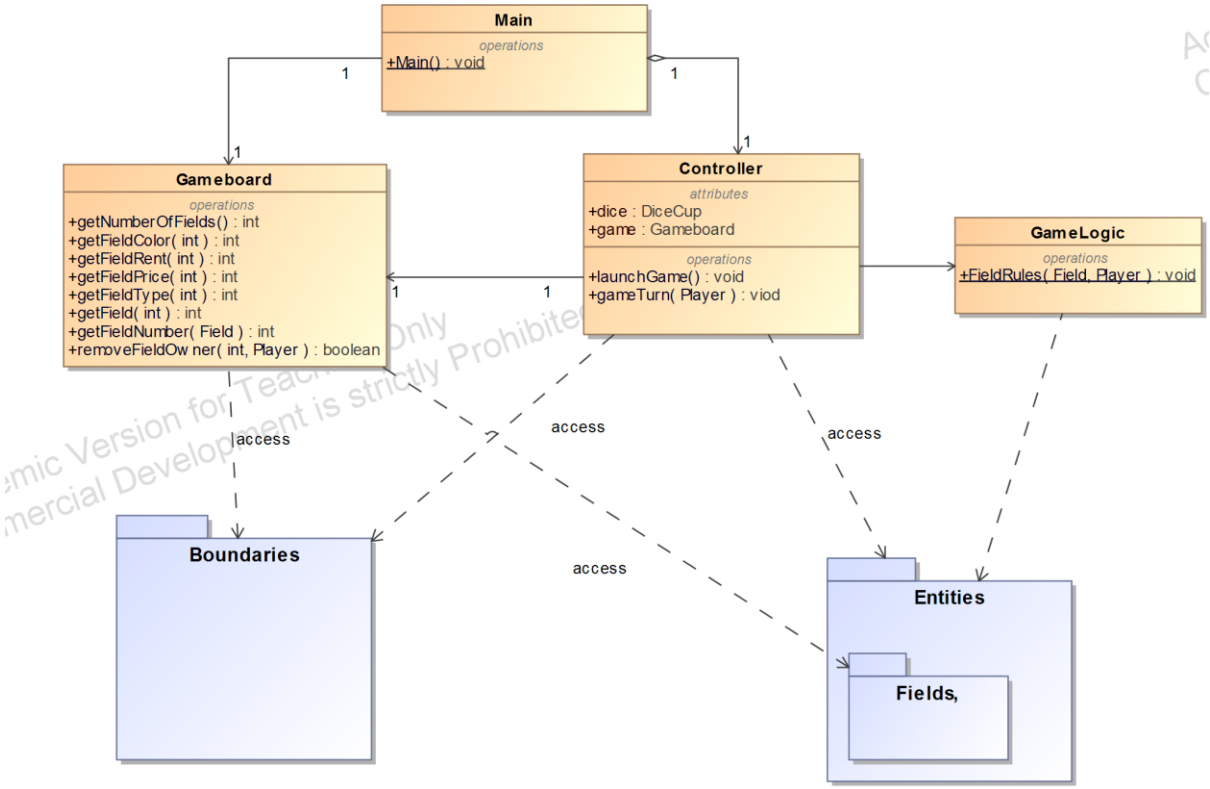



*Figur 6: BCE model iteration 3 (final)*

# 7 Design

## 7.1 Klassediagrammer

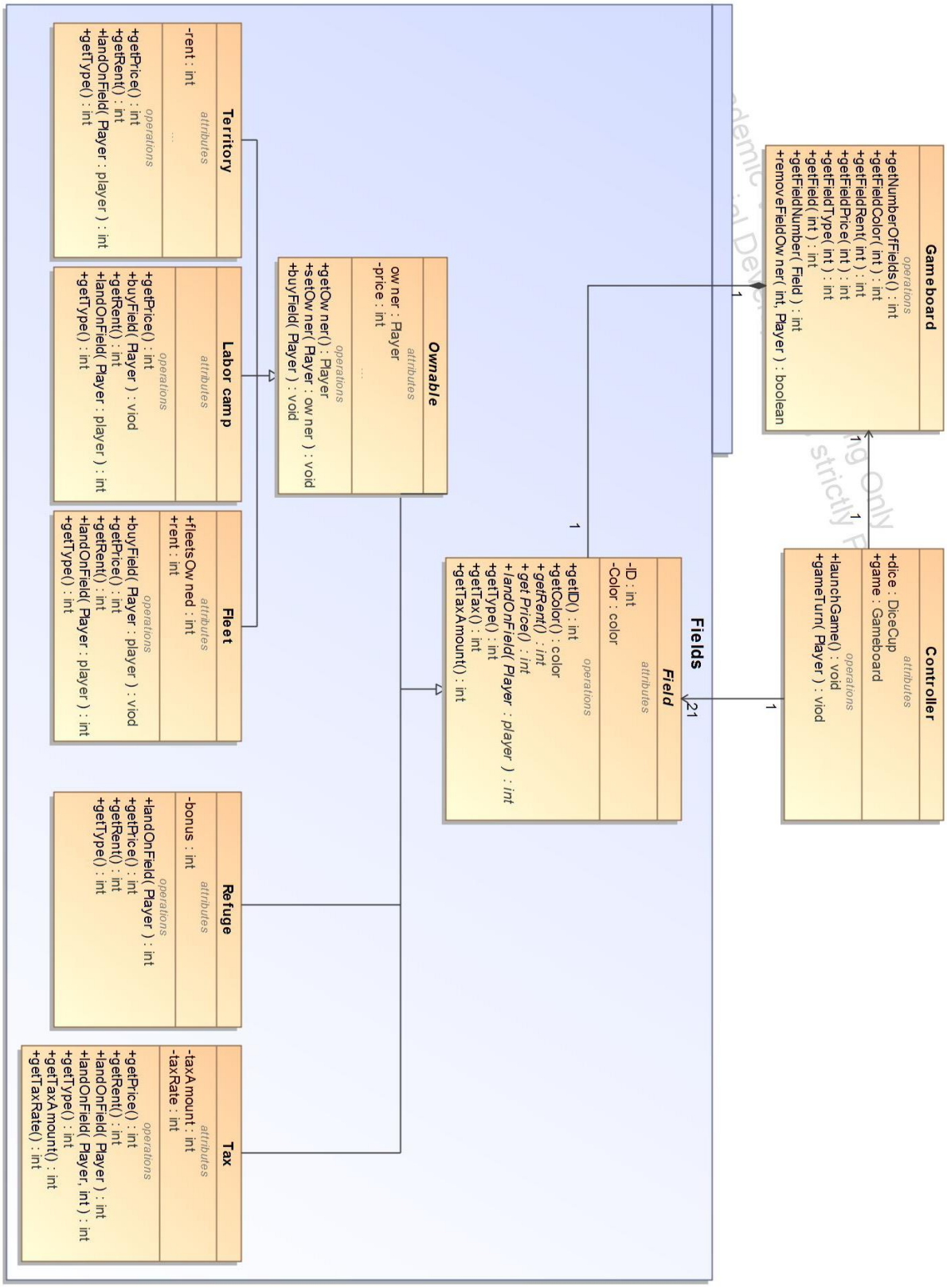
Design klassediagrammet giver et overblik over de forskellige klassers attributter og metoder. Derudover viser klassediagrammet, hvorledes de forskellige klasser spiller sammen. Da vores klassediagram er for stort til at vises i et enkelt billede, har vi valgt at bryde det op i mindre dele. For overblikkets skyld har vi dog lavet et klassediagram som viser, hvordan fx controlleren går ud i de forskellige klasser - her vist som pakker.




*Figur 7: Design klassediagram*

Da det kun er pakkerne som er synlige, skal den stiplede linje fra hhv. Controller og Gameboard forstås således, at de har adgang de respektive klasser i mapperne.

Nu skal vi se nærmere på de enkelte klasser, og deres samspil med hinanden. Nedenstående klassediagram viser, hvorledes vores pakke 'Fields,' er bygget op af et arvehierarki.



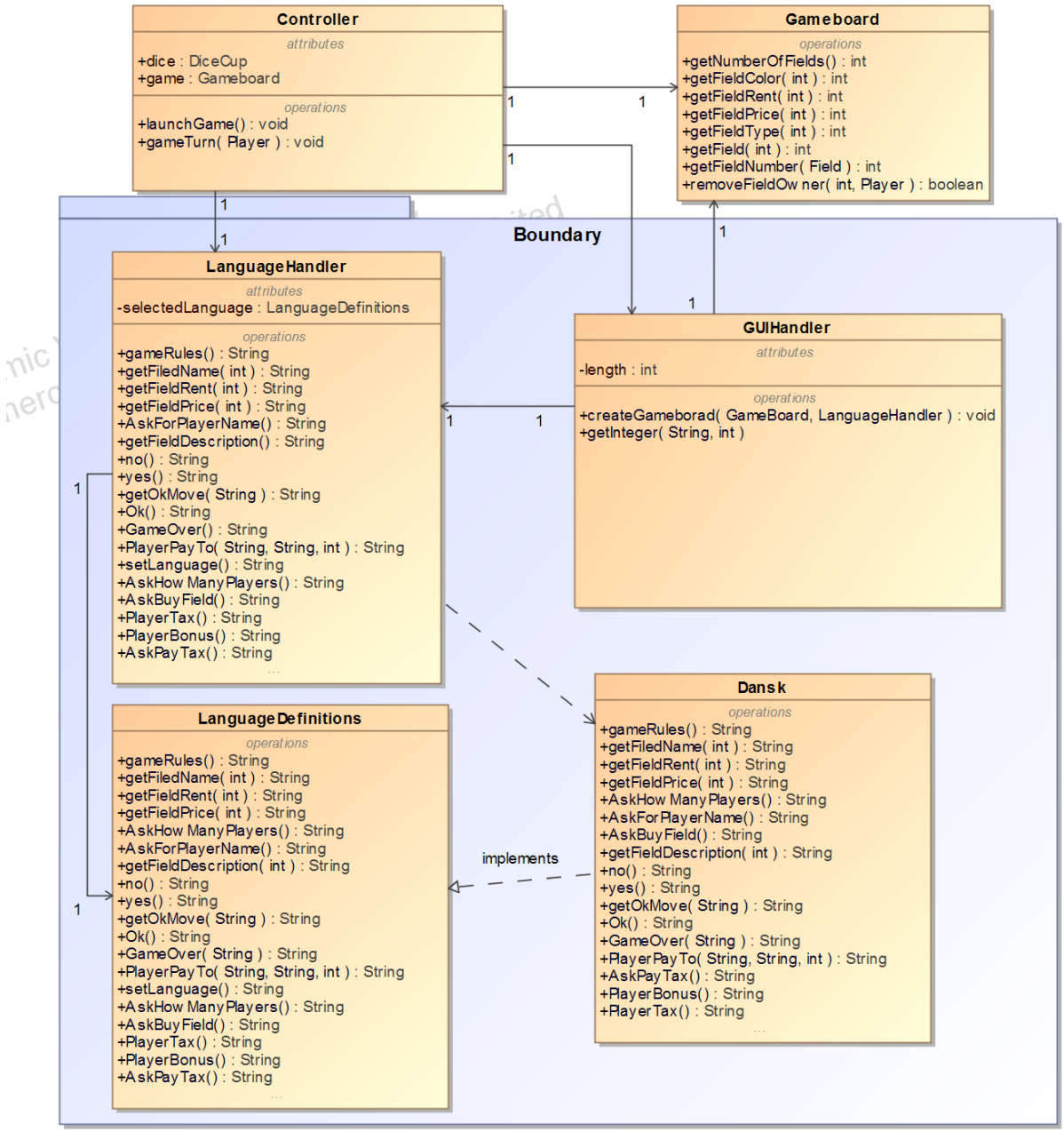

*Figur 8: Design klassediagram - Fields pakken*

I figur 8 ser vi Field-klassen som er superklassen - altså den klasse som de andre klasser nedarver fra. Dette fremgår af UML diagrammet, i form af de "tomme pile". Klassen *Field* er iøvrigt en abstrakt klasse, da den blot skal definere et koncept, i form af et felt. Dette fremgår også af UML diagrammet, idet klassens navn skrives i kursiv. Karakteristisk for abstrakte klasser er, at vi ikke kan lave instanser af en sådan type klasse, fordi dens metoder blot beskriver noget generelt. Abstrakte metoder fra superklassen nedarves til klasserne og i underklassen skal disse metoder defineres fuldt ud. Således får man specialiseret hver enkelt klasse med sine egne karakteristika.

Fra *'Field'* nedarves klasserne *'Ownable'*, 'Refuge' og 'Tax'. De to sidstnævnte henter metoder og attributter fra superklassen *Field*, fordi de to felter ikke kan købes. Da det er muligt at købe felter i spillet, har vi lavet den abstrakte klasse *'Ownable'* som nedarver fra superklassen *'Field'*. *'Ownable'* er således en underklasse til *'Field'* men superklasse til de felter som er mulige at eje, nemlig: 'Territory', 'Labor Camp' og 'Fleet'. *'Ownable'* indeholder derfor alle de metoder og attributter som nedarves fra *'Field'* men også de metoder og attributter som er nødvendige for dens underklasser. I *Ownable's* underklasser specialiseres yderlige ved at fx 'Fleet' får fx attributten 'rent' og 'fleetsOwned'.

Figur 9 viser vores boundary-pakke. Her har vi klassen LanguageHandler, der laver metoder, som hentes fra 'LanguageDefinitions'. Klassen 'Dansk' implementeres i klassen 'LanguageDefinitions'. Her er der tale om polymorfi, da 'LanguageDefinitions' beskriver Dansk's grænseflade. Vi lægger op til, at en anden klasse skal kunne udføre samme metode, men med en anden returnering. Herved kan man lave en ny klasse med et andet sprog. 'LanguageDefinitions' indeholder nemlig en masse String-metoder, hvortil klassen 'Dansk' implementerer sine Strings i 'LanguageDefinition's metoder.

Der er trukket en dependency linje mellem 'LanguageHandler' og Dansk-klassen fordi 'LangugageHandler' er afhængig af, at der er et sprog implementeret.

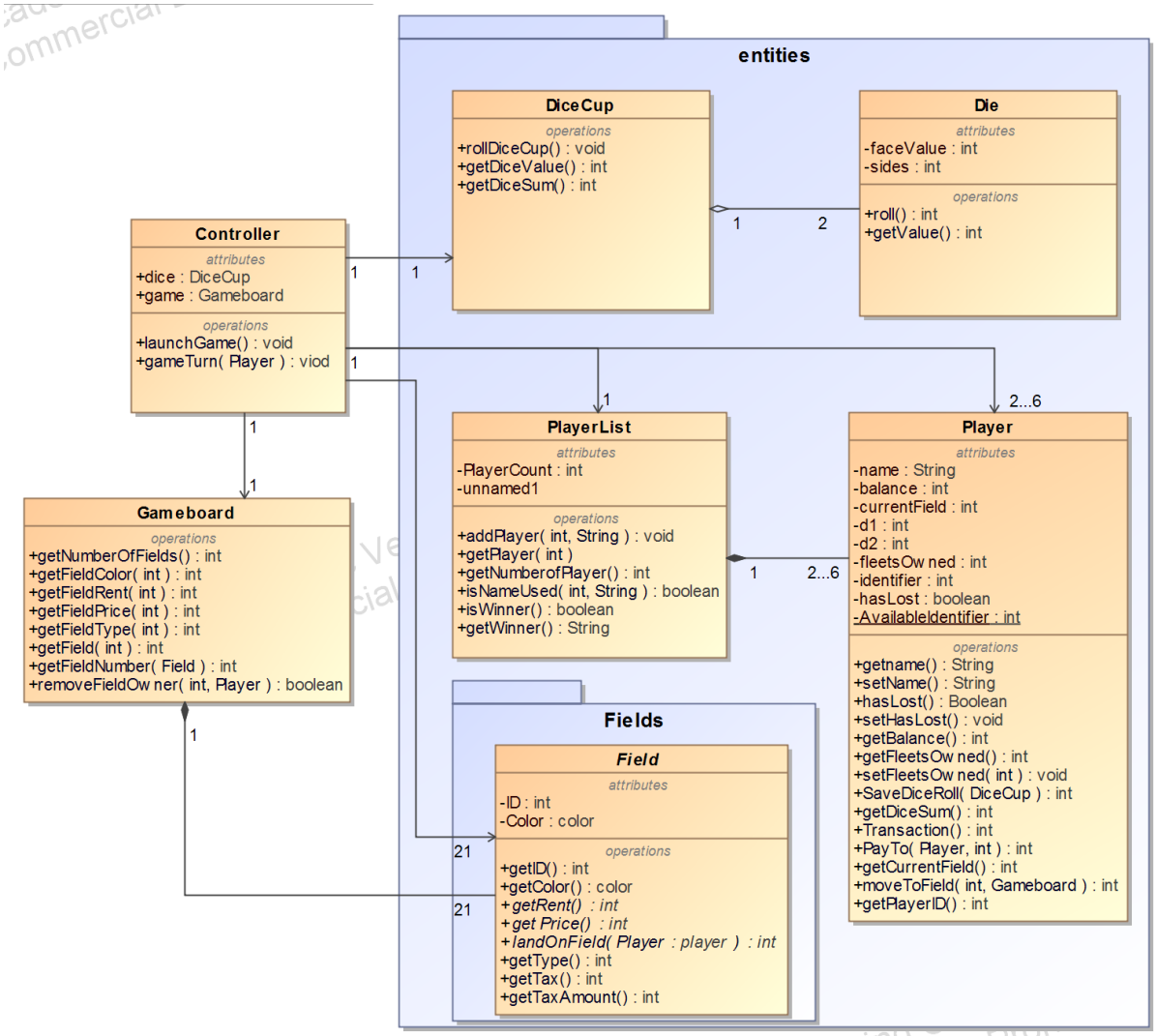



*Figur 9: Design klassediagram - Boundary pakken*

Figur 10 viser entitets-pakken med de tilhørende klasser. Af diagrammet fremgår det, at forholdet mellem DiceCup og Die kan beskrives med aggregation. DiceCup er ikke så brugbar uden Die, men den kan godt stå alene. Multipliciteten illustrerer ligeledes, at der går 2 terninger til 1 refelbæger.

Forholdet mellem GameBoard og *Field* er et compositions-forhold, da Gameboard er meget afhængig af *Field.* Der er ikke noget Gameboard uden *Field*. Her multipliciteten 1 til 21, idet vi har et Gameboard overfor 21 felter.

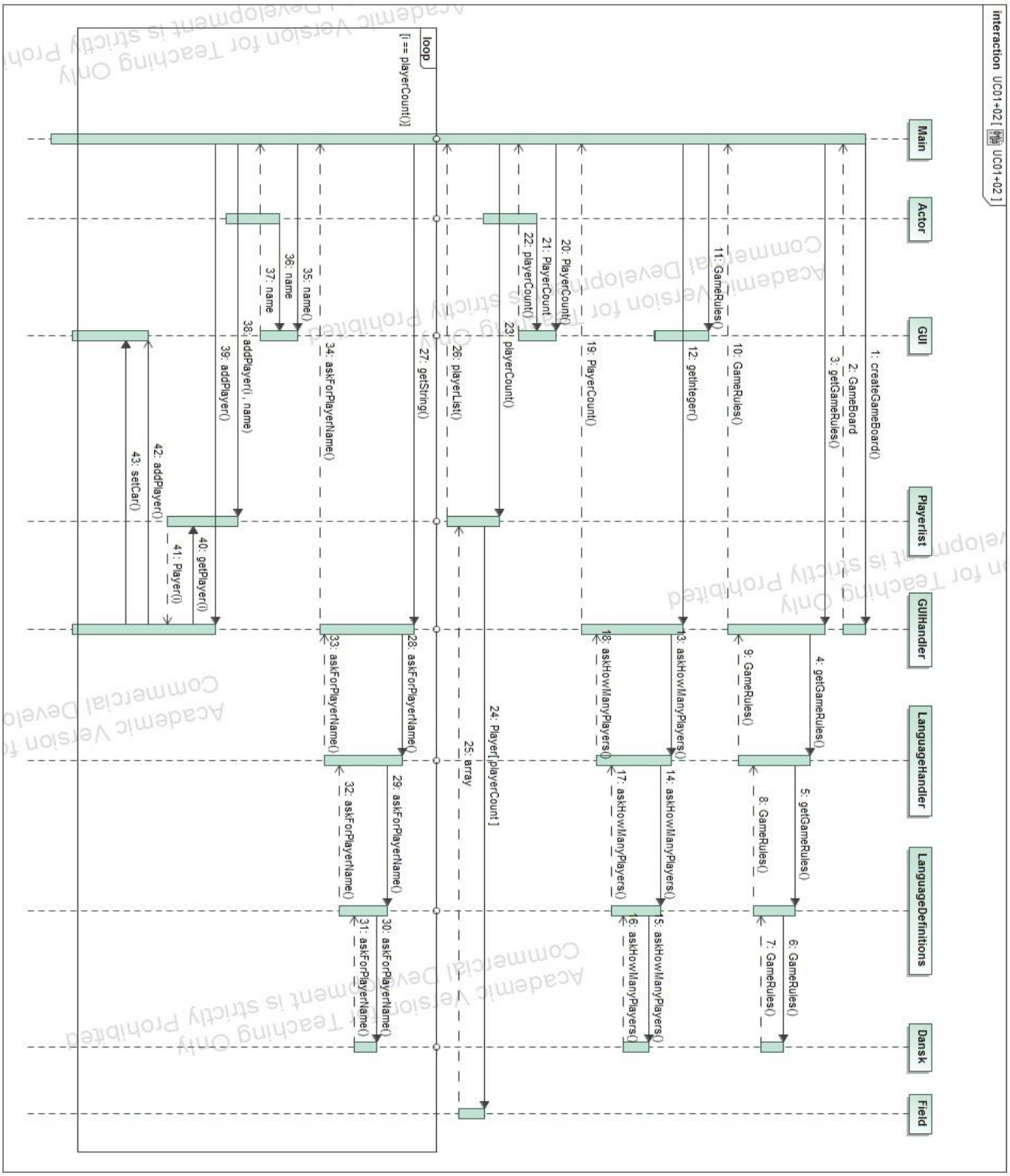
Der er også et compositions-forhold mellem PlayerList og Player, idet PlayList laver et array af Players. Multipliciteten mellem de 2 klasser er 1 til 2...6. Det betyder, at der laves én PlayerList i forhold til om der er 2 til 6 spillere.




*Figur 10: Design klassediagram - Entitets-pakken*

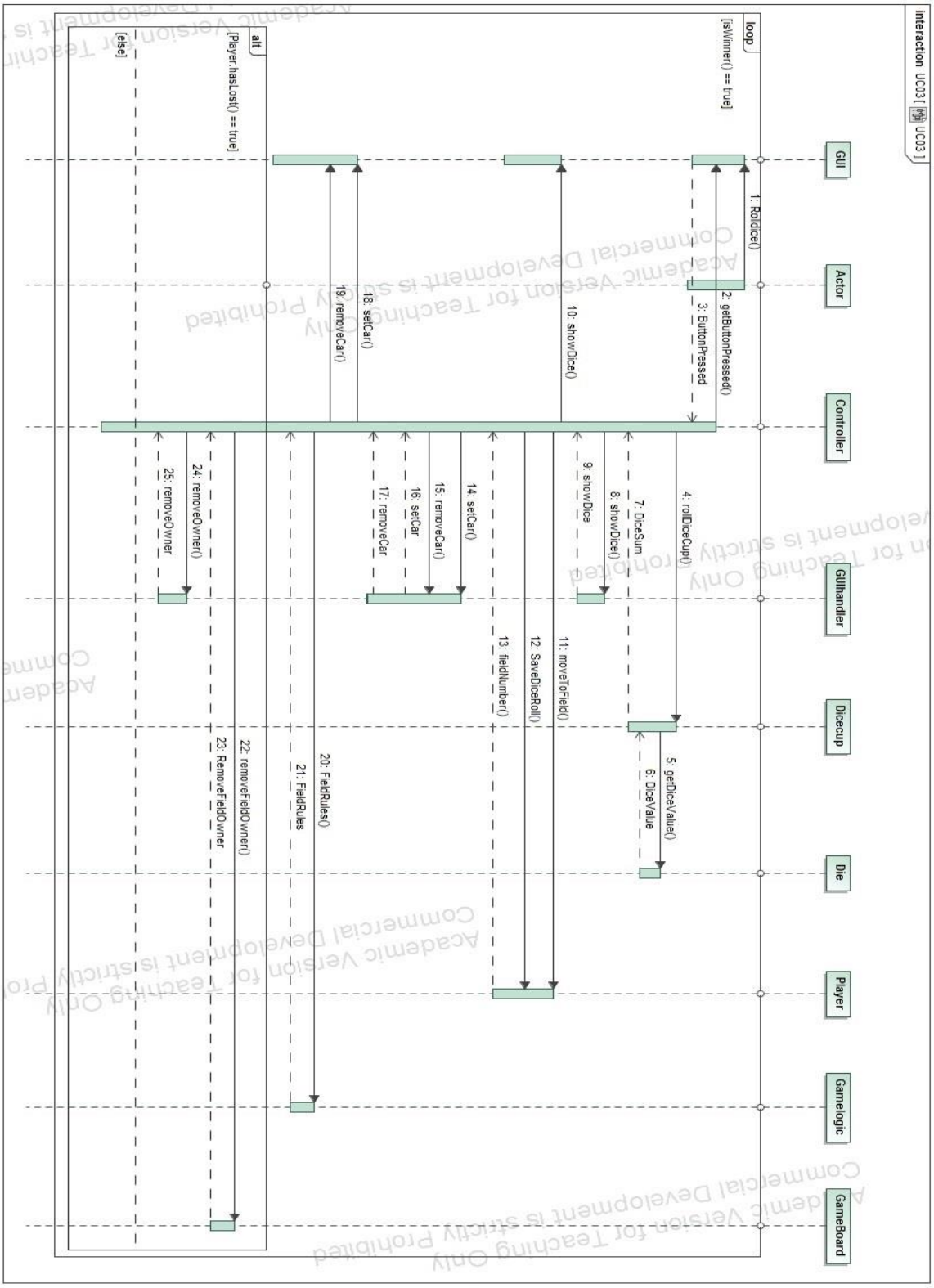
## 7.2 Sekvensdiagrammer

### 7.2.1 Sekvensdiagram for UC01 & UC02



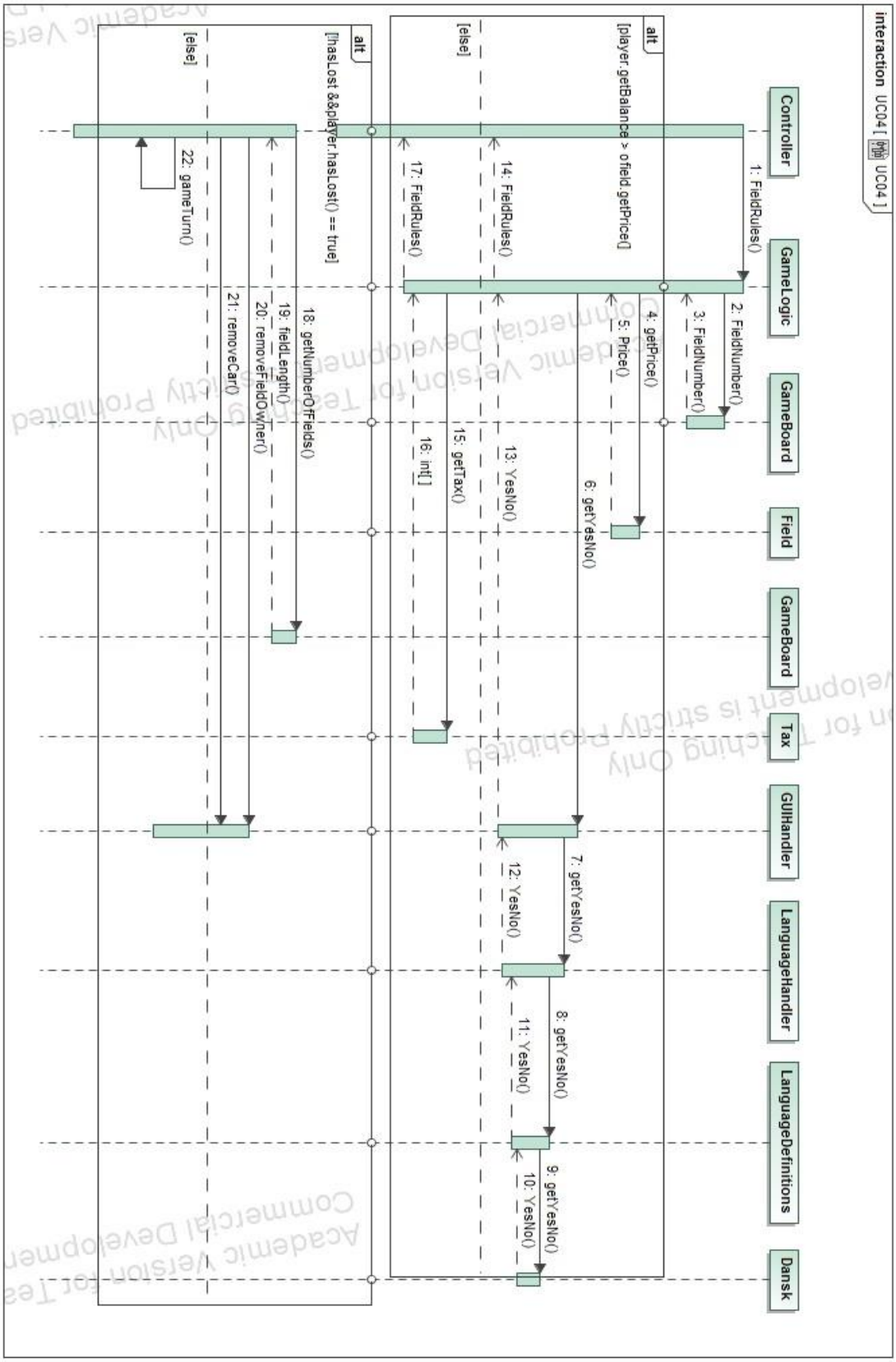

*Figur 11: Design Sekvensdiagram UC01+02*

### 7.2.2 Sekvensdiagram for UC03



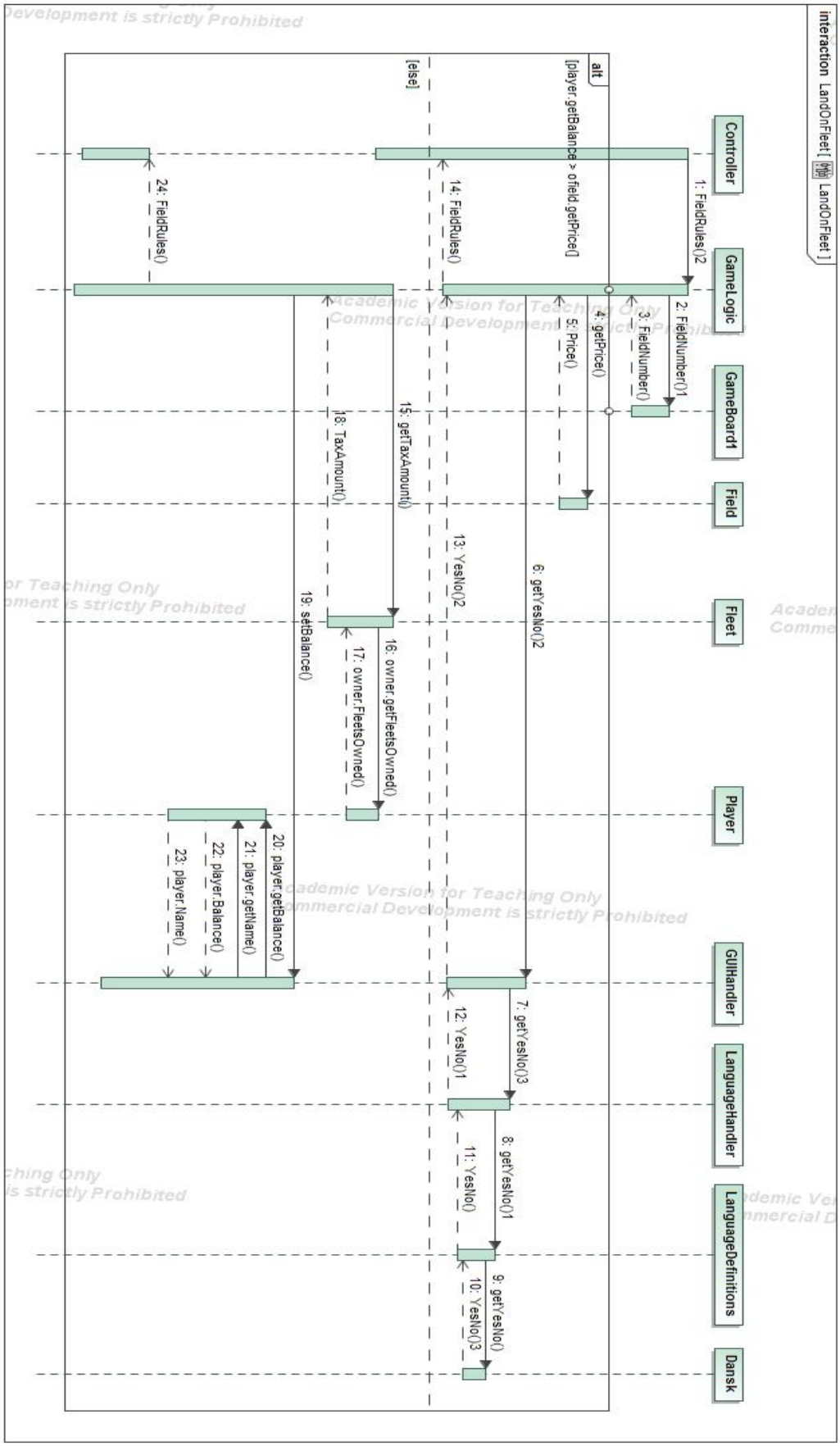

*Figur 12: Design Sekvensdiagram UC03*

### 7.2.3 Sekvensdiagram for UC04




*Figur 13: Design Sekvensdiagram UC04*

*Figur 14: Design Sekvensdiagram LandOnFleet*



**Design sekvensdiagram - UC01+02:**
Vores første design sekvensdiagram beskriver use case 01 og 02. Rettere beskriver vi hvordan aktøren opretter en profil for hver af de spiller de ønsker i spillet. Her har vi et minimum på 2 spillere og et maksimum på 6 spillere. Alle klasser i systemet er ikke vist i diagrammet, kun de essentielle klasser for selve aktionen at oprette en spiller.

**Design sekvensdiagram UC03:**
Design sekvensdiagram UC03 beskriver hvordan en spil tur forløber i vores program. Her er udeladt de alternative aktioner der forekommer alt efter hvilket felt man lander på. Derimod er den del af en spiltur repræsenteret ved at controlleren henter FieldRules() fra GameLogic, hvilket bestemmer hvilken aktion der sker ved et givet felt. 'Alternative' boksen som starter ved aktion 18. viser hvordan programmet reagerer når en spiller har tabt og hvordan programmet sletter spilleren.

**Design sekvensdiagram UC04:**
I UC04 ser vi hvordan programmet bestemmer hvilken effekt et specielt felt har på spilleren. I dette tilfælde har vi valgt at vise hvordan Tax fieldet bliver brugt. Da disse felter fungerer meget ens har vi valgt at repræsentere dem alle med et eksempel. Den første alternativ boks viser hvad der sker når en spiller lander på et felt der kan købes og den anden alternativ boks viser hvordan systemet fjerner ejeren fra hans købte felter når spilleren taber.

**Design sekvensdiagram LandOnFleet:**
I dette diagram har jeg antaget at spilleren er landet på et "Fleet" felt og at dette felt enten kan købes eller er ejet af en anden spiller. Alternative boksen viser at hvis feltet ikke kan købes så skal spilleren betale taxAmount til ejeren alt efter hvor mange "Fleet" felter ejeren har i forvejen.

**Kommentar:**
Vi har med vilje ikke taget alle messages og aktioner som koden laver i vores use cases med i diagrammet. Nogle af aktionerne som at få "ok" knappen fra GUIHandleren og at hente sprog fra Dansk hver gang spiller skal godkende deres kast eller noget lignende, er ikke særlig interessant i diagrammet og derudover ville det gøre diagrammet meget uoverskueligt. Selvom diagrammerne dermed ikke fuldstændig afspejler vores kode, giver de stadigvæk en meget detaljeret ide om hvordan klasserne interagere med hinanden.

# 8 Implementering

## 8.1 Implementation på baggrund af GRASP

I dette afsnit vil vi gennemgå de klasser vi har programmeret og de vigtige beslutninger der er blevet foretaget i kodningsprocessen i forhold til både GRASP og intelligent programmering.
Programmet er skrevet i 4 packages: boundary, controller, entity og en rungame package. Vores main ligger i rungame. Programmet er inddelt på den måde i et forsøg på at udnytte GRASP principperne Low Coupling og High Cohesion. Dette giver programmet en god struktur, som samler de forskellige klasser i pakker til deres respektive ansvar. Samtidigt adskiller de dele af programmet, som ikke burde interagere, og skaber en kodestruktur, som er let at forstå og videreudvikle på.

## 8.2 Nedarvning og Polymorfi

Vi bruger nedarvning til at fordele metoder som er (eller kan blive) generelle for flere klasser. Dette har vi gjort når vi har udviklet felterne. Der er flere ting som er generelle eller kan blive generelle for alle felterne, og det er derfor smart at give dem en superklasse, der holder styr på, hvad der er generelt. Vi bruger f.eks. vores *Field*-klasse til at tildele alle vores felter ID og farve. Det bliver smart, når vi laver vores array af felter i GameBoard, hvor vi kan tildele alle vores forskellige felter et ID og en farve.

## 8.3 Ownable

Vi har lavet endnu en superklasse i feltklasserne kaldet ”*Ownable*” den udvider fra Field klassen, så vi kan tildele de typer felter der kan ejes en ejer. Igennem ”*Ownable*” sender vi metoder videre til at købe felter, da det kun er de felter der kan ejes.

## 8.4 LandOnField

I field klassen laver vi en abstract metode som vi kalder LandOnField. Når man laver en abstract metode, tvinger man alle underklasserne (som ikke også er abstract) til at implementere metoden, det er på denne måde vi er i stand til at lave en metode, som er generel for alle felter, men udfører forskellige ting, dette kaldes polymorfi. Klasser der udvider på en en abstract class skal ‘extende’ klassen den udvider fra.

LandOnField klassen anvender også i et tilfælde ‘Overloading’, dette er når man i en klasse har to forskellige implementeringer af en metode, som skelnes fra hinanden ved at de har forskellige parametre.

Det er smart at kunne kalde vores felter med den samme metode, men hvor der sker forskellige ting, da vi på den måde behøver mindre kontrollogik i controlleren. Således kan vores børn og børnebørn (klasserne der extender ownable, som er et barn af fieldklassen ) have forskellig logik når vi lander på et felt, selvom de kaldes fra samme metode.

Nu vil vi gennemgå polymorfien i vores feltklasser, som kommer til udtryk i metoden LandOnField. LandOnField implementerer de aktioner, der hænder når en spiller lander på et felt. For eksempel:

- LandOnField i refugeklassen tildeler spilleren en bonus. Den eneste parameter til metoden i dette tilfælde er spilleren (player).
- LandOnField i tax findes i to udgaver. Den ene metode der kun har *Player* som parameter, trækker et fast beløb i skat. Den anden udgave af metoden har som parametre både *Player* og *taxRate* (skattepct. som beregnes af saldobalancen), og trækker denne beregnede værdi. De metoder er implementeret ved 'Overloading'.
- LandOnField i Territory, som er et barn af ownable, checker om der er en ejer på feltet, hvis der er, skal spilleren betale en fast leje, den eneste parameter er *Player*. Ownable klassen har en intern variabel, som peger på hvilken spiller der evt. ejer feltet.
- LandOnField i Fleet checker for ejer på feltet, dernæst hvor mange '*fleets'* dén ejer ejer, og beregner lejen ud fra antallet af ejede fleets. Den eneste parameter er *Player*. *Player* klassen har en intern variabel, med tilhørende Getter/setter metoder, der holder styr på hvor mange fleets en given spiller ejer.
- LandOnField i Labour Camp checker for ejer på feltet, dernæst hvor mange '*Labor Camps'* dén ejer ejer, og beregner lejen ud fra antallet af ejede Labor Camps. Den eneste parameter er *Player*. *Player* klassen har en intern variabel, med tilhørende Getter/setter metoder, der holder styr på hvor mange Labor Camps en given spiller ejer.

## 8.5 Language

Vi anvender også nedarvning i vores LanguageDefinition klasse, som er en interface klasse. Som tidligere beskrevet indeholder den en masse tomme streng-metoder, som bliver nedarvet til "dansk", hvor vi så skriver teksten, der skal stå i strengene. Det er smart, da det betyder, at vi nemt kan tilføje et nyt sprog uden at fjerne dansk, ved at lave f.eks en klasse der hedder "english", som implementerer language definitions igen. Den linje kode, som kan skifte mellem sprog ligger inde i main.

## 8.6 Programstruktur

Under udviklingen af brætspillet forsøgte vi at holde logikken, hvor den naturligt hører hjemme. F.eks. har vi lagt funktionaliteten, hvor vi overfører penge mellem spillere, ind i Player klassen. Det giver spillet en form for naturlig logik: En spiller betaler en spiller. Hvis vi i stedet f.eks. havde valgt at samle transaktioner i en account klasse, ville man enten være nødt til enten lave et metodekald, som gemmer pengene, der skal overføres, og så en metode der rent faktisk overfører pengene. Omvendt kunne man lave én metode, som havde parametrene payer og payee, men dette virkede umiddelbart ikke så logisk for os, og vi lagde derfor payTo funktionen i Playerklassen. Selve betalingen sker nede i Fieldklassens LandOnField metode. Fordelen ved dette er, at man automatisk har information om, hvilket felt, hvilken spiller og evt. hvilken ejer, ét sted bla. fordi feltklasserne er defineret sådan, at de ved hvem dens evt. ejer er.

## 8.7 Datastruktur

Vores GameBoard definerer et array af vores feltklasse, det tilsvarende array af feltnavne er erklæret i sprogklassen (dansk). Når felterne oprettes i GameBoard, tildeles de en int id, som anvendes til at sammenkæde feltet med det modsvarende felt i sprogarrayet. Fordelen er, at rækkefølgen mellem de to arrays ikke behøver at være ens. Det vil sige, at vi giver udvikleren mulighed for efterfølgende at ændre rækkefølgen i GameBoard arrayet, uden at han behøver at opdatere sprogklassen. Dette er en fordel idét den mindsker arbejdet, der skal udføres for at rette spille. Derudover mindsker det risikoen for fejl, da man skal lave færre rettelser.
Vi lavede et spiller ID, da vi ville have en måde vi altid kunne genkende spillerne fra hinanden, men det viste sig, at vi bare kunne checke deres navne-streng, da vi alligevel var nødt til at programmere en funktion, der blokerede spillere for at hedde det samme. Dette viste sig undervejs fordi GUI biblioteket kræver at spillere har forskellige navne.

## 8.8 Controllers

Spillet bliver initialiseret i main der kalder controlleren, som sørger for at holde spillet kørende. Controlleren starter og slutter spillet, og bestemmer hvad der sker under turene, der henter feltlogik fra GameLogic, som indeholder reglerne for felterne.
Det var meningen, at vores controller skulle være vores eneste 'bro' imellem entitetsklasserne og boundaryklasserne for at overholde Low Coupling princippet fra GRASP. Men vi har måttet være nødt til at hente GameBoard ind i GUIHandler, da det var den nemmeste måde vi kunne behandle oprettelsen af felter intelligent.

# 9 Test

Efter færdig udviklingen af spillet, er programmet testet i en funktionstest, så vi verificere om programmet er funktionelt og opfylder kundens krav. Dette er testet med JUnit og brug af programmet.

## 9.1 Grey box test

Vi har testet om det er muligt at få programmet til at crashe, ved at give programmet input som ikke er forventet i koden.

**Test af spiller navns registrering**

| **Input** | **Output - Spiller brik navn** | **Besked om tur** | **Godkendt** |
|---|---|---|---|
| *"123qweasdzxc456rtyfg hvbn789uiojklm,."* * 4 | "123qweasdzxc456rtyf" | Hele navnet vises. | Ja |
| "øæå!"#¤%&/()=@£$€{[]}" | "øæå!"#¤%&/()=@£$€{[]} | "øæå!"#¤%&/()=@£$€{[]} | Ja |
| 〝中国人〞 | 中国人 | 中国人 | Ja |

*Tabel T1*

I tabel T1 kan det ses at det at både danske bogstaver, danske specialtegn, samt kinesiske tegn kan anvendes til spillernavn. Dette er grundet at GUI'en bruger en String til at læse navnet som er angivet. Se bilag 9.

## 9.2 Test Cases

| | |
|---|---|
| Test ID | TC01 |
| Beskrivelse | Denne test case tester Use Case 1 og 2.<br>Denne test viser: om spillet kan startes, om velkomst beskeden bliver vist, om man kan vælge antallet af spillere (2-6) og om spillernavne registreres. |
| Krav | 1.1 & 1.10 |
| Forudsætning | Adgang til computer med applikationen på. |
| Efterfølgende betingelser | Spillet skal stadigvæk køre efter navne er indtastet, og spillet skal fortsætte. |
| Test procedure | 1. Kører Jar-fil (Start spil)<br>2. Se om velkomstbesked bliver vist<br>3. Vælg antallet af spillere (2-6) |

| | |
|---|---|
| | 4. Indtast spillernavne<br>5. Konstatér om alle spillere har en brik og om start balance for spillerne er 30.000. |
| Test data | 1. Åbnet Jar-fil (spil)<br>2. Klikkede på “ok” knap.<br>3. Indtastede 4 tekstfelt og klikkede på “ok” knap.<br>4. Indtastede følgende navne i tekstfelt: Dan, Mette, Mogens og David , efter hvert navn blev der klikket ok før indtastning af det næste. |
| Forventet resultat | - Vi forventer at:<br>- Spillet åbnes og GUI vises<br>- Velkomst besked vises (spilleregler)<br>- Besked om indtastning af antal spillere vises<br>- Antal af spillere i tal indtastet i felt og afsluttes med klik på ok knap<br>- Besked om indtastning af spillernavne vises.<br>- Efter afsluttet indtastning af spillernavne har alle spillere fået en brik og en start balance på 30.000. |
| Faktisk resultat | - Spillet starter<br>- GUI vises med følgende besked: *"Reglerne i dette spil er som følger: Begge spiller starter med en balance på 1000. Den første til at opnå en balance på 3000 vinder spillet. Hvis begge spilleres balance er over 3000 i den runde, hvor mindst en spiller opnår 3000 vinder spilleren med flest point. Skulle balancerne være ens er spillet uafgjort. Der kastes med to terninger, terningernes udfald bestemmer feltet man lander på. Et felt har et navn, og en værdi som påvirker ens balance. Good luck, have fun.”*<br>- Efter klik på “ok” knap vises teksten: *“Hvor mange spillere? (mellem 2-6)”*<br>- Efter indtastning af “4” i feltet og der er klikket på “ok” knap vises teksten: *“Indtast spillernavn:”*<br>- Efter indtastet spillernavn “Dan” afsluttet med klik på “ok” vises en sort racerbils brik tilhørende Dan med en balance på 30.000.<br>- Igen vises beskeden *“Indtast navn:”* de resterende spillernavne indtastes med samme fremgangsmåde.<br>- Efter alle navne er indtastet vises en besked *“Det er Dan’s tur, Tryk OK for at kaste terningerne og rykke”.*<br>- Dertil vises også de resterende spilleres balancer og figurer. Mette har en grøn/gul racerbil, Mogens har en blå traktor, og David har en stribet UFO. |
| Status | Godkendt. Der tages ikke højde for om et spillernavn ender på s, der tilføjes altid “‘s” til en spillers navn i beskeden der viser hvilke spillers tur der er. |
| Testet af | Janus Andreasson |

| | |
|---|---|
| Dato | 23/11/2016 |
| Test miljø | Asus Laptop - OS: Windows 10 v. 1607 build 14393.447<br><br>DTU's WINdatabar (Se bilag 8).<br><br>Eclipse Java EE IDE for Web Developers.<br>Version: Neon Release (4.6.0)<br>Build id: 20160613-1800 |

*Tabel 7: Test*

| | |
|---|---|
| Test ID | TC02 |
| Beskrivelse | Tester use case 3: Spil tur, hvor spilleren trykker på 'OK' for at slå med terningerne og rykker tilsvarende x antal felter frem. |
| Krav | 1.2, 1.3 & 1.4 |
| Forudsætning | Adgang til computer med kompatibelt program |
| Efterfølgende betingelser | Spillet skal fortsætte indtil alle spillere på nær 1 er bankerot, dvs. at balance = 0 |
| Test procedure | 1. Åbn program<br>2. Kør programmet<br>3. Vælg antal spillere 2-6<br>4. Indtast navn på spillere<br>5. Tryk på OK for at rulle med terningerne<br>6. Kan bekræfte at man kan spille én tur. |
| Test data | 1. Åbnet Jar-fil (spil)<br>2. Klikkede på "ok" knap.<br>3. Indtastede 6 tekstfelt og klikkede på "ok" knap.<br>4. Indtastede følgende navne i tekstfelt: Tobias, Mikkel Nikolaj, Jonas, Janus, Justin , efter hvert navn blev der klikket ok før indtastning af det næste. |
| Forventede resultater | Vi forventer at:<br>1. Spillet åbnes<br>2. Spilleregler bliver præsenteret<br>3. Antal spillere kan vælges<br>4. Navne kan indtastes<br>5. Spillerbrik rykker felter frem tilsvarende til det adderede antal øjne på terningernes udfald. |
| Faktiske resultater | Når man spiller en tur:<br>1. Spillet åbnes<br>2. Spillereglerne præsenteres<br>3. Man kan indtaste antal spillere (2-6)<br>4. Man kan indtaste spillernavne |

| | |
|---|---|
| | 5. Man kan trykke på 'OK' knap for at slå med terningerne<br>6. Spillerbrik rykker felter frem tilsvarende til det adderede antal øjne på terningernes udfald. |
| Status | Godkendt |
| Dato | 23/11 - 2016 |
| Test miljø | Mac OS X El Capitan - version 10.11.6<br>Eclipse Java EE IDE for Web Developers<br>Version: Neon Release (4.6.0)<br>Build id: 20160613-1800 |
| Testet af | Tobias Tang Hansen |

*Tabel 8: Test*

| Test ID | TC03 |
|---|---|
| Beskrivelse | Denne test case tester hvorvidt spillebrættet registrerer at en spiller er landet på et felt og om spilleren får konsekvensen, som medfølger feltet. |
| Krav | 1.5, 1.5.1, 1.6, 1.7, 1.7.1, 1.7.2, 1.8, 1.8.1, 1.8.2, 1.9, 1.9.1, |
| Forudsætning | Spillet er startet op og antal spillere er valgt. |
| Efterfølgende betingelser | Der er observeret på spillers konto at den korrekte konsekvens er hændt ifølge tabellen i bilag 8. |
| Test procedure | 1. Spillet igangsættes, til denne test benyttes 2 spillere: X og Y.<br>2. Inden X og Y's kast nedskrives nuværende kontobalance, denne kaldes K_før. Balancen efter kastes udførelse noteres til K_eft.<br>3. Terningerne kastes nu, feltet identificeres, findes i tabellen i bilag 8 og der tjekkes nu om K_nu - K_eft = feltets værdi.<br>4. Feltets værdi skal være ens med den i bilag |
| Test data | |
| Forventede resultater | Forventet resultater ved:<br>- Landing på uejet felt (territorium, labor camp, fleet): spiller får mulighed for at takke ja eller nej til købet af feltet.<br>- Landing på ejet felt (territorium, labor camp, fleet): hvis spilleren selv ejer feltet bør ingenting ske, hvis ikke skal spilleren betale lejen til den rigtige ejer.<br>- Landing på 'refuge'-felt: spilleren får en bonus svarende til bilag<br>- Landing på 'tax'-felt: spilleren skal betale et beløb som |

| | |
|---|---|
| | afgift for at lande på et af de to 'tax'-felter. |
| Faktiske resultater | X landede på territorium feltet 'Sort Grotte' og på fleet feltet 'Piraterne' og fik muligheden begge gange for at købe disse.<br>Y lander på labor camp feltet 'Hytterne i bjergene' og fik muligheden for at købe det.<br>X landede på territorium feltet 'Graven', hvilket er ejet af Y, der blev overført 300 fra X's konto til Y's konto. Beløbet passer med den feltet er tildelt.<br>X landede på territorium feltet 'Stamme lejr', hvilket er ejet af X selv. GUI viser at "Spiller 'X' betaler 100 til 'X'", og der sker ingen ændring på X's konto.<br>X lander på refuge feltet 'Befæstede by' og X's konto stiger fra 8150 til 13150, hvilket passer til værdien feltet er tildelt.<br>Y landede på tax-feltet 'Karavanen' og får mulighed for at vælge mellem at betale 4000 eller 10% af hans formue. Der vælges 10% og Y's konto falder fra 17650 til 15885, hvilket svarer til et tab på 1765.<br>X lander på tax-feltet 'Karavanen' og får mulighed for at vælge mellem at betale 4000 eller 10% af hans formue. Der vælges 4000 og X's konto falder fra 4800 til 800.<br>Fleet<br>***X lander på Labor Camp feltet 'Graven', som er ejet af Y. GUI viser "Spiller 'X' betaler 0 til 'Y'.", men bør være 100 * slået terningesum (*2, hvis én spiller ejer begge labor camp felter).*** |
| Status | Delvist godkendt - labor camp feltet fejler som det eneste. Gruppen er blevet informeret om fejlen i funktionaliteten og idéer til løsninger bliver foretaget og diskuteret. |
| Dato | 23/11/2016 |
| Test miljø | Mac OS X El Capitan - version 10.11.2<br>Eclipse Java EE IDE for Web Developers<br>Version: Neon Release (4.6.0)<br>Build id: 20160613-1800 |
| Testet af | Mikkel Feng Frederiksen |

*Tabel 9: Test*

| | |
|---|---|
| Test ID | TC04 (fortsættelse af TC03) |
| Beskrivelse | Denne test case tester hvorvidt problematikken med Labor Camp felterne er løst og at spillet dermed følger kundens visioner omkring hvilke effekter felterne skal have på spillernes konti. |

| | |
|---|---|
| Test procedure | 1. Spillet spilles igennem med spillerne X og Y indtil at én af de to spillere lander på et labor camp ejet af den anden spiller<br>2. De to spilleres konti før og efter kastet af terninger noteres ned og bruges til analysen. |
| Forventede resultater | Det forventes at spiller X, som lander på spiller Y's Labor Camp felt får fratrukket 100 * det antal øjne X har slået eller det dobbelte af dette hvis spiller Y ejer begge Labor Camp felter. |
| Faktiske resultater | Spillet Y er landet på spiller X's Labor Camp felt, og der er blevet overført 600 til X's konto. Summen af terningerne er desuden vist på GUI til at være 6.<br>Spiller X er landet på spiller Y's Labor Camp felt (på dette tidspunkt ejer Y begge af disse) og det bliver overført 600 fra X til Y. GUI viser desuden at spiller X har slået en 1'er og en 2'er, hvilket dermed vil passe med kravet. Se også bilag |
| Status | Godkendt |
| Dato | 24/11/2016 |
| Test miljø | Mac OS X El Capitan - version 10.11.2<br>Eclipse Java EE IDE for Web Developers<br>Version: Neon Release (4.6.0)<br>Build id: 20160613-1800 |
| Testet af | Mikkel Feng Frederiksen |

*Tabel 10: Test*

| | |
|---|---|
| Test ID | TC05 |
| Beskrivelse | Denne test case, tester om spillere starter med en pengebeholdning på 30.000 og at spillet skal afgøres hvis alle undtagen én spiller er bankerot. Man går bankerot når spiller's pengebeholdning = 0. Endvidere bliver der også testet for om ens balance kan blive negativ. |
| Krav | 1.10, 1.11 |
| Forudsætning | Adgang til computer med kompatibelt program |
| Efterfølgende betingelser | - Spillere skal starte med pengebeholdning på 30.000<br>- Spillet skal fortsætte indtil alle spillere på nær 1 er bankerot, dvs. at balance = 0<br>- Derudover kan pengebeholdning ikke blive negativ. |

| | |
|---|---|
| Test procedure | 1. Spillet igangsættes ved at trykke Run Main<br>2. Information om spillet printes på GUI'en<br>3. Antal spillere vælges<br>4. Spillernavne indtastes<br>5. Spiller eksekvere spillet ved at trykke på OK for at kaste med terningerne<br>6. Spiller træffer valg i løbet af spillet jv. spilleregler ved tryk på OK og JA/NEJ<br>7. Spiller fortsætter indtil kun én spiller har pengebeholdning tilbage. |
| Test data | - Åbnet Jar-fil (spil)<br>- Klikkede på "ok" knap.<br>- Indtastede 2-6 tekstfelt og klikkede på "ok" knap.<br>- Indtastede følgende navne i tekstfelt: Tobias, Mikkel Nikolaj, Jonas, Janus, Justin , efter hvert navn blev der klikket ok før indtastning af det næste. |
| Forventede resultater | Vi forventer at:<br>1. Spillere har 30.000 i pengebeholdning ved 2-6 spillere<br>2. Spillet fortsætter indtil kun én spiller har pengebeholdning ved 2-6 spillere<br>3. At pengebeholdningen ikke kan blive negativ |
| Faktiske resultater | - Spillere har 30.000 som pengebeholdning ved 2 spillere<br>- Spillere har 30.000 som pengebeholdning ved 3 spillere<br>- Spillere har 30.000 som pengebeholdning ved 4 spillere<br>- Spillere har 30.000 som pengebeholdning ved 5 spillere<br>- Spillere har 30.000 som pengebeholdning ved 6 spillere<br>- Spillet afsluttes når kun én spiller har pengebeholdning < 0 ved 2 spiller<br>- Spillet afsluttes når kun én spiller har pengebeholdning < 0 ved 3 spiller<br>- Spillet afsluttes når kun én spiller har pengebeholdning < 0 ved 4 spiller<br>- Spillet afsluttes når kun én spiller har pengebeholdning < 0 ved 5 spiller<br>- Spillet afsluttes når kun én spiller har pengebeholdning < 0 ved 6 spiller<br>- Pengebeholdning kan ikke blive negativ. |
| Status | Godkendt |
| Dato | 24/11 - 2016 |
| Test miljø | Mac OS X El Capitan - version 10.11.6<br>Eclipse Java EE IDE for Web Developers<br>Version: Neon Release (4.6.0)<br>Build id: 20160613-1800 |
| Testet af | Tobias Tang Hansen |

*Tabel 11: Test*

## 9.3 Traceability matrix

De funktionelle krav er blevet testet og indskrevet i en matrix visende hvilke test cases, der dækker over hvilke krav. Det ses at TC4 tester 2 af kravene også testet i TC3. Dette er sket, da TC3 fejlede testen af Labor Camp feltet, som ikke gav den rigtige effekt på en spillers balance, hvis han landede på et Labor Camp felt ejet af en modspiller. Fejlen blev spottet til at være et kald på en forkert metode i klassen plus et par andre småting, hvilket ikke var det sværeste at gøre bod på.

### 9.3.1 Test tracing

| **Krav** | **TC1** | **TC2** | **TC3** | **TC4** | **TC5** |
|---|---|---|---|---|---|
| 1.1 | **X** | | | | |
| 1.2 | | **X** | | | |
| 1.3 | | **X** | | | |
| 1.4 | | **X** | | | |
| 1.5 | | | **X** | | |
| 1.5.1 | | | **X** | | |
| 1.6 | | | **X** | | |
| 1.7 | | | **X** | | |
| 1.7.1 | | | **X** | | |
| 1.7.2 | | | **X** | | |
| 1.8 | | | **X** | | |
| 1.8.1 | | | **X*** | **X**** | |
| 1.8.2 | | | **X*** | **X**** | |
| 1.9 | | | **X** | | |
| 1.9.1 | | | **X** | | |
| 1.10 | **X** | | | | **X** |
| 1.11 | | | | | **X** |

*Tabel 12: Traceability matrix*

* Fejlede tests

** Gentestning af fejlede tests

## 9.4 JUnit

Vi har benyttet JUnit til at teste vores mest omfattende metoder i de følgende klasser:

**TestDie klassen**
I denne klasse tester vi om hvorvidt Roll() metoden returnere et statisktisk sandsynligt udfald af værdier mellem 1-6. Det forventes over 60000 kast vil fordelingen af hvert muligt udfald være ~10000, der er taget forbehold for en mulig afvigelse på 4%.

Dernæst køres en test af die.roll() som gør det muligt, at se om der er et mønster ved terningekast.

**TestDiceCup klassen**
Denne klasse tester om fordelingen ved $1.1 * 10^6$ diceCup.rollDiceCup() har en teoretisk korrekt fordeling, igen er det taget forbehold for en mulig afvigelse på 4%. Se bilag 6 for gennemgang af den teoretiske værdi.

**TestFleet klassen**
Der udføres 5 tests i denne klasse, en test af hvert mulig udfald fra den nedarvede metode LandOnField i fleet klassen hhv. hvis en spiller ejer 1, 2, 3 eller 4 fleets.
Det forventes at renten ændre sig i forhold til antallet af fleets som en givet spiller ejer.

BuyField() metoden testes også og bekræfter om en spiller er i stand til at købe et felt.

**TestLaborCamp klassen**
Denne klasse tester om den nedarvede metode LandOnField() giver den korrekte effekt på balancen. LandOnField er delt i to tests, som tester hhv. hvis en spiller lander på et LaborCamp felt hvor ejeren kun ejer et felt af denne felttype, og den næste hvor ejeren ejer begge felter.

Det skal siges at denne klasses test i starten fejlede (se bilag 7), kildekoden er derefter blevet rettet sådan at felttypen LaborCamp fungerer korrekt.

**TestRefuge klassen**
Tre tests køres i denne klasse alle test tester hvordan LandOnField() metoden har effekt på balancen. Der testes om en spiller modtager hhv. 500 og 5000, ved at lande felt instanser som matcher kundens version af spillet. Den sidste test ser om det er muligt at påvirke balancen negativt ved at angive en instans af felttypen Refuge med en negativ “bonus”.

**TestTax klassen**
Omfatter to test som tester hvorvidt LandOnField() og den overloadede version af LandOnField fungere som forventet. LandOnField() skal beskatte en spiller med 4000. Den overloadede version beregner det korrekte beløb ud fra feltets tax, og beskatter spilleren. Det forventes at en spiller skal betale 10% af sine samlede formue.

**TestTerritory klassen**
Vi tester samtlige metoder i Territory klassen; Der testes om getter metoder til Rent, Price og Type returnere de formodet værdier. Der testes tre instanser af nedarvningen af LandOnField() metoden. To af instanserne er Rent sat til positive beløb, den sidste til et negativt beløb. Det er interessant at teste med et negativt beløb, da det er muligt at flytte penge fra en felt-ejers konto til spilleren som lander på feltets konto vil dette kunne bruges til videreudvikling af andre spil. Se bilag 10.

# 10 Versionsstying

Under udviklingsprocessen af programmet, har vi benyttet GIT. Via git repositories kan vi have en lokal kopi af branchen kaldt “development”. Når en collaborator så har lavet ændringer i koden, pushed ændringerne til det repository som ligger online. Derved vil alle kunne hente den nyeste version af koden og videreudvikle på den.

Da koden var færdigudviklet, refaktoreret, testet etc. pullede vi koden fra development branchen til hoved branchen master.

## 10.1 Import af projekt fra Git til Eclipse

Link til projekt: https://github.com/jufa2401/CDIO_Del_3/tree/master

Man kan let importere et projekt fra Git til Eclipse.

1. Åbn eclipse
2. Tryk på file → Import → Git → Projects from Git → Clone URI
    a. Gå ind på GitHub
    b. Find projektet
    c. Tryk på “Clone or download”
    d. Kopier URL’en
3. Indsæt URL’en i Eclipse
4. Vælg et tomt directory
5. Importer projektet.

*note: Hvis GIT repository’et er privat, skal man være logget ind, og være collaborator før man kan hente projektet.*

Ved hjælp af Github kan vi uploade og se hinandens ændringer (versioner) af programmet og downloade hinandens kode til vores eget repository. For at lave og downloade direkte fra ens lokale repository skal man være logget ind med sit GitHub login, gennem eclipse.

# 11 Konfigurationsstyring

Udviklingsplatform og produktionsplatformen er gennem dette projekt den samme. Da vi hver især arbejder på forskellige computere, benyttes flere operativsystemer. De to operativsystemer som benyttes, er:

- Windows 10 version 1607 (samt version 1511 på computere i databaren)
- OS X El Capitan version 10.11.6.

Vi benytter Eclipse Neon som vores udviklingsværktøj med versionsnummer 4.6.0. Det programmeringssprog, som benyttes i Eclipse Neon hedder Java SE Development kit 8u111.

## Kør programmet

Her er en vejledning til hvordan man starter programmet. Hvis man ikke allerede har projektet importeret til sit eclipse bedes man følge instruktionerne under afsnittet versionsstyring.

1. Åbn eclipse
2. Find Java Projekt-mappen
3. Highlight “src” mappen ved at venstreklikke på den.
4. Find knappen “Run main” i toolbaren
5. Tryk på “Run main”

Nu burde programmet køre og spillet kan spilles.

# 12 Konklusion

Med formålet at fremstille, teste og dokumentere et matador lignende brætspil, har vi nu gennemgået kravspecifikationen metodisk og sikret os at den bliver overholdt. Vi har skabt et fungerende program med en velfungerende kode, vi har dokumenteret og illustreret hvordan programmet virker ved hjælp af forskellige slags diagrammer og vi har sikret os at vores program virker ordentlig, ved at teste det med flere forskellige metoder som f.x JUnit testen og formelle test.

## 12.1 Refleksioner

Koden kan refaktoreres yderligere så unødvendige metoder i nogle klasser bliver fjernet. fx. nedarves getPrice() til Refuge klassen fra den abstrakte superklasse Field.

Det kan vurderes om hvorvidt klasserne som indeholder felterne ikke skal være entities, men i stedet fungere som deres egen kontrollere, derved vil de kunne kontakte GUIHandleren direkte.

I vores program skelner vi vores felttyper fra hinanden ved give hvert felttype et type id, her kunne vi have brugt *instanceof*. Derudover tester vi også om et felt er ownable ved at kigge på price, vi kunne her også have brugt *instanceof* til at checke om en klasse var en ownable.

Vi var opsat på at adskille vores Boundary og Entitets-klasser. Som konsekvens af dette endte vi ud med at have kald til GUI'en i både Controller og GameLogic. Det medfører også (se BCE fra tidligere) at både Controller og GameLogic har associationer til de samme underklasser(Player, Field, LanguageDefinitions). Dette kunne tale for, at Controller og GameLogic samles i en, ellers designe om for at komme af med dobbeltassociationerne.

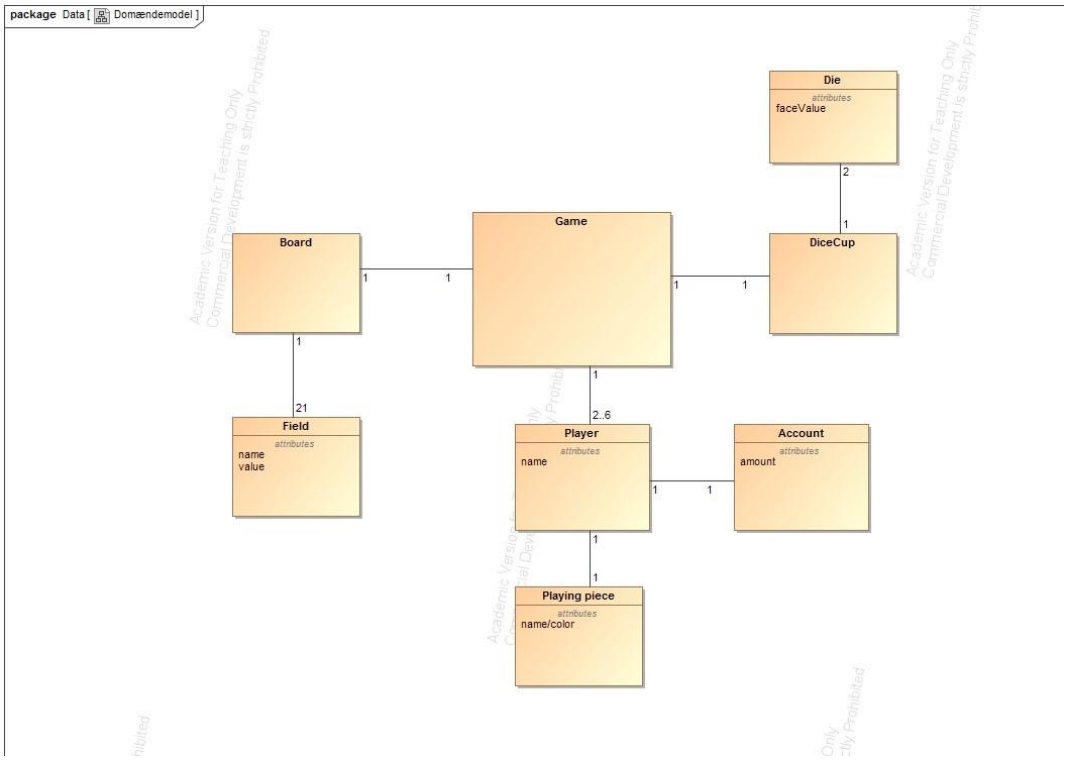
# 13 Litteraturliste


1. *Applying UML and patterns: An introduction to object-oriented analysis and design and iterative development, third edition - Larman, Craig - October 20th, 2004 - Addison Wesley Professional.*
2. *http://stackoverflow.com/ - 2008 - CEO Joel Polsky*
3. *Kode fra: Tilgængelig på CN -> Lektion 10 -> DIV-> 06 CDIO del1 Full 2014*

# 14 Bilag

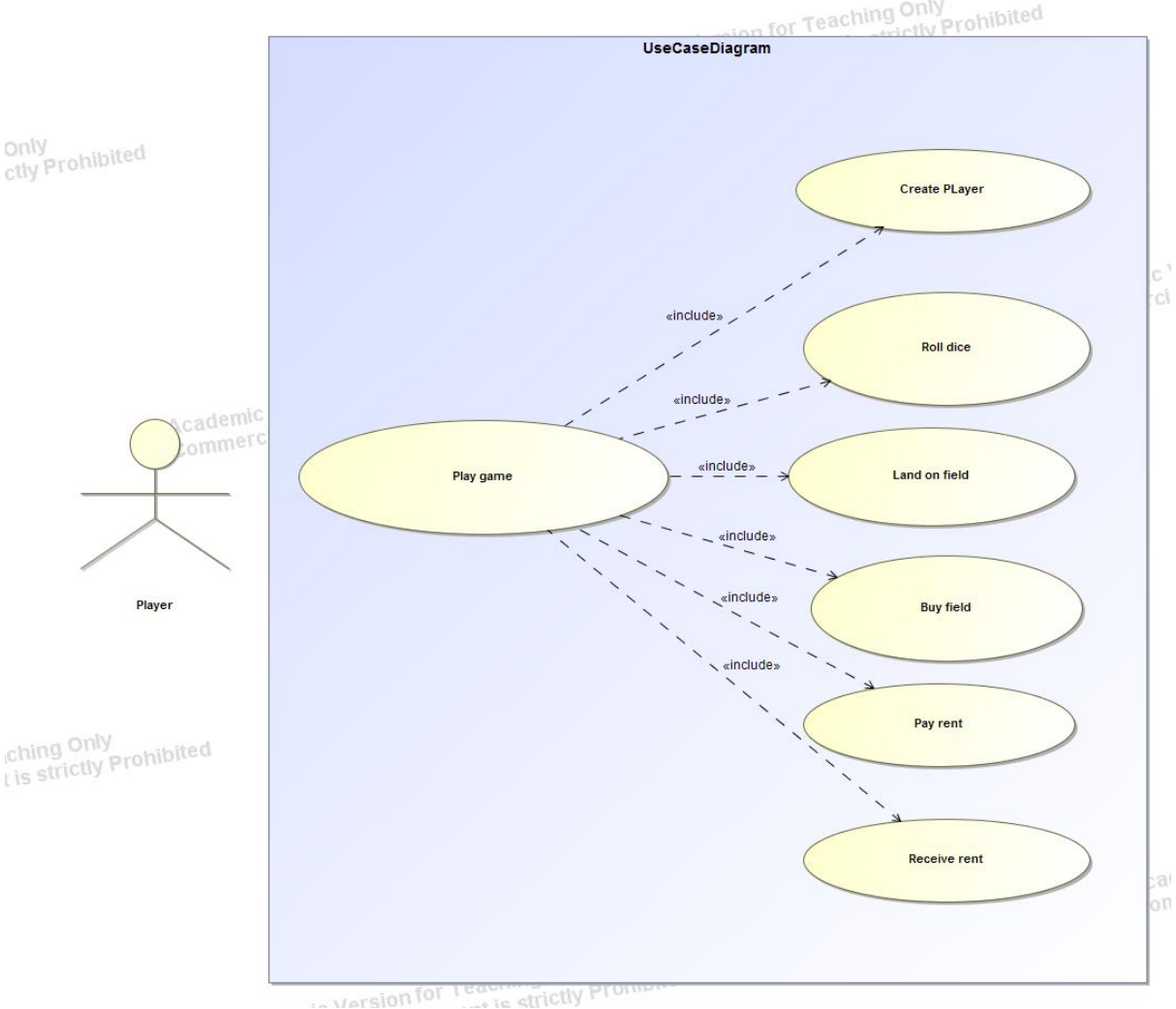
## 14.1 Bilag 1

Første iteration af domænemodellen



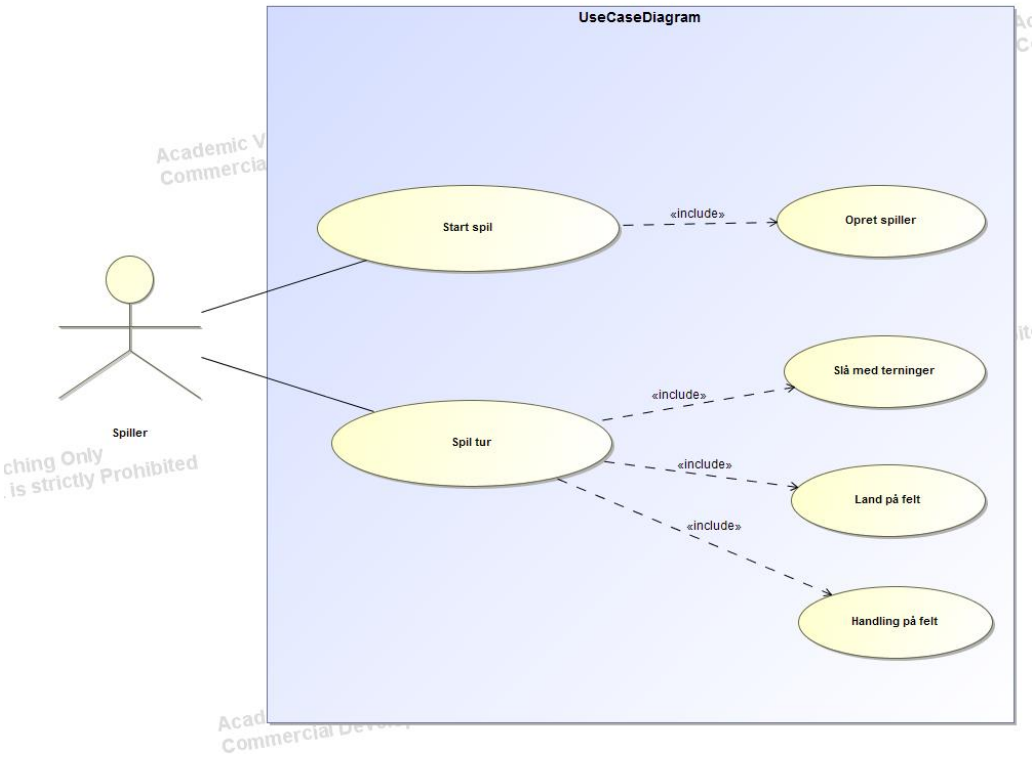
## 14.2 Bilag 2

Første iteration af Use Case diagrammet:



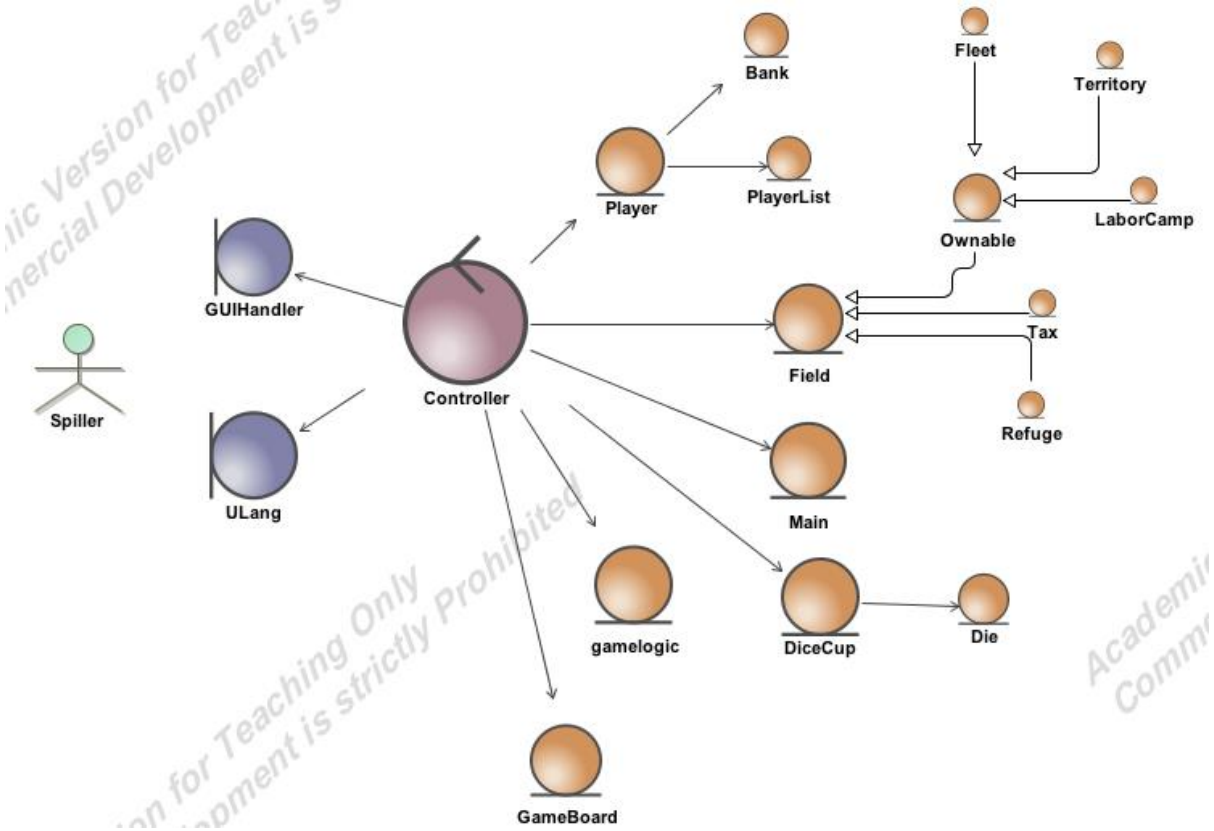
## 14.3 Bilag 3

Anden iteration af Use Case diagrammet:

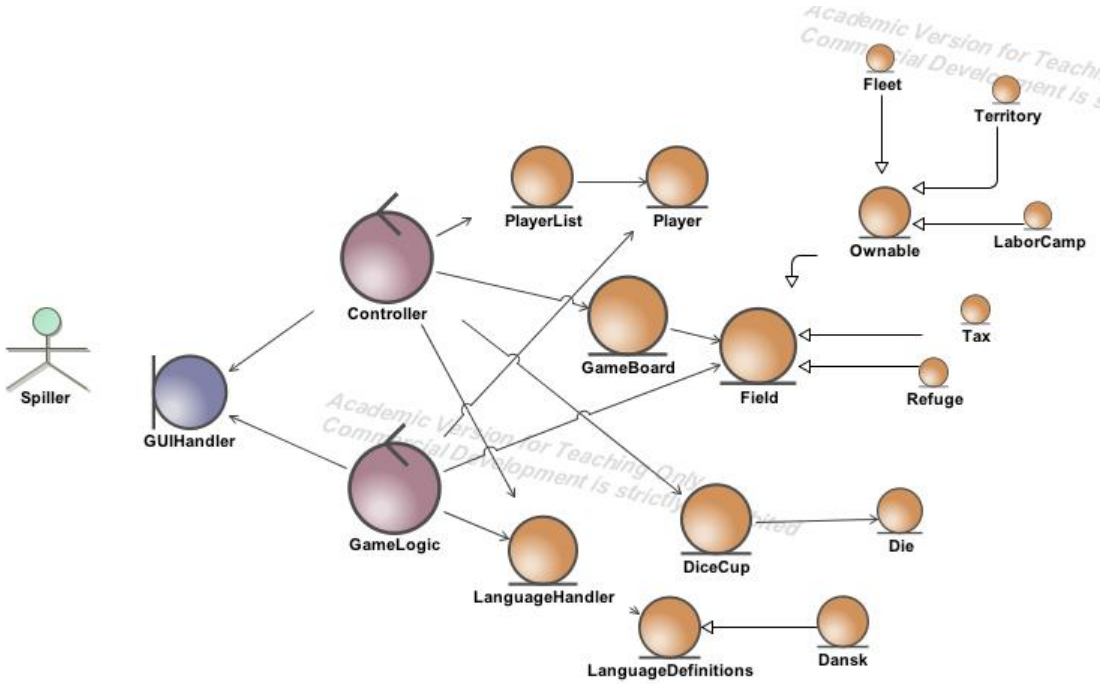


# 14.4 Bilag 4

BCE model første iteration.




BCE model med 2 controllers.

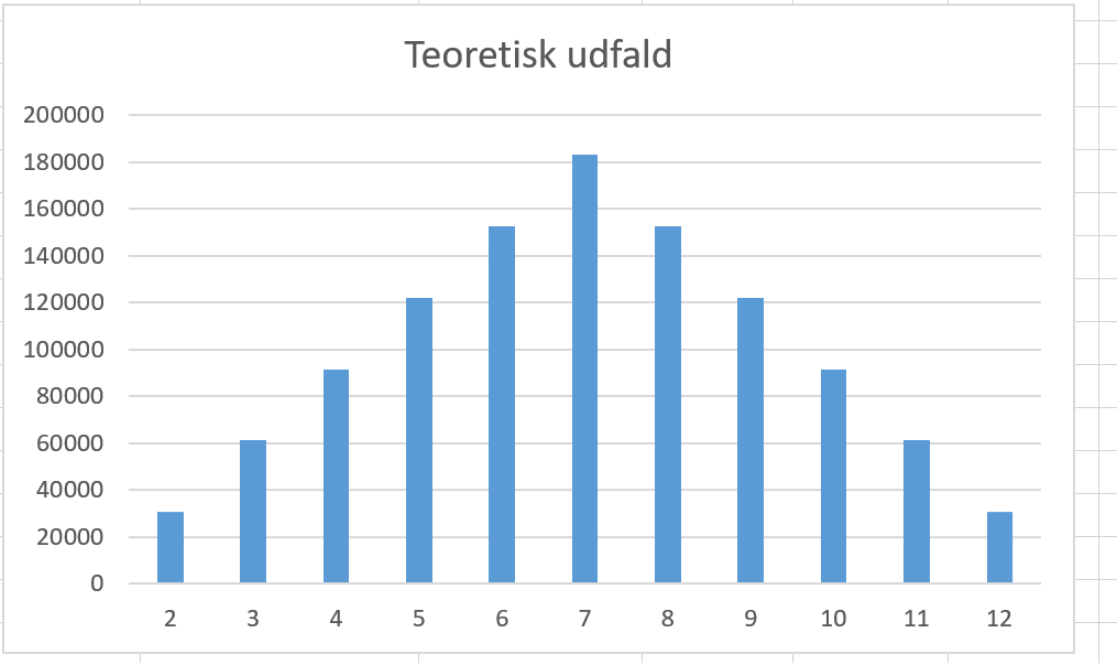


## 14.5 Bilag 5

Forklaring af bilag:

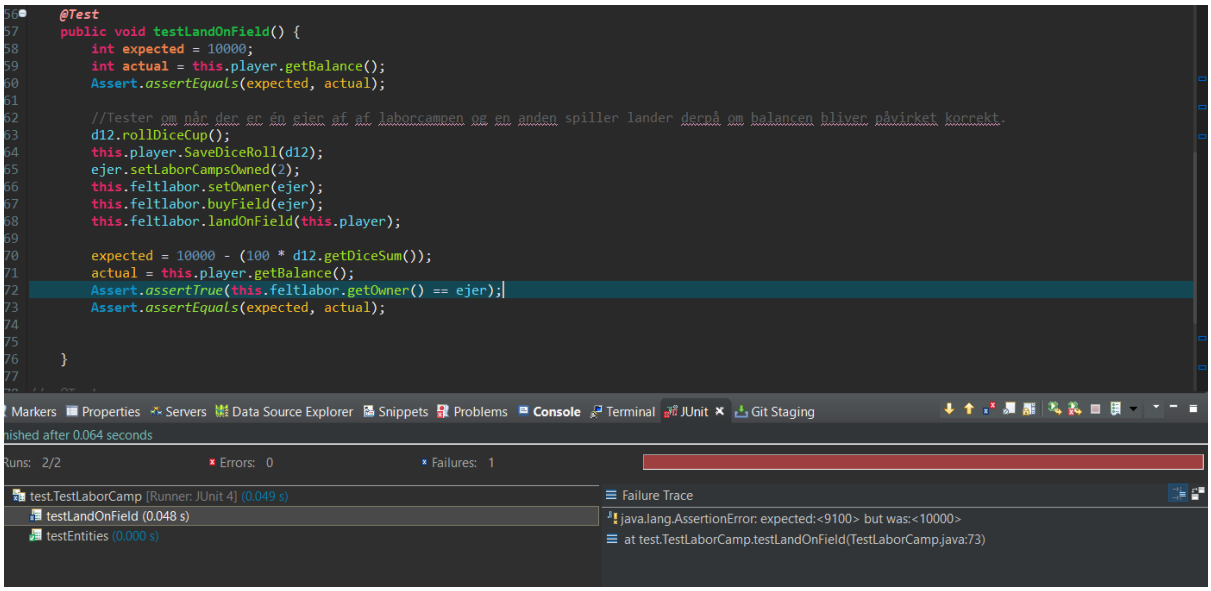
- Øjnenes værdi er terningernes sum af de mulige udfald
- Det teoretiske udfald i % er fundet på nettet, og dækker den procentdel man kan forvente i udfald.
- Teoretisk udfald (antal) er det teoretiske udfald på baggrund af 110.000 kast.
- Upperlimit: Vores accept af, at de forskellige udfald må afvige med 4% over det forventede udfald:
- Lowerlimit: Vores accept af, at de forskellige udfald må afvige med 4% under det forventede udfald.

| Øjnenes værdi | Teoretisk udfald (%) | Teoretisk udfald (antal) | Upperlimit | Lowerlimit |
|---|---|---|---|---|
| 2 | 2,78 | 30580 | 31803,2 | 29356,8 |
| 3 | 5,56 | 61160 | 63606,4 | 58713,6 |
| 4 | 8,33 | 91630 | 95295,2 | 87964,8 |
| 5 | 11,11 | 122210 | 127098,4 | 117321,6 |
| 6 | 13,89 | 152790 | 158901,6 | 146678,4 |
| 7 | 16,67 | 183370 | 190704,8 | 176035,2 |
| 8 | 13,89 | 152790 | 158901,6 | 146678,4 |
| 9 | 11,11 | 122210 | 127098,4 | 117321,6 |
| 10 | 8,33 | 91630 | 95295,2 | 87964,8 |
| 11 | 5,56 | 61160 | 63606,4 | 58713,6 |
| 12 | 2,78 | 30580 | 31803,2 | 29356,8 |

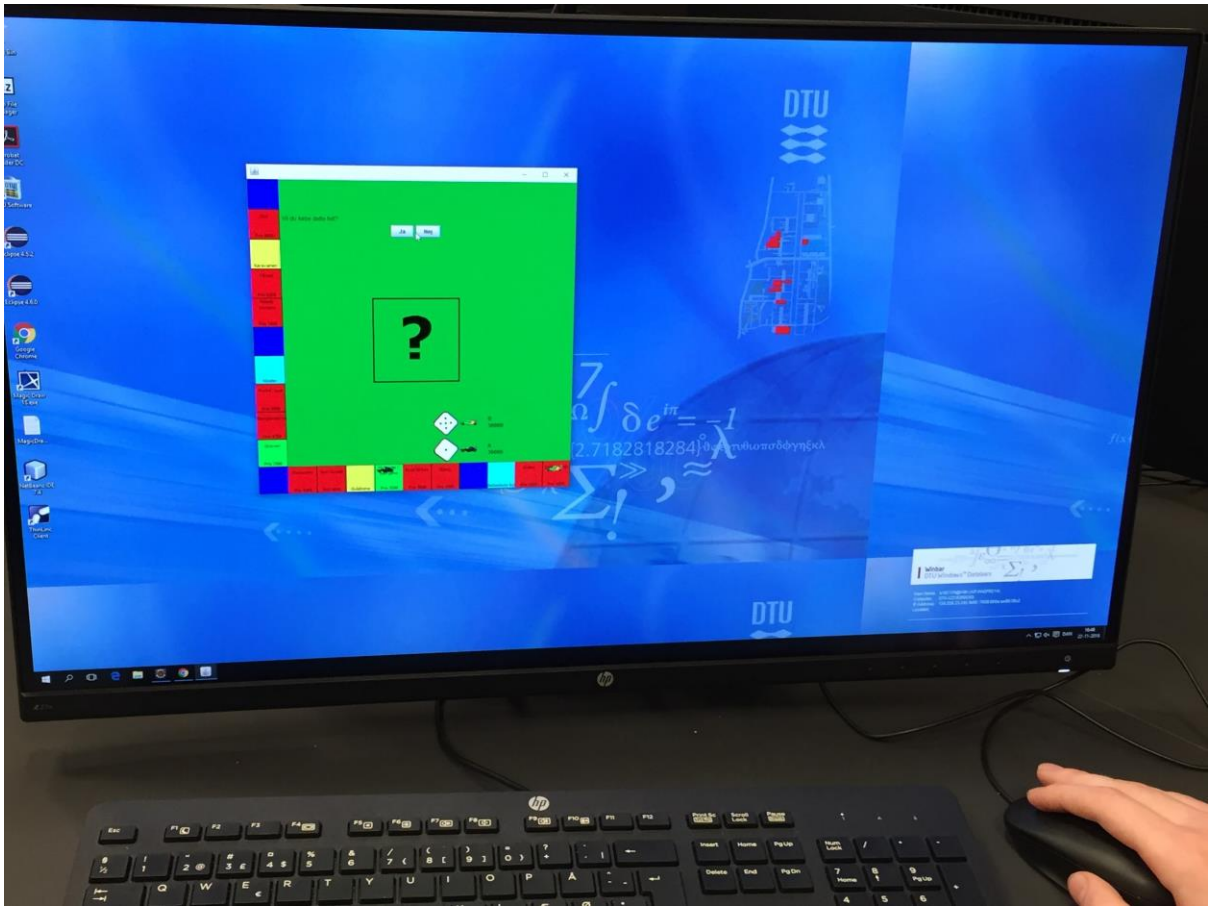


## 14.6 Bilag 6

Junit test af LaborCamp klassen inden koden blev konfigureret.
Det kan ses at ejeren af feltet er sat til “ejer”, dette bekræftes at den highlightede Assert…. metode. Fejlen lå på daværende tidspunkt i en switch statement i klassen LaborCamp.



## 14.7 Bilag 7 - Dokumentering af spillet på DTUs databar i 303A

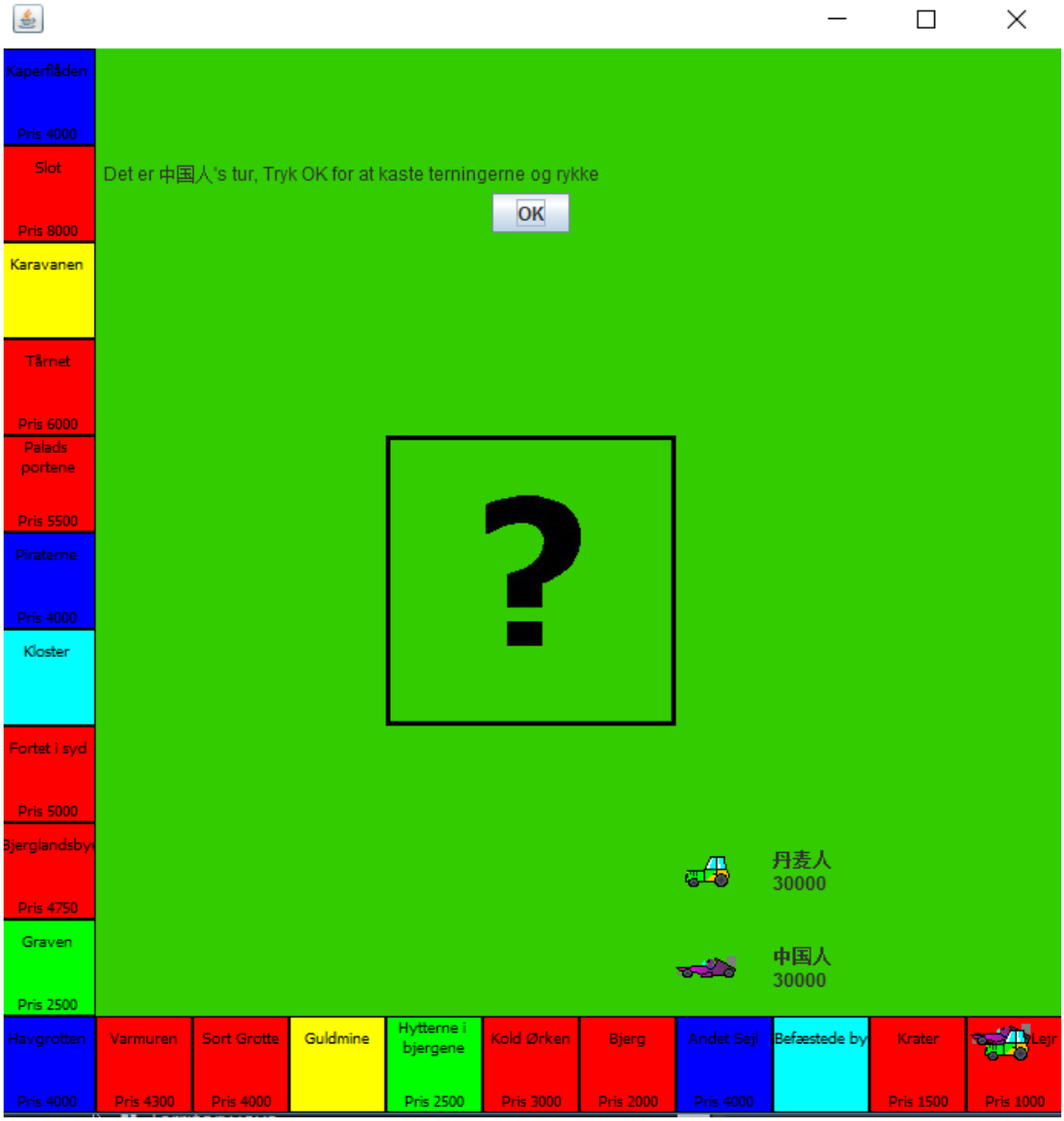


## 14.8 Bilag 8 : Feltliste og disses effekter på balance

**Feltliste**

| | | | | |
|---|---|---|---|---|
| 1. | Stamme lejr | Territorium | Leje 100 | Pris 1000 |
| 2. | Krater | Territorium | Leje 300 | Pris 1500 |
| 3. | Befæstede by | Helle | Modtag 5000 | |
| 4. | Andet sejl | Flåde | Betal 500-4000 | Pris 4000 |
| 5. | Bjerg | Territorium | Leje 500 | Pris 2000 |
| 6. | Kold ørken | Territorium | Leje 700 | Pris 3000 |
| 7. | Hytterne i Bjergene | Arbejdslejr | Leje 100 x øjne | Pris 2500 |
| 8. | Guldmine | Skat | Betal 2000 | |
| 9. | Sort Grotte | Territorium | Leje 1000 | Pris 4000 |
| 10. | Varmuren | Territorium | Leje 1300 | Pris 4300 |
| 11. | Havgrotten | Flåde | 500-4000 | Pris 4000 |
| 12. | Graven | Arbejdslejr | Betal 100 x øjne | Pris 2500 |
| 13. | Bjerglandsbyen | Territorium | Leje 1600 | Pris 4750 |
| 14. | Fortet i syd | Territorium | Leje 2000 | Pris 5000 |
| 15. | Kloster | Helle | Modtag 500 | |
| 16. | Piraterne | Flåde | Betal 500-4000 | Pris 4000 |
| 17. | Palads portene | Territorium | Leje 2600 | Pris 5500 |
| 18. | Tårnet | Territorium | Leje 3200 | Pris 6000 |
| 19. | Karavanen | Skat | Betal 2000 | |
| 20. | Slot | Territorium | Leje 4000 | Pris 8000 |
| 21. | Kaperflåden | Flåde | Betal 500-4000 | Pris 4000 |

## 14.9 Bilag 9 - Grey box testing med kinesiske tegn som input



## 14.10 Bilag 10 - JUnit test af Territory klasse. (1 af 3)

```java
import static org.junit.Assert.*;

public class TestTerritory {

    private Player player;
    private Territory territory1000;
    private Territory territory4000;
    private Territory territoryNegative500;
    Player ejer = new Player("ejer", 5000);

    @Before
    public void setUp() throws Exception {
        this.player = new Player("Doland Dak", 10000);
        //Player ejer = new Player("ejer", 1000);
        this.territory1000 = new Territory(1, Color.black, 2000, 1000);
        this.territory4000 = new Territory(2, Color.blue, 6000, 4000);
        this.territoryNegative500 = new Territory(3, Color.yellow, 100, -500);
        this.territory1000.setOwner(ejer);
        this.territory4000.setOwner(ejer);
        this.territoryNegative500.setOwner(ejer);
    }

    @After
    public void tearDown() throws Exception {
        this.player = new Player("Doland Dak", 10000);

    }

    @Test
    public void testEntity() {
        Assert.assertNotNull(this.player);
        Assert.assertNotNull(this.territory1000);
        Assert.assertNotNull(this.territory4000);
        Assert.assertNotNull(this.territoryNegative500);

        Assert.assertTrue(this.territory1000 instanceof Territory);
        Assert.assertTrue(this.territory4000 instanceof Territory);
        Assert.assertTrue(this.territoryNegative500 instanceof Territory);
    }
```

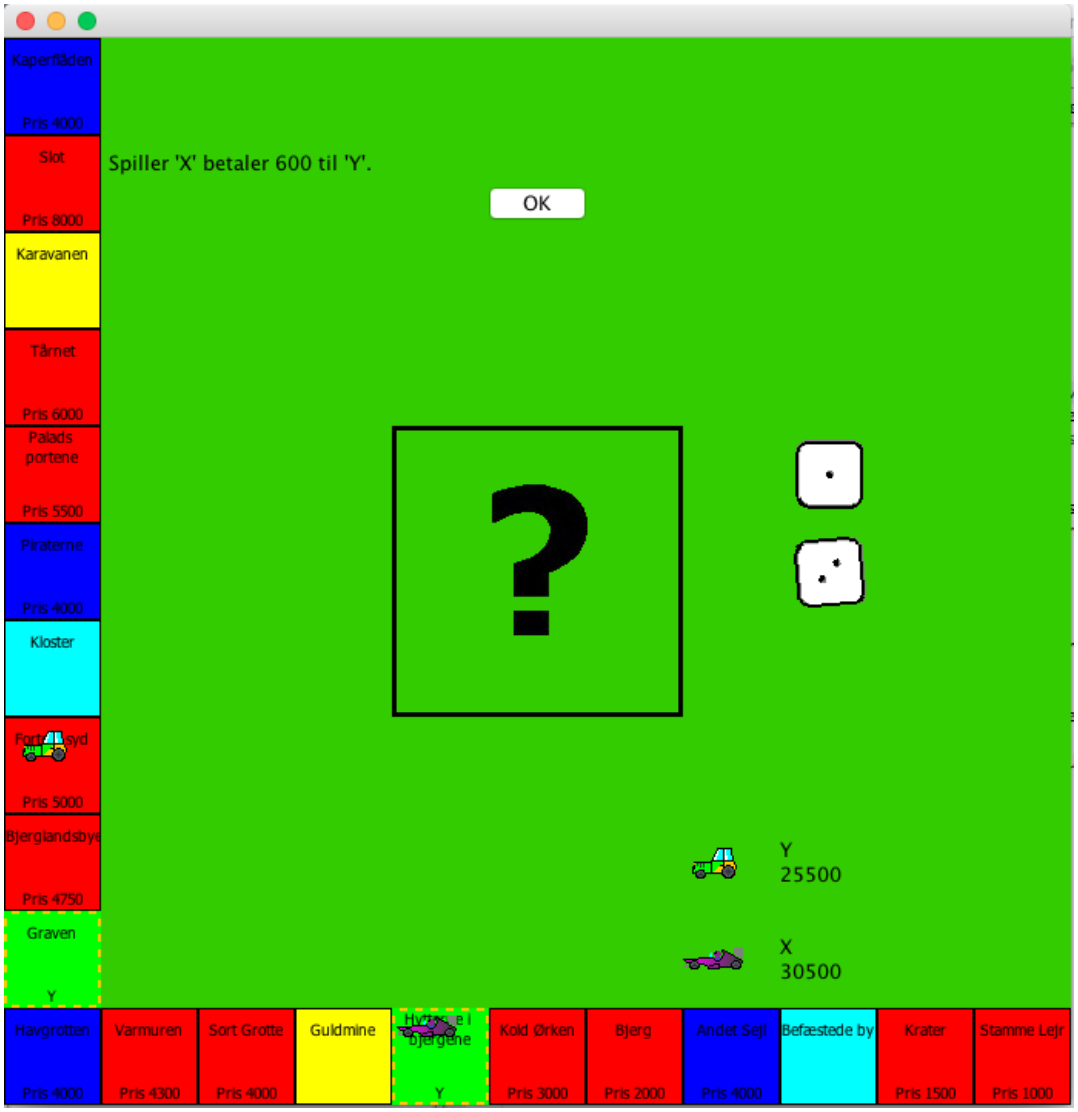
(2 af 3)

```java
    @Test
    public final void testGetRent() {
        int expected = 1000;
        int actual = this.territory1000.getRent();
        Assert.assertEquals(expected, actual);

    }

    @Test
    public final void testGetPrice() {
        int expected = 2000;
        int actual = this.territory1000.getPrice();
        Assert.assertEquals(expected, actual);
    }

    @Test
    public final void testGetType() {
        int expected = 5;
        int actual = this.territory1000.getType();
        Assert.assertEquals(expected, actual);
    }

    @Test
    public final void testLandOnField1000() {
        int expected = 10000;
        int actual = this.player.getBalance();
        Assert.assertEquals(expected, actual);

        //Tester LandOnField metoden og om den påvirker balances som forventet.
        this.territory1000.landOnField(this.player);

        expected = 10000 - 1000;
        actual = this.player.getBalance();
        Assert.assertEquals(expected, actual);
        }
```

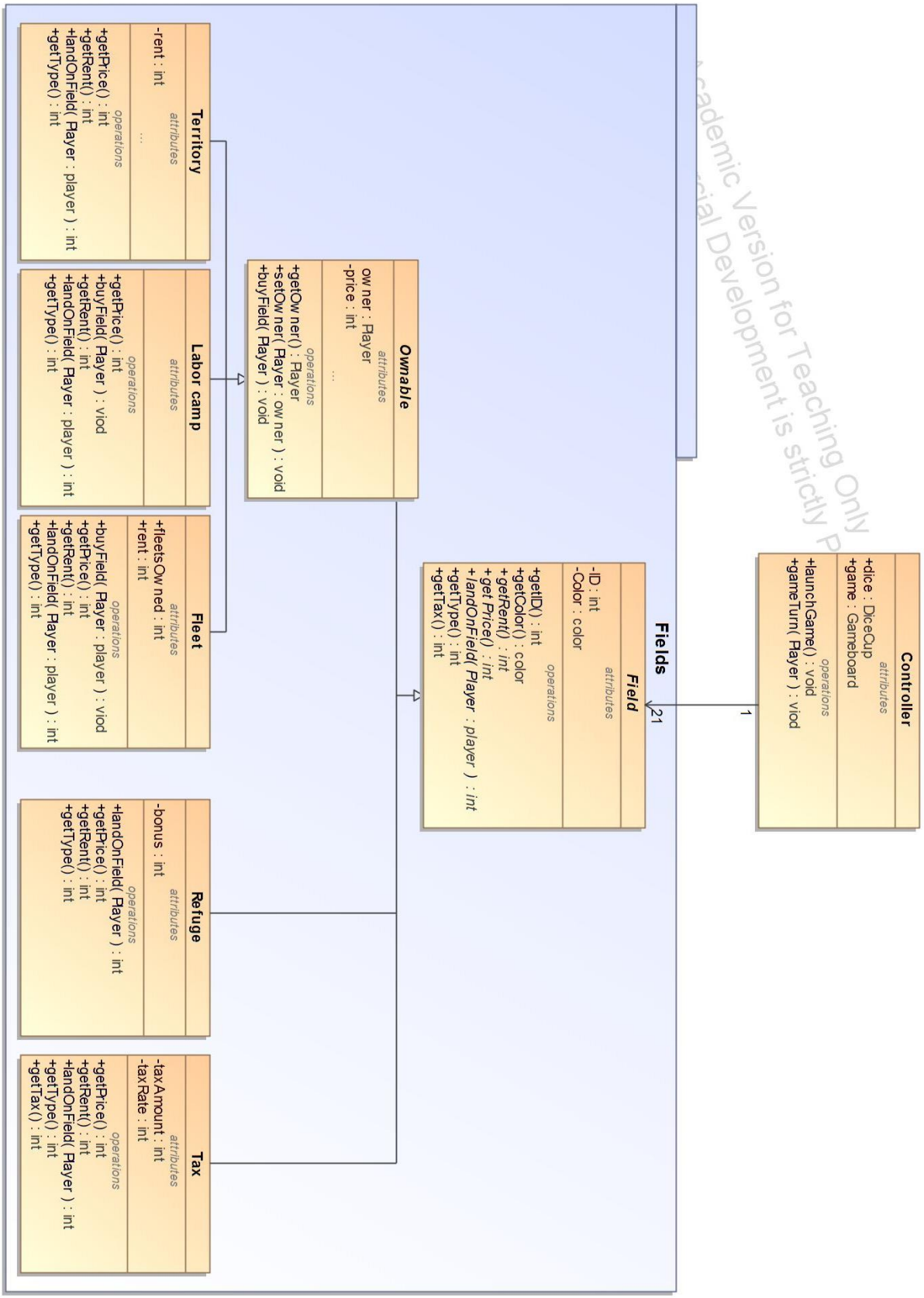
(3 af 3)

```
87
88      @Test
89      public final void testLandOnField2000() {
90          int expected = 10000;
91          int actual = this.player.getBalance();
92          Assert.assertEquals(expected, actual);
93
94          //Tesert udfaldet af LandOnField metoden i Territory og hvordan balancen påvirkes.
95          this.territory4000.landOnField(this.player);
96
97          expected = 10000 - 4000;
98          actual = this.player.getBalance();
99          Assert.assertEquals(expected, actual);
100     }
101
102     @Test
103     public final void testLandOnFieldNegative500() {
104         int expected = 10000;
105         int actual = this.player.getBalance();
106         Assert.assertEquals(expected, actual);
107
108         //tester om hvordan rent metoden fungere sammen med player klassen.
109         // tester om en spiller modtager penge når han skal betale et negativt beløb - Overholder klassen matematiske regler (-- = +).
110         this.territoryNegative500.landOnField(this.player);
111
112         expected = 10000 + 500;
113         actual = this.player.getBalance();
114         Assert.assertEquals(expected, actual);
115
116         //tester om ejeren af feltet er blevet trukkt penge.
117         expected = 5000 - 500;
118         actual = ejer.getBalance();
119         Assert.assertEquals(expected, actual);
120     }
121
122 }
```

## 14.11 Bilag 11: TC4

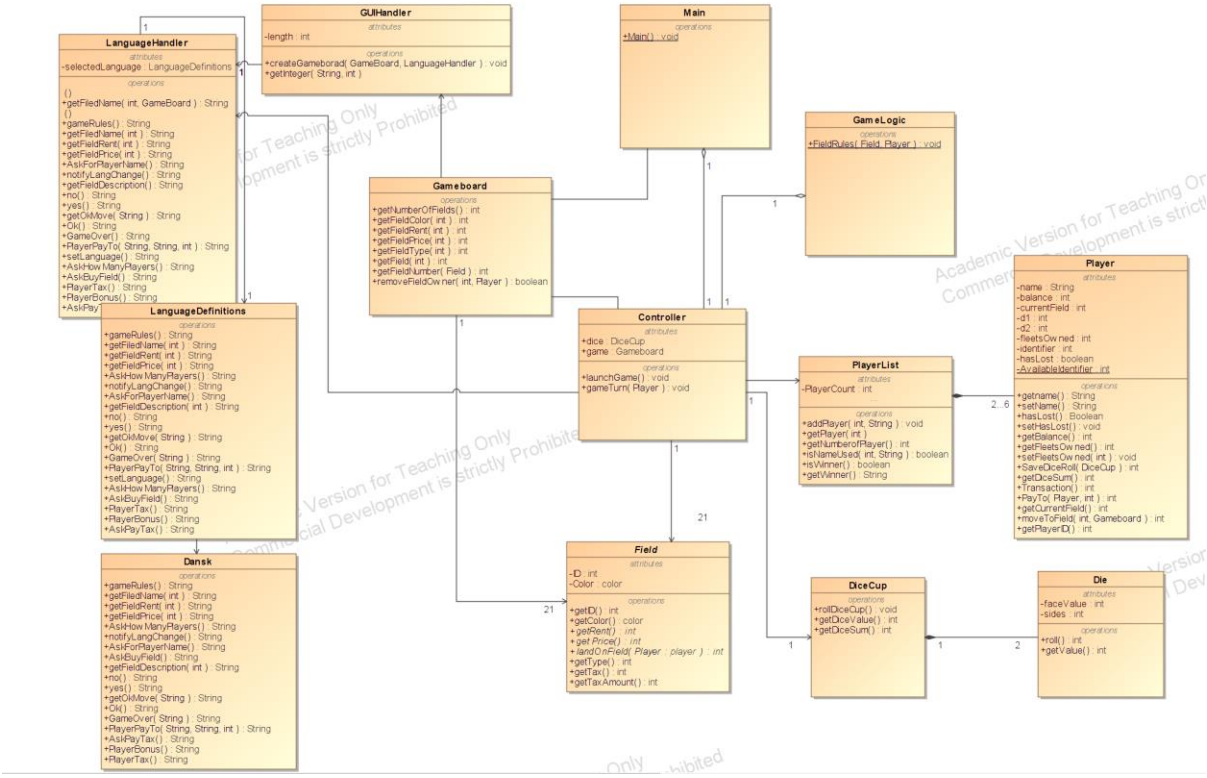


14.12 Bilag 12: En af de tidligere iterationer af arvehierarkiet *Field:*



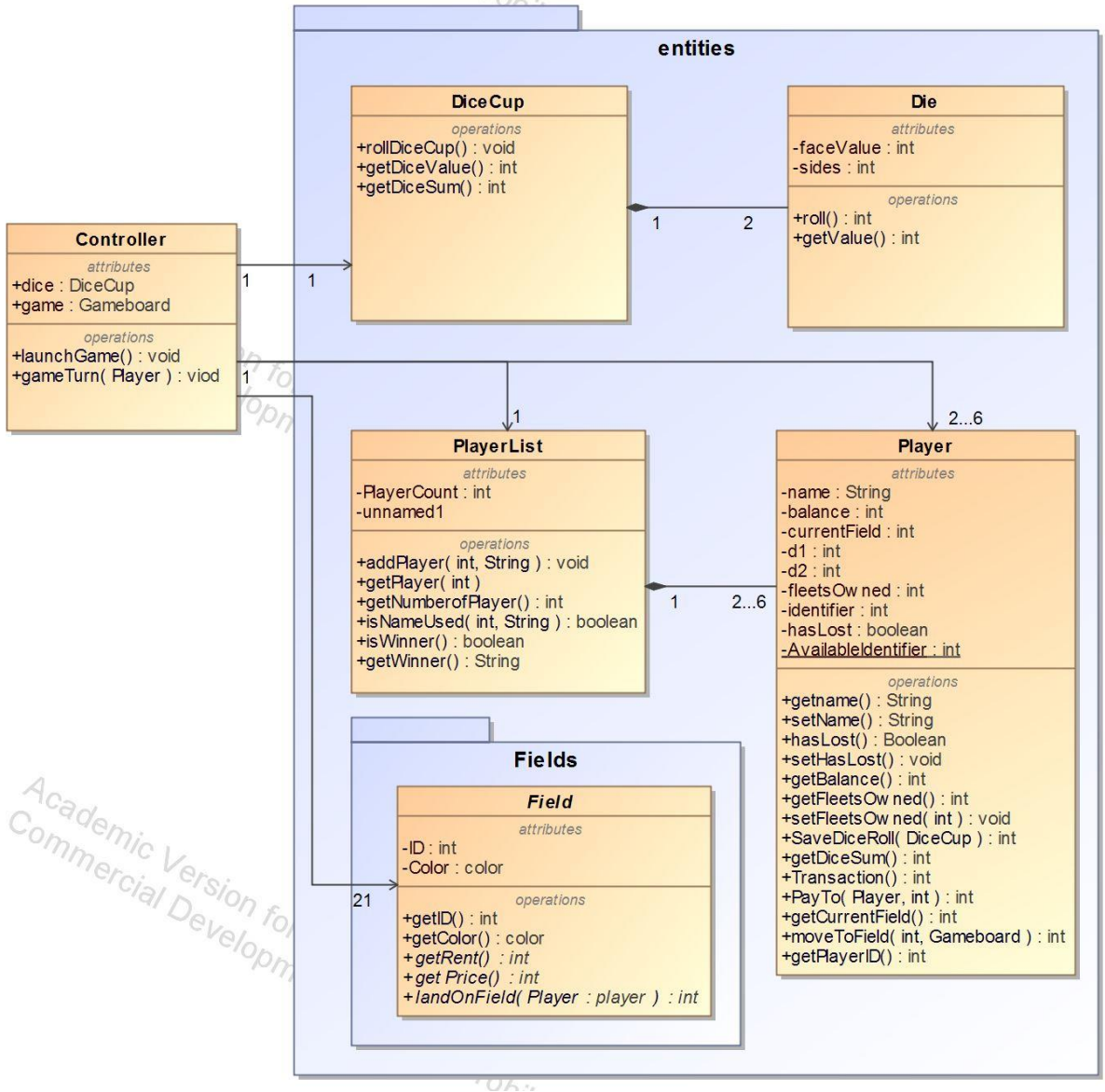
## 14.13 Bilag 13

En tidligere iteration af design klassediagrammet før det blev delt op i pakker.



## 14.14 Bilag 14

En af de tidligere iterationer af entitets-pakken



## 14.15 Bilag 15

Tredje iteration af use case diagrammet