Projektopgave efterår 2013 - jan 2014
02312-14 Indledende programmering
02313 Udviklingsmetoder til IT-Systemer.
Projektnavn: CDIO delopgave 3
Gruppe nr: 51.
Afleveringsfrist: mandag den 02/12 2013 Kl. 5:00
Denne rapport er afleveret via Campusnet (der skrives ikke under)
Denne rapport indeholder 22 sider ekskl. forside, og bilag.

s123064, Nielsen, Martin

s130045, Kheder, Mustafa

s103185, Sløgedal, Magnus B.

s134000, Budtz, Christian

s133984, Freudendahl, Jens-Ulrik

s134004, Vørmadal, Rúni Egholm

Indholdsfortegnelse

1 Timeregnskab

Matador del 3.							
Time-regnskab	Ver. 2013-09-07						
Dato	Deltager	Design	Impl.	Test	Dok.	Andet	Ialt
13/11/13	Rúni	2	3				5
14/11/13	Rúni		2.5	0.5			3
15/11/13	Christian	3				0.5	3.5
15/11/13	Rúni	2.5				0.5	3
15/11/13	Magnus	2.5				0.5	3
15/11/13	Martin	2.5				0.5	3
15/11/13	Jens	2.5				0.5	3
16/11/13	Christian	2					2
17/11/13	Magnus		1	1	0.5		2.5
18/11/13	Rúni		2				2
18/11/13	Christian	1			2		3
18/11/13	Jens		3				3
19/11/13	Martin		5.5				5.5
19/11/13	Christian	0.5	2				2.5
19/11/13	Magnus	0.5	2				2.5
19/11/13	Jens		2				2
21/11/13	Christian		0.5		3		3.5
21/11/13	Magnus	0.5					0.5
22/11/13	Magnus	1	2				3
22/11/13	Martin	4					4
22/11/13	Rúni	1	1.5	0.5			3
22/11/13	Christian	1	2				3
22/11/13	Martin	2.5					2.5
24/11/13	Jens	2					2
25/11/13	Martin	3					3
25/11/13	Christian				2		2
26/11/13	Martin	5					5
26/11/13	Magnus	2	1	1			4
27/11/13	Magnus		3				3
28/11/13	Magnus		1	1	1		3
28/11/13	Martin	3					3
28/11/13	Christian				2		2
29/11/13	Martin	2			2		4
29/11/13	Christian		1		2		3
29/11/13	Rúni	1	0.5	1	2		4.5
29/11/13	Magnus	1	1		2		4
29/11/13	Jens				1	1	2
30/11/13	Christian				0.5		0.5
30/11/13	Martin				1.5		1.5
30/11/13	Rúni		0.5	1	2		3.5
30/11/13	Magnus				1		1
1/12/2013	Christian				1		1
1/12/2013	Magnus				5		5
	Sum	48	37	6	30.5	3.5	125
	Magnus	7.5	11	3	9.5	0.5	31.5
	Christian	7.5	5.5	0	12.5	0.5	26
	Rúni	6.5	10	3	4	0.5	24
	Martin	22	5.5	0	3.5	0.5	31.5
	Jens	4.5	5	0	1	1.5	12
	Mustafa	0	0	0	0	0	0

2 Indledning

I IOOuterActive har vi fået en ny ordre fra kunden, som indvolverer at udvide det sidste spil vi lavede (CDIO.del2)til et rigtigt brætspil, med nye felt typer, samt at brug af polymorfi og muligheden for at købe felter.

Denne rapport er skrevet til personer med en generel viden omkring UML (Unified Modeling Language) og programmeringssproget Java. Målet med denne rapport er at dokumentere vores arbejde på denne ordre i IOOuterActive.

Til udarbejdelse af rapporten og programmet har vi brugt følgende værktøjer:

- Eclipse - Vores valgte integrated development environment.
- GitHub - Vores version control system of choice.
- Google Docs - for at synkronisere rapportskrivningen.
- LaTeX - For at få et pænere final product.
- Software Ideas Modeller - For vores UML-diagrammer.

Rapporten beskriver følgende emner:

- Krav - Vi har foretaget en kravspecificering og en use-case analyse, hvor vi kommer ind på en fully dressed use-case beskrivelse af "Land on fleet", for at give os en bedre forståelse af kundens ønsker, og omfanget af opgaven.
- Design - Vi har dokumenteret vores design i UML form for at give os et overblik over programmet og fastlægge placeringen af koden. vi bestræber os på at overholde GRASP koncepterne i designet.
- Implementering - Vi har implementeret det valgte design. Dette er blevet dokumenteret i rapporten ved hjælp af beskrivelser af alle klasserne, en forklaring af konceptet arv i en objekt orienteret sammenhæng samt termet abstract.
- Test - Vi har testet programmet, både manuelt og via JUnit, og benyttet FURPS+ for at prøve at sikre en god maintenance og operation af programmet.

3 Krav

Kundens vision er at få leveret et terningspil, der kan spilles af 2-6 spillere. Spillerne rykker rundt på et ringformet spillebræt med 21 felter. Hver spiller starter med 30.000. Spillet slutter når kun en spiller ikke er bankerot.

Domænemodel

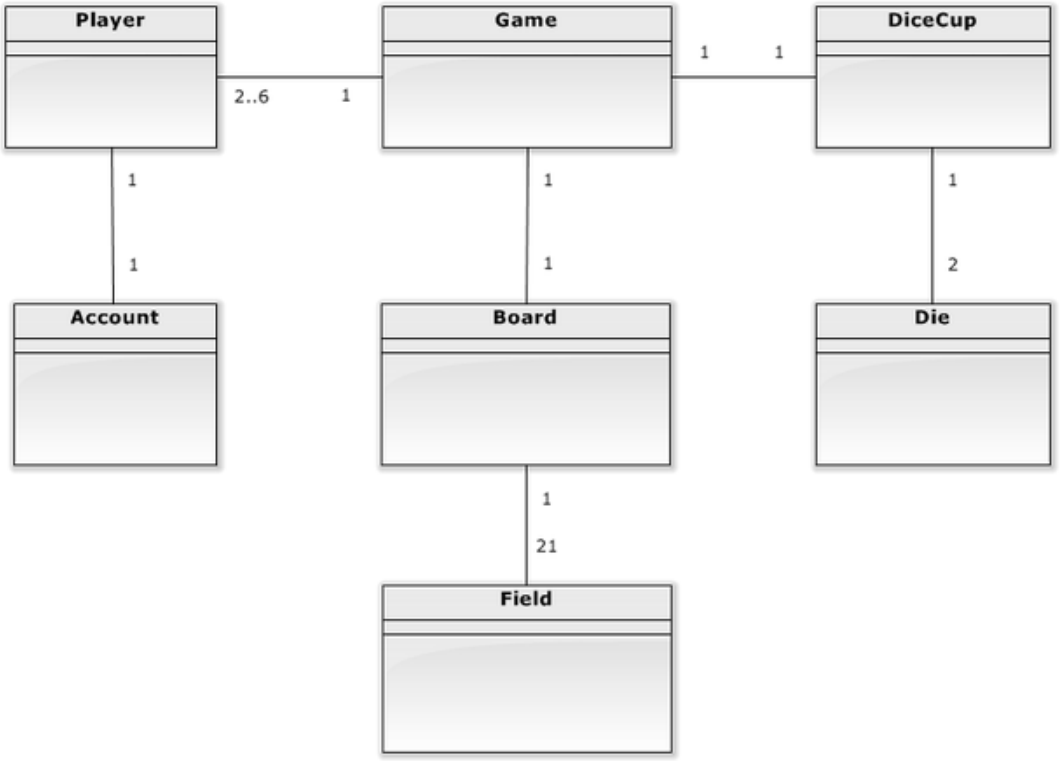

Figure 1: *Domæne model*: Viser hvordan spillet hænger sammen ude i den virkelige verden. Det man ikke kan se her, er at Field, indeholder forskellige slags felter.

Use Cases

Vi har valgt at betragte spillet som bestående af en overordnet spil 'use case', hvor spillerne efter tur slår med terninger og rykker rundt på felterne. Vi betragter hvert ophold på et felt som en separat sub use case, der igen udvides i flere trin (se Use case diagram). Nogle af felterne kan ejes og modelleres derfor i en separat use case - der igen er delt op i flere typer af felter - alt efter hvordan lejen af feltet afgøres.

Af kunden er det specificeret at netop 'Land on Fleet' skal være særligt grundigt dokumenteret - 'fully dressed'. Vi har valgt at beskrive en sub use case, 'Land on Field', der er højere i hierarkiet, som 'fully dressed', idet der er mange generelle elementer, der går igen i de forskellige use cases. 'Land on Fleet' er således beskrevet som en extension af den mere generelle 'Land on Ownable Field', der igen er et specialtilfælde af 'Land on Field'.

Use Case Diagram

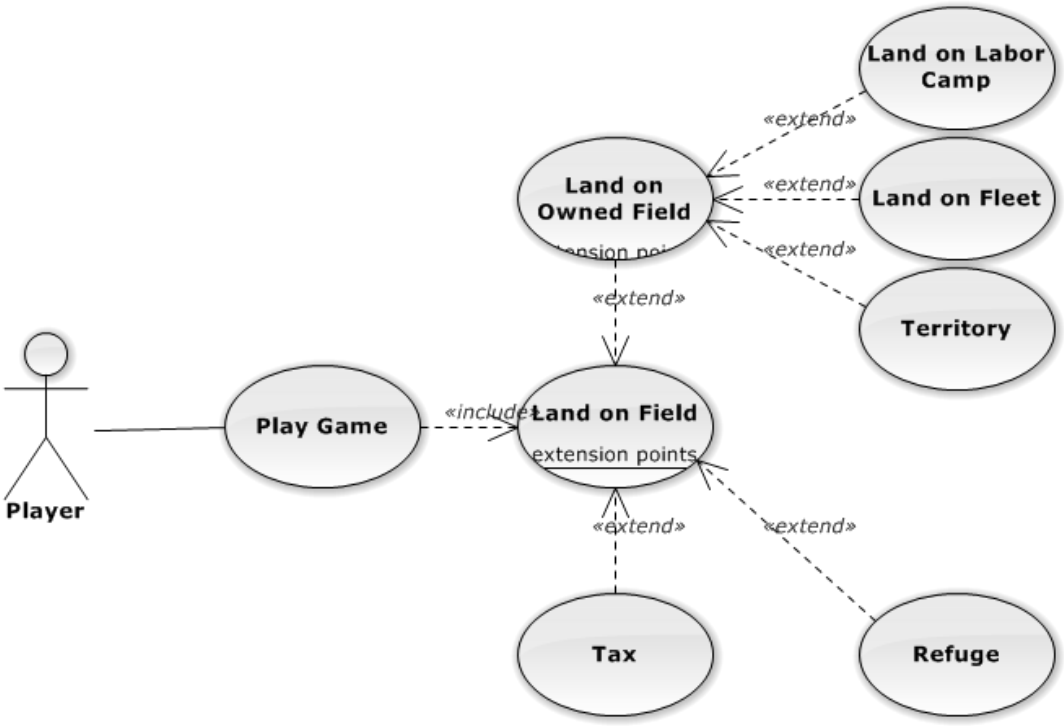

Figure 2: *Use Case Diagram*: Vi har valgt at beskrive 'Land on Field' som en include i Play Game - selvom der ikke er flere use cases der include'r 'Land on Field'. Dette er gjort for at synliggøre relationen mellem sub use cases.

Main Use Case 'Spil Terningspil'

1. Spillerne starter spillet og vælger hvor mange spillere de vil være.
2. De indtaster spillernavne.
3. Spillerne starter med 30000 points hver. Spillebrættet har 21 felter, der ligger i en ring. Nogle af felterne kan ejes af spillerne.
4. Spillerne slår efter tur med 2 terninger og rykker summens øjne frem på spillebrættet. Alt efter hvilket felt man lander på mister eller får man penge og en ejer modtager evt. pengene. Se sub use case 'Land on Field'
5. Spillet slutter når alle undtagen én spiller er fallit og den tilbageværende spiller vinder.

Extension:
1a. Spillerne kan vælge sprog[1].

Sub Use Case 'Land on Field' (fully dressed)
Included in Main Use Case (4.)

[1]Er ikke specificeret i oplægget, men vi har valgt at implementere det, som en ekstra udfordring.

Scope: Terningspil.
Level: Subfunction.
Primary Actor: Aktive spiller (spilleren der har tur).
Stakeholders and Interests: Alle spillere:

- Er interesserede i at spillerne får opdateret deres pengebeholdning i overensstemmelse med reglerne for felttypen.
- Er interesserede i at finde ud af om den aktive spiller går fallit.
- Forventer entydig kommunikation af resultatet af at lande på feltet.

Preconditions: Spillet er startet og der er en aktiv spiller (der har turen), der lander på et felt.
Postconditions: De involverede spillers pengebeholdning bliver opdateret. Spillerens tur afsluttes.
Main Success Scenario:

1. Spilleren lander på feltet.
2. Spillerens penge beholdning opdateres.
3. Spillerens tur afsluttes.

Extensions:
2a. Feltet kan ejes - Sub Use Case Land on Ownable Field.
2b. Feltet er af typen 'Tax' - Sub Use Case Land on Tax.
2c. Feltet er af typen 'Refuge' - Sub Use Case Land on Refuge
2d. Spilleren går fallit

1. Spillerens penge sættes til 0.
2. Spilleren får ikke flere ture.

Special Requirements, Technology and Data variations list.
Der er i den givne opgave ikke stillet krav inden for ovenstående.
Frequency of occurrence.
Hver runde i spillet.
Open Issues.
Beskrevet under afklaring og antagelser.

Sub Use Case Land on Ownable Field

Extends *Land on Field.*
Feltet kan ejes.

1. Feltet har ingen ejer.
   (a) Spilleren tilbydes at købe feltet, har tilstrækkeligt med penge og køber feltet.
      i. Spilleren bliver ejer af feltet.
      ii. Prisen for feltet fratrækkes spillerens saldo.
   (b) Spilleren tilbydes at købe feltet, har tilstrækkeligt med penge, men køber ikke feltet.
   (c) Spilleren har ikke tilstrækkeligt med penge til at købe feltet.

2. Feltet er allerede ejet af spilleren.

3. Feltet er ejet af en anden spiller.

    (a) Spilleren betaler leje til ejeren.

Extensions:
3.1a. Feltet er et 'Territory' - Sub Use Case Land on Territory.
3.1b. Feltet er et 'Labor Camp' - Sub Use Case Land on Labor Camp.
3.1c. Feltet er et 'Fleet' - Sub Use Case Land on Fleet.
3.1d. Spilleren har ikke tilstrækkeligt med penge til at betale ejeren.

1. Ejeren får spillerens resterende penge.

2. Spilleren går fallit.

Sub Use Case Land on Territory
Extends *Land on Ownable Field.*
1. Spilleren der lander på 'Territory' betaler ejeren et foruddefineret beløb.

Sub Use Case Land on Labor Camp
Extends *Land on Ownable Field.*
1. Spilleren der lander på 'Labor Camp' betaler ejeren 100 gange øjnene på det slag, der har bragt ham til feltet.

Sub Use Case Land on Fleet
Extends *Land on Ownable Field.*
1. Spilleren der lander på 'Fleet' betaler ejeren $2^n \cdot 250$, hvor n er antallet af fleets, som ejeren ejer.

Sub Use Case Land on Tax
Extends *Land on Field.*
1. Spilleren mister et foruddefineret beløb.
Extension: 1a. På nogle 'Tax' felter kan spilleren i stedet vælge at betale en procentdel af sin samlede formue

Sub Use Case Land on Refuge
Extends *Land on Field.*
Spilleren modtager et foruddefineret beløb.

Non funktionelle krav

I den givne opgave er der det er et krav at der er udarbejdet Kravspecificering:

1. Et use-case diagram og use-case beskrivelser. Som minimum skal "Land on fleet" være fully dressed.

2. Domæne-model og BCE-diagram

Kodemæssigt forlanges:

1. Lav passende konstruktører.
2. Lav passende get og set metoder.
3. Lav passende toString metoder.
4. Lav en klasse GameBoard der kan indeholde alle felterne i et array.
5. Tilføj en toString metode der udskriver alle felterne i arrayet.
6. Lav en Junit test til hver af felttypernes "landOnField"-metode.
7. Lav det spil kunden har bedt om med de klasser I nu har.
8. Benyt den udleverede GUI.

Designmæssigt forlanges:

1. DSD'er (Design Sekvens Diagrammer). Som minimum "Land on fleet".
2. Forklaring af arv, keywordet abstract, og konceptet i at implementere landOnField metoder i både super og subclasses (override).
3. Dokumentation for test med screenshots.
4. Dokumentation for overholdt GRASP.

Afklaring og antagelser

1. Vi adspurgte kunden om felterne skulle ligge i nummereret rækkefølge på brættet. Kunden svarede at det var valgfrit, men så gerne at de lå tilfældigt.
2. Vi antager at alle spillere starter på første felt
3. Vi antager at alle felter bevarer deres værdi når de er købt - således at det koster 10% af ens saldo + 10% af indkøbsprisen for de Tax felter hvor det er muligt at betale 10% af sin formue.

Ikke honorerede krav

Vi har forsøgt at honorere alle krav.

4 Design

Ud fra ovenstående use case analyser er vi fortsat med at designe en løsning på opgaven. I lighed med den foregående opgave har vi besluttet os for en løsning med en decorator, der sørger for at gøre det let at oversætte spillet til et andet sprog. I vores tilfælde har vi en lidt speciel GUI, der - imodsætning til en 'normal' GUI - afventer en forespørgsel fra vores Gamecontroller/Decorator. Dette afspejles i vores BCE diagram. I vores BCE diagram er lagt op til en central controller, hvilket vi dog har valgt at 'bøje' lidt i klassediagrammet.

BCE Diagram

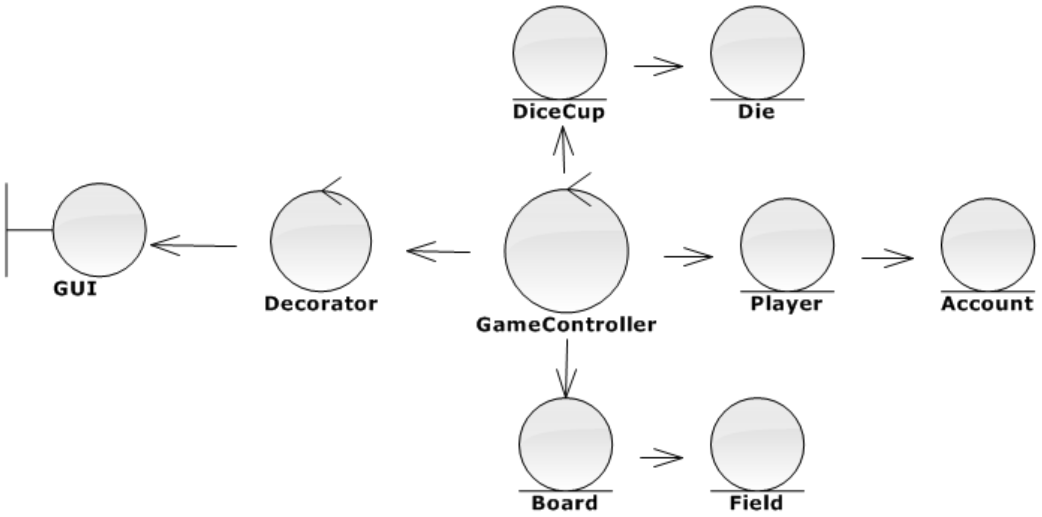

Figure 3: *BCE diagram*: Bemærk den lidt usædvanlige relation mellem GameController Decorator og GUI. GUI afventer en forespørgsel fra Gamecontroller/Decorator, hvorfor pilen vender modsat vanlig interaktion med en GUI.

Design Klasse diagram

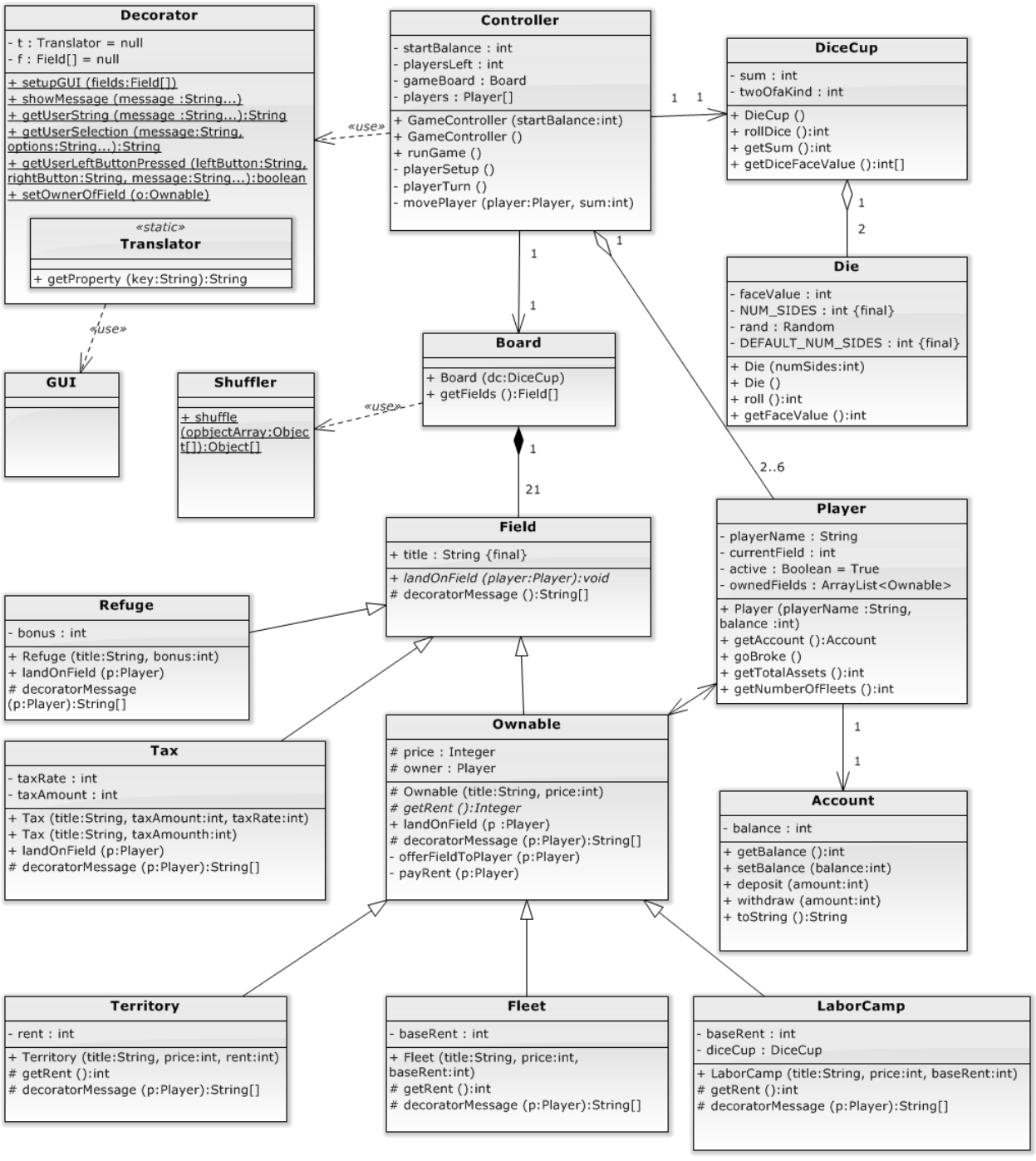

Figure 4: *Design klasse diagram*: Flere 'use' relationer er udeladt for overskuelighedens skyld. Således er der en 'use' relation fra Refuge, Tax, Territory, Fleet, LaborCamp og Player til Decorator, der er en statisk klasse og dermed kan tilgås globalt. Desuden er udeladt en association fra LaborCamp til DiceCup, der bruges til at tilgå terningslaget. I design klassediagrammet har vi udviklet BCE modellen yderligere, så den rent faktisk afspejler koden. Der er dukket to nye hjælperklasser op - Shuffler og Translator, der er henholdsvis en statisk hjælperklasse, der kan blande objekter i et array og en indre klasse, der oversætter Strings via en properties fil, der er sprogspecifik.

Arv, Polymorfi og Abstract

Klassen Field er nu videreudviklet til at have et arvehieraki, der reflekterer at Fields er forskellige, men har ensartede attributter og metoder - altså udviser polymorfi. Polymorfi er: *"Muligheden for at bruge den samme kode på flere forskellige objekter, og for at den kode kan opføre sig forskelligt alt efter hvilket objekt der er tale om."*[**?**] I vores kode bruger vi polymorfi til at differentiere mellem forskellige felters landOnField metode, og vi bruger det i decoratorMessage til at bestemme hvilket text output, der skal sendes til vores Decorator og videre til GUI.

Alle felterne har attributten 'title' tilfælles, hvorfor den ligger i øverste klasse 'Field' i arvehierarkiet. Desuden implementerer Field metoden decoratorMessage(), og en abstrakt metode - landOnField(). Det at en metode er abstrakt, betyder at den ikke bliver implementeret i den klasse hvor den er erklæret abstrakt - det vil sige den ikke har en metode-body. Fordelen ved at erklære en abstrakt metode i superklassen er at man tvinger sub klasser til at implementere klassen, og man derfor kan være sikker på at alle subklasser har metoden. Når en metode bliver erklæret abstrakt skal klassen også erklæres abstract. For en klasse betyder dette at den ikke længere kan instantieres, så hvis man skal bruge den bliver man nødt til at nedarve den til en sub-klasse.

Ownable er super klasse for de felter der kan købes - og en sub klasse af Field. Det er meningsfyldt, da de alle skal kunne købes - altså har en pris, 'price' og en ejer 'owner' og skal implementere metoder til at beregne leje - getRent().

Med de to super klasser Field og Ownable har vi opnået at kunne genbruge så meget kode som muligt og undgår dermed at skulle rette flere steder i koden når metoderne skal opdateres.

Sekvensdiagrammer

Ud fra uses cases og klasse diagram har vi forsøgt at analysere flowet i spillet og modelleret det i de følgende sekvensdiagrammer.

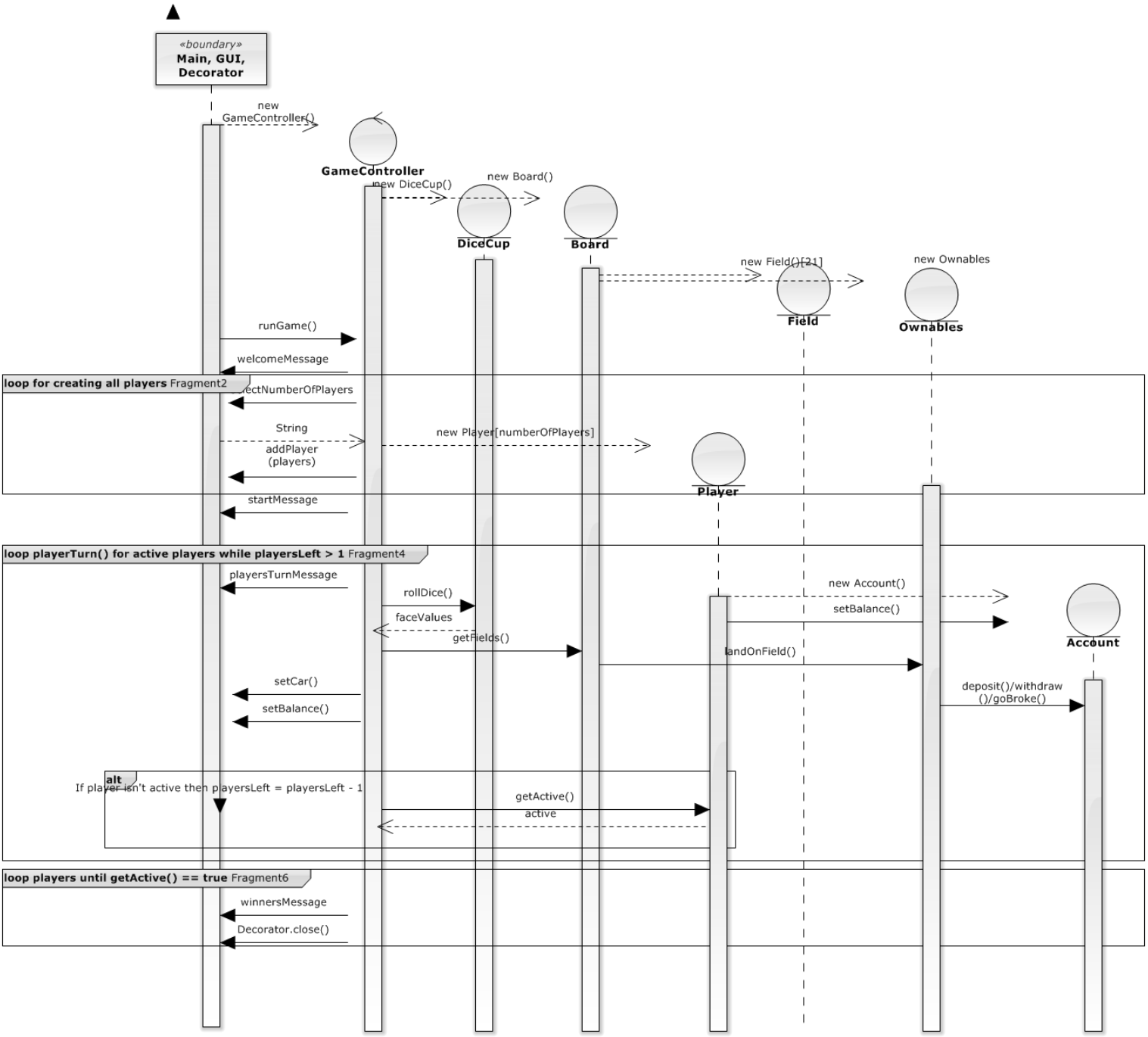

Figure 5: *Sekvensdiagram 1*: Play Game.

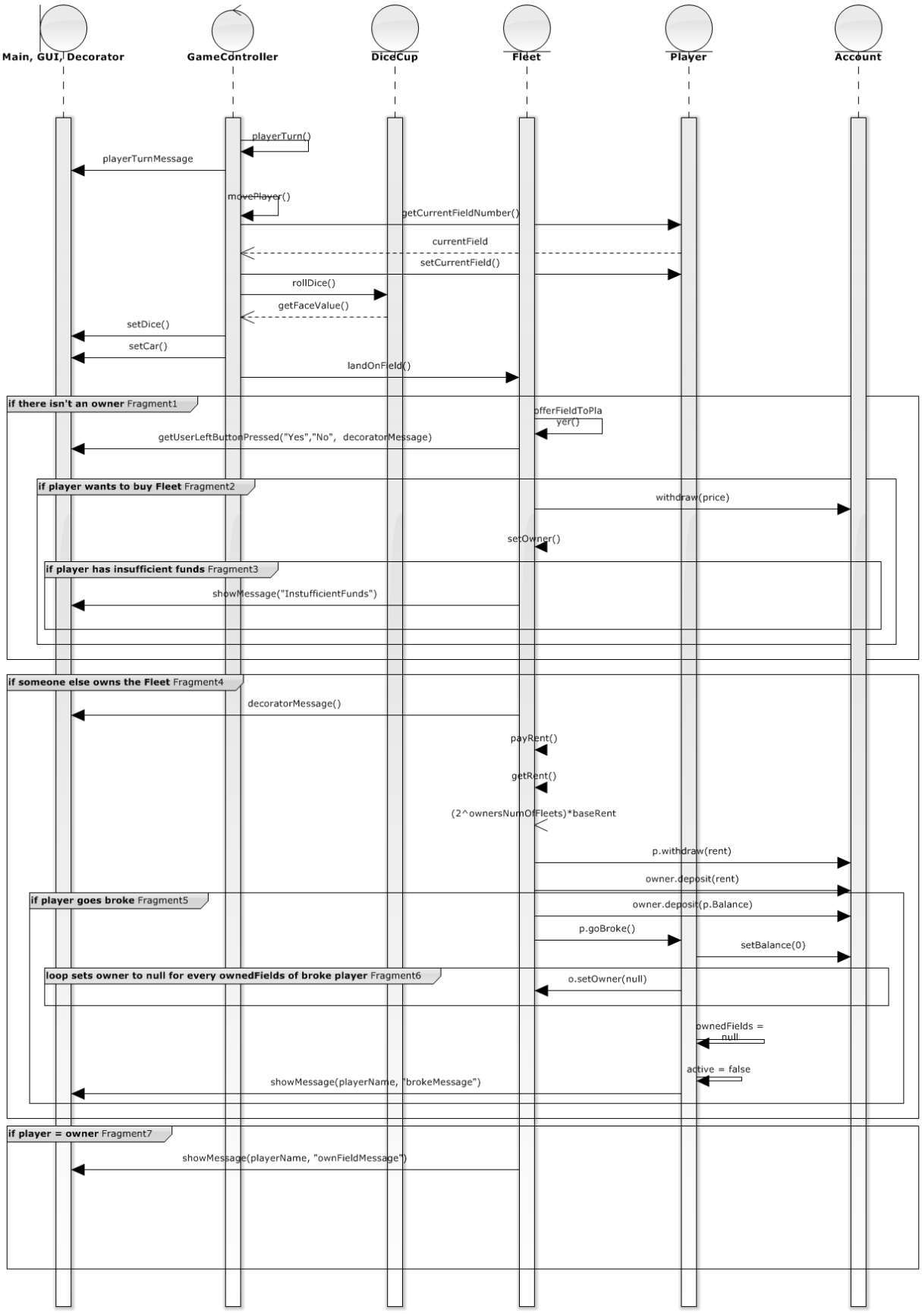

Figure 6: *Sekvensdiagram 2*: Land on Field.

Sekvensdiagrammerne er blevet lidt forsimplet for at fremme forståelse, og for at ikke få for mange metodekald, der ikke ville bidrage til forståelsen af diagrammet. Klasser der fungerer sammen og/eller ensartet er slået sammen. Klasser af begrænset vigtighed for diagrammerne er også udeladt. Shuffler indeholder en statisk metode, der blander felterne i starten af spillet. DiceCup og Die arbejder så tæt sammen, at Die er undladt. Die er, som navnet hentyder, en terning. Ownable er en superklasse til alle typer af felter, der kan ejes. Fields er en superklasse til alle typer af felter. De forskellige typer af felter, er slået sammen i det første diagram, og er kaldt Ownables, og i det andet diagram, hvor vi viser landOnFleet, har vi valgt at udelukke de andre felter, og ladt Fleet være den Ownable vi modellerer, da den er en subklasse af "Ownable". Mange returns er også blevet undladt for overskueligheden. Der er brugt tre forskellige ikoner for klasser, nemlig Boundary, Control og Entity (BCE), og man kan skelne dem ad ved at Boundary ser ud som, at den rører ved en væg, Control har en lille pil på sig, og Entity ser ud som den rører ved noget gulv. Derudover er der brugt to typer kasser. Loops, som repræsenterer loops, hvis køringspremisser eller formål er beskrevet i den lille tekstboks øverst i firkanten. Det andet er alt, som er kort for alternate. Dette viser et andet udfald, der kan ske. Det bliver brugt til at vise 'if's og øverst i boksen er præmissen for det givne if-alternativ anført.

Det første sekvensdiagram viser et overordnet forløb af programmet, mens det andet diagram viser alle de forskellige udfald, der kan opstå, når en spiller lander på et felt af typen Fleet. I det første er det på et lidt højere niveau end i det andet, da vi i det andet graver helt ind og ser på alle de forskellige ting, der kan ske, og ender derved med seks alternate-bokse.

Hvis en spiller lander på en Fleet, er der allerede tre muligheder. Hvis ingen ejer Fleet, skal det tilbydes til spilleren, hvorefter han kan vælge om han vil købe den eller ej, men der er en ekstra mulighed i det, at han måske ikke har penge nok.

Mulighed nummer to er, at en anden spiller ejer den. Så skal rent regnes sammen, flyttes, og vi må se om spilleren er gået fallit. Hvis han er det, skal spilleren smides ud af spillet, og hans felter skal frigøres.

Mulighed nummer tre er, at han selv ejer feltet, og så skal han bare have en besked om, at han er landet på sit eget felt.

GRASP patterns

Vi har forsøgt at arbejde med 'responsibility driven design' og implementeret Controller, Creator, Information Expert, Low Coupling, High Cohesion og Polymorphism.

Controller paradigmet er søgt overholdt ved vores GameController klasse, der håndterer al spil logik og delegerer ansvaret videre. I vores projekt har vi en lidt speciel statisk GUI, der tillader at vi tilgår den fra alle klasser i programmet. Da den algoritme som felterne bruger er afhængig af spillerens input fra GUI'en, har vi skullet vælge mellem at implementere 1) en løsning hvor GameControlleren først tilgår feltet for at høre om det kan ejes, dernæst om den kan købes eller er ejet, og dernæst eventuelt sætter ejeren for feltet, eller 2) som vi i stedet har valgt - en løsning hvor felterne optræder som sub-controllere og selv tilgår GUI'en for at håndtere hvad netop det felt gør. På den måde bryder vi bevidst med BCE paradigmet (I det felterne også er entities), men opnår en mere overskuelig løsning kodemæssigt og da relationen er en 'use' relation med GUI'en øger det ikke coupling væsentligt. Cohesion bevares også, da det er felterne selv, der logisk set kender reglerne for hvad der sker når en spiller lander på feltet. Det er muligt

at genskabe BCE, ved at hvert felt får sin egen controller (hvilket ville være Pure Fabrication), men i det givne tilfælde er det meget begrænset hvad en entity klasse skulle indeholde, hvorfor vi har valgt at undlade at splitte klasserne op.

Creator er søgt overholdt ved at de klasser der 'contains' eller 'aggregates' andre klasser, også har ansvar for at instantiere dem. GameControlleren instantierer DiceCup, der igen instantierer Die. Ligeledes oprettes Player med account og Board med Fields. Vores Boundary klasse er et specialtilfælde, da den er statisk, men initialiseres af GameControlleren, der har hovedansvaret for at interagere med den.

Information Expert er søgt overholdt, ved kun at bevare information og referencer i én klasse - den der er den mest oplagt til at indeholde informationen. Vi har brudt med paradigmet enkelte steder for at opnå en mere overskuelig løsning. Således har Gamecontrolleren en integer (playersLeft), der holder styr på hvor mange players der er tilbage. Denne information er dubleret fra players, der selv holder styr på om de er med i spillet med deres boolean active. På samme måde ved både felterne hvem der ejer dem og playerne ved hvilke felter de ejer. Dette giver desuden en højere coupling - da der er en reference begge veje - Det giver også en lavere cohesion, da det nu er spredt over to klasser. Man kan argumentere for at det stadig er logisk, da en grundejer kender sine grunde og skødet på grunden ligeledes er påtrykt en ejer. Vi gør det under alle omstændigheder for at opnå en simplere implementering af Fleet, der er nødt til at vide hvor mange fleets ejeren har. Alternativet er at Fleet skal modtage en reference til board og iterere over alle felterne for at afgøre hvor mange fleets ejeren har. Et andet alternativ er at oprette en skødedatabase som kan adspørges når behovet opstår - det ville svare til GRASP konceptet indirection (og pure fabrication).

Low Coupling er opnået ved få koblinger pr. klasse - Dice er koblet til DiceCup, Account til Player og Fields til Board, der igen er koblet til GameController (se klasse diagram). GameControlleren har naturligt lidt højere coupling - da Dicecup, Players og Board (og Decorator) alle har betydning for game-flowet. Det er et acceptabelt antal bindinger - der understøtter high cohesion. Vi implementerer en undtagelse med et specifikt felt - 'Labor Camp', der har en reference til DiceCup. Det er gjort for at kunne beregne lejen - uden at skulle passe summen af øjne til feltet. En alternativ løsning kunne være at implementere DiceCup som en Singleton, der kunne tilgås globalt. Som allerede nævnt har vi en 'use' relation fra alle felttyperne til GUI - idet de tilgår GUI for at spørge spilleren om han vil købe grunden eller betale et fast beløb eller procent af formue i skat. Da det er en 'use' relation er coupling stadig lav.

High cohesion er forsøgt bevaret ved at klassernes ansvarsområder er nært beslægtede. Således har Die kun ansvar for terningernes øjne og at 'slå' et nyt tilfældigt slag. Dicecup håndterer summen af øjnene og relaterede opgaver og så fremdeles. Dette er i høj grad understøttet af low coupling, hvilket også afspejles i at vi bryder med high cohesion samtidig med at vi bryder med low coupling i tilfældet med ejerskabet af fields (som beskrevet ovenfor under information expert)

Polymorphism er implementeret i vores felter. Her er ensartede klassers kode forsøgt samlet i superklasser, således at vi genbruger kode i højest muligt omfang og undgår at skrive den samme kode to gange. Det afspejles tydeligst i decoratorMessage() metoden, der nedarver i 2 niveau og sammensætter en ensartet besked til spilleren, sammensat af en besked om hvilket felt man er landet på (bestemt af Field), en besked der afhænger af om feltet kan ejes (bestemt af Ownable) og en besked fra selve feltet. Dette understøtter reusability, idet ændringer i koden kun skal indføres et sted.

5 Implementering

Komplekse udsnit af kildekode. Implementering af arv. Kode evt. i bilag Vores implementering afspejler vores klassediagram.

Klasser

GameController

Er en facadecontroller - delegerer så meget ansvar som muligt videre (jvf. GRASP). Holder således kun de nødvendige instances af koblede klasser. Som nævnt gør vi en undtagelse og lader Gamecontrolleren holde en int, der angiver hvor mange spillere, der er tilbage i spillet. Dette er gjort for at undgå 'tung' kode, hvor man er nødt til at iterere over spillerne for at finde ud af hvor mange der er tilbage.

Decorator

Decorator er designet efter decorator-patternet, dens primære funktion er at dekorere output til GUI'en. Ligesom GUI'en er Decorator en static klasse, så vi kan tilgå den alle steder fra, dette er et bevidst designvalg da vi har brudt med Border - Controller - Entity på dette punkt og udskriver text fra vores Field klasser.

Decorator indeholder jævnføre design-patternet kopier af GUI's metoder, disse er designet via. varargs, så tekststrenge kan deles op og oversættes individuelt før de bliver sat sammen og passet til de relevante GUI metoder. Til at oversætte tekst bruger vi en java.util.Properties klasse, med tilhørende .properties filer for hvert sprog i spillet, der er blevet extended for at give os en specifik handling af den særlige case hvor properties returnere null (den ønskede key findes ikke).

Board

Board er en simpel klasse der indeholder vores fields i et array, dens constructor tager en DiceCup reference, som den passer videre til LaborCamp, og opretter derefter alle vores fields. Til sidst bruger den hjælper klassen Shuffler, til at blande felterne, så boardet bliver nyt for hvert spil.

Field

Field er roden (super class) for vores felt-klasser. Den er en abstrakt klasse og har nogle abstrakte metoder.

Abstrakt betyder for klasser at de ikke kan instantieres, og det betyder for metoder at de ikke bliver implementeret i denne klasse (ingen metode body). Til gengæld bliver alle subklasser tvunget til at implementere samtlige abstrakte metoder.

Field har også en metoden decoratorMessage, denne metode er ligeledes implementeret i samtlige subklasser, men er ikke abstrakt da vi skal bruge den implementation i Field.

decoratorMessage er designet til at bruge nedarvningen, (gennem kald til super.decoratorMessage) til at bestemme hvilken tekst der skal sendes til decoratoren. Den er lidt specielt opbygget da den, i stedet for at konkatenere strengene og returnere dem som en string, opbygger et String array som returneres til Decorator som et vararg.

Ownable

Ownable er en abstract subklasses af field, dens hovedformål er at implementere landOnfield for alle klasser der kan ejes. Dette er gjort fordi metoden landOnfield er den samme for alle felter der kan købes, den eneste forskel er lejen. Lejen beregnes derfor af en abstrakt getRent metode som sub-klasserne implementerer.

Fleet

Fleet et felt og er en subklasse af Ownable, der er karakteriseret ved, at man opkræver mere i leje pr gang nogen lander på det, jo flere Fleets, man ejer. Nøjagtigt ligesom vi kender færgerne fra matador. Formlen går $2^{(fleets)} \cdot 250$. Der ved opkræver man 500 pr gang, hvis man ejer 1 færge; 1000, hvis man ejer 2; 2000, hvis man ejer 3; og 4000, hvis man ejer alle fire færger.

Vores feltklasser holder som sagt en smule kontrol, da metoden landOnField() har en use-relation til GUI'en.

Constructoren til Fleet holder en String title, int price og int baseRent.

LaborCamp

LaborCamp er et felt og er en subklasse af Ownable, der er karakteriseret ved, at man betaler 100 gange øjnene af den uheldiges kast. Desuden vil man betale dobbelt, hvis en spiller ejer begge LaborCamps. Borset fra prisen, er det ækvivalenten til sodavandsfabrikkerne i Matador. Formlen for den opkrævede leje er ((antal LaborCamps ejet af opkræveren) * (kastet) * 100).

Constructoren indeholder en String title, int price og int baseRent.

Territory

Territory er et felt og er en subklasse af Ownable. Det er det mest normale af de forskellige slags felter, i det der er 11 af dem, og det er meget ligefrem i forhold til nogle af de andre felter, i det, at man betaler en fast pris, hver gang man lander på dem. De varierer i pris og leje, men er konstante. Selve constructeren holder en String title, en int price, og en int rent.

Tax

Tax er en subklasse af Field, og er et felt, der ikke er en Ownable. Det er et felt, hvor man betaler skat, hvis man lander der. Skatfelterne fungerer ligesom i Matador. Der er to af dem, og på det ene bliver der betalt et fast beløb af 2000, og på det andet kan man vælge mellem at betale 4000, eller 10% af sin samlede formue. Constructoren holder en String title, int taxAmount og int taxRate. Her handler det om at have lidt hovedregning, da det forkerte valg kan koste en dyrt.

Refuge

Refuge er et felt og en subklasse af klassen Field. Hvor man modtager 500 eller 5000 for at lande på feltet. Constructoren indeholder en private attribute (int bonus) og en String title. Derudover har den nedarvet metoden LandOnField(), som fortæller GUI at spilleren har landet på Refuge og fortæller hvor meget attributen bonus skal deposit til

playerAccount. Derudover har den også en String[] decoratorMessage og en toString() metode, som returnerer ($title + \backslash bonus : " + bonus$).

Die

Klassen Die har to attributter (faceValue: Int og NUM_SIDES: Int). Den har et objekt der refererer til klassen Random, som bruges til at generere en pseudotilfældig integer med variablen NUM_SIDES og en integer med variablen faceValue, der holder terningens værdi.

Klassen Die er udvalgt fra det forrige terningespil med den mest interessante metode roll(), der henter en pseudo tilfældig integer mellem 0 og 5 fra objektet Random, hvor faceValue sætter denne random til ++1 og derefter returnerer faceValue.

DiceCup

Klassen indeholder metoden rollDice og getterne getSum, getDiceFaceValues og getTwoOfAKind, hvor der er også er en toString, som dog ikke bliver anvendt i spillet. DiceCup constructoren instantierer de to terninger som skal bruges i spillet.

Når GameControlleren beder om at få kastet terningerne kører DiceCup metoden rollDice, så summen af de to terninger bliver beregnet. Når summen af de netop kastede terninger skal bruges, bruger GameController getSum, som returnere summen. Fordi at GUI'en også skal vise øjnene pa de to terninger, indeholder DiceCup ogsa getDiceFaceValues som returnere et array af de to terningers øjne, som GameControlleren så kan tilkalde. Derudover har vi også en getter til TwoOfAKind (getTwoOfAKind). Denne indeholder ikke selve reglerne til spillet, men giver GameControlleren muligheden for at kunne tilkalde og få at vide om terningerne har slået to ens, og hvad der er slået to ens af. getTwoOfAKind returnerer derfor enten 0, hvis terningerne ikke er lige ellers 1 for to 1'ere og 2 for to 2'ere osv.

String toString kan blive brugt hvis man vil vide terningernes værdier. Den kan være nyttig hvis man ikke kører spillet igennem en GUI, så bliver en String returneret med forklaring på hvilke værdier terningerne har.

Player

Player har til formål at opbevare data for den enkelte spiller. Den indeholder variable for navn og placering på brættet, en account til at styre spillerens balance, en liste med alle felter som spilleren ejer og en bools variabel der styrer om man stadig er med i spillet. Herudover har player alle nødvendige get/set metoder og en toString.

Player har også en goBroke metode, der er bruges til at rydde op efter en spiller der er gået falit. goBroke sætter spillerens formue til 0, fjerner ham som owner af alle fields, og sætter den boolske "active" variabel til false.

Account

Klassen Account har atributten int, der modsvarer pengebeholdningen. Den har derudover en constructor, der opretter en account med et beløb og en default constructor, som opretter en account med standardbeløbet 30.000.

Account var i den tidligere rapport specificeret til at den ikke måtte få en negativ score. Dette er fortsat, men nu er spilleren bare identificeret som bankerot. Derfor er der implementeret tjek for negative beløb i metoderne. Disse tjek er behandlet med

exceptions. Derudover har klassen Account en standard getScore og en setScore, der indeholder et tjek for om metoden forsøger at kalde med en negativ int.

Metoden deposit tjekker for om den modtager en negativ int og for om score vil gå over maksimumværdien for int. Metoden withdraw kontrollerer for en negativ int og for om score vil gå under 0. toString er autogenereret af eclipse.

6 Test og Kvalitetssikring

Brugertest

Vi har kørt nogle bruger tests, for at se om spillet opfører sig som forventet. Vi har forsøgt at finde de fejl som gør at spillet af en eller anden grund giver et forkert output, så som at spiller ikke går fallit når han skal, men får lov at spille videre. Under **Bilag** kan man se screenshots fra en brugertest.

Under gennemførelsen af en brugertest, så det ud som at en vinder ikke blev præsenteret for spillerne. For at checke om controlleren kørte loopet som finder vinderen, addede vi nogle linjer som blev printet hvis dele af loopen blev kørt. Da vi så kørte den sidste brugertest fandt vi ud af at vinderen altid spiller 1 selv om denne spiller var gået fallit **Bilag9**. Fejlen var at da vi havde tilføjet den linje som blev printet, så havde vi glemt at omslutte if statement med "{" således at alle spillere for den sags skyld kunne være vindere. Ellers har vi kørt en fuldkommen brugertest hvor vi kommer igennem alle scenarier vi kunne komme i tanker om. Bilag 1-9 er de scenarier vi forventer når vi lander på et felt. Hvis man lander på et felt som ikke har en ejer, kan man vælge at købe det, eller lade være **Bilag1**.

Hvis en spiller lander på et felt som er ejet af en anden, så får man ikke mulighed for at købe det, men skal betale til ejeren **Bilag2**. Man kan også se at navnet på feltet og navnet som bliver præsenteret som ejer af feltet er det samme. Det er den gule bil (Christian) som er endt på Magnus felt og skal betale ham leje.

Når man lander på det ene af tax felterne skal man have 2 muligheder for at betale (enten 10% eller 4000kr) **Bilag7**. Mens det andet tax felt kun skal kræve et bestemt beløb **Bilag4**. Disse felter får man ikke mulighed for at købe, ligesom refugee heller ikke kan købes, men her får man penge **Bilag3**.

Et andet scenarie er hvis du lander på dit eget felt, så skal der ikke ske spilleren noget. Spilleren skal kun få at vide at han er landet på sit eget felt **Bilag5**, her kan nævnes at vi har gjort således at felterne på GUI'en opdateres så man kan se hvem er ejer af feltet. Det er også aktuelt når en spiller går fallit, så bliver den spiller fjernet som ejer af de felter han havde og andre spillere får mulighed for at købe det **Bilag6**.

Vi skulle også håndtere at en spiller ikke kan købe et felt for så at gå fallit fordi han ikke har penge nok, og dette håndterer spillet **Bilag8**. Spilleren får at vide at han ikke har penge nok, og spillet kører videre. dog kan spilleren godt gå fallit når han lander på tax feltet hvor man vælge mellem 10% og 4000 kr. Hvis han har råd til det ene, men ikke det andet, kan han alligevel gå fallit ved forkert valg.

To fejl som vi fandt ved brugertest, men som ikke er rettet, er hvis man skriver et grotesk langt navn for en spiller. Navnet bliver så langt at når det er spillerens tur, dækker navnet "OK" knappen, og spillet kan ikke fortsættes **Bilag10**. Det skal altså være et virkelig, virkelig langt navn. Hvis spillerne beslutter sig for at have samme navn, så bliver bliver to spillere oprettet med hver sin konto, men på GUI'en har de samme farve bil **Bilag11**, altså er der kun en bil på brættet, men som alligevel har to destinationer. Man kan derfor kun se bilen og balancen for den ene spiller ad gangen.

Black Box (Junit)

Vi har lavet en JUnit test class FieldJUnitTest, der tester LandOnField for alle vores forskellige Field-klasser. Den opretter først sin egen test GUI, den skal bruges da der er

brugerinput i LandOnField, og derefter kører den en @test for hver Field-type. Indlejret i disse test er også nogle test på ting fra Ownable, såsom at gå fallit og lande på ens eget felt.

Vi har kørt FieldJUnitTest løbene og brugt den til at rette adskillige logiske fejl, til et punkt hvor FieldJUnitTest nu kører fejlfrit.

FURPS+

FURPS+ er en forkortelse for Functionality, Usability, Reliability, Performance, Supportability. FURPS+ er en god checkliste for at opnå et så godt program som muligt, og ikke glemme nogle vigtige dele.

- **Functionality:** Vi har lavet et funktionelt spil, som opfylder alle kode mæssige krav, som for eksempel at bruge arv når vi programmerede felt klasserne og metoden landOnField i klassen Field.
- **Usability:** Spillet er brugervenligt, og der behøves ikke at læses nogen brugervejledning for at kunne gennemføre spillet. Dog skal man kende noget til matador regler for at forstå spillet, men du gennem spillet skal du kun trykke "OK" eller får du to valgmuligheder, hvor der er beskrevet hvad de betyder.
- **Reliability:** Gennem bruger tests og en stor J-Unit test, har vi elimineret alle de fejl, som opstår gennem de forventede scenarier. Som et af kravene har vi med J-Unit testet feltet af typen fleet for:
  - at spiller ikke får fratrukket beløb når han lander på felt som han selv ejer.
  - Når en spiller skal betale til en anden spiller, at den rigtige spiller får bestemt beløb og den anden fratrukket samme beløb.
  - Når en spiller køber en fleet, bliver prisen af fleet fratrukket spillerens konto, og spiller bliver sat som ejer af feltet.
  - Hvis en spiller er ejer af to fleets, så får den spiller som lander på en af hans fleet, fratrukket et beløb som svarer til $250 \cdot 2^{(}antalaffleets)$. Og ejeren tilføjet samme beløb.
  - I tilfælde af at en spiller ikke har nok kapital ved landing på en andens spillers fleet, så bliver spillerens $balance = 0$, og $active = false$ (spilleren bliver inaktiv og er ikke mere del af spillet). Derefter bliver de penge som spilleren havde, overført til ejeren af fleet.

  Så vi har et pålideligt spil, som ikke lukker ned uforventet.
- **Performance:** Dette har ikke været relevant faktor i vores spil. Spillet bruger så få ressourcer at vi har ikke haft brug for kørtids-analyser og optimering, spillet optager heller ikke mærkbar disk-space.
- **Supportability:** Vi har gjort en hel del ud af at fremtidssikre koden, så vi kan genbruge så meget som muligt af koden senere. For eksempel har vi undgået "hard coding", og al tekst i spillet bliver gennem identifiers oversat, ved hjælp af decorator og properties filer, til meningsfyldt tekst. Properties filerne indeholder al tekst, og det gør det nemt at have overblik og ændre på teksten. Vi stødte dog på et problem, som vi løste med gøre felterne til subcontrollere, det gør koden lidt

sværere at vedligeholde, da små ændringer i Decorator kan gøre at vi må gennemgå gamecontroller samt alle subcontrollerne for fejl.

- **+ (Plusset):** Spillet kræver at computeren har og kan køre en opdateret version af java og at mus og tastatur er tilkoblet. Spillet har ingen lyd, så det er ikke en nødvendighed. Ellers er der ikke nogen nævneværdige krav som spillet har til computeren.

7 Konklusion

Vi har alt i alt fremstillet et spil, der funktionelt set lever op til specifikationerne - og tilføjer ekstra funktionalitet. Det kan spilles med begrænsede forkundskaber, om end et kendskab til reglerne er nødvendigt for at få det optimale udbytte. Det er pålideligt, men mangler et tjek for brugernavnenes længde - så det er muligt at ødelægge brugeroplevelsen for sig selv ved at indtaste grotesk lange brugernavne.

Vi har taget nogle designbeslutninger, der skaber højere kobling og bryder med BCE paradigmet. Det giver en overskuelig kode, men skaber det problem at det er sværere at vedligeholde, idet ændringer i Decorator kan skabe behov for ændringer i flere forskellige klasser.

Idet formålet var at undgå for meget kode i vores GameController, kan man forsøge at opnå det samme ved at indføre en dedikeret FieldController, der delegeres ansvaret for sub use casen 'Land on Field'. Vores dobbelte kobling mellem Player og felterne, kan evt. løses ved at indføre et 'tinglysningsregister' - en (singleton) klasse, der kun har til opgave at holde styr på ejerskabet af fields.

Referencer

[1] bla hansen. BWorld. `http://aiweb.techfak.uni-bielefeld.de/content/bworld-robot-control-software/`, 2008. [Online; accessed 19-July-2008].

[2] Happy Kittens. *Definitivily not the title.* burnedbooks.com, -8.

Bilag 1

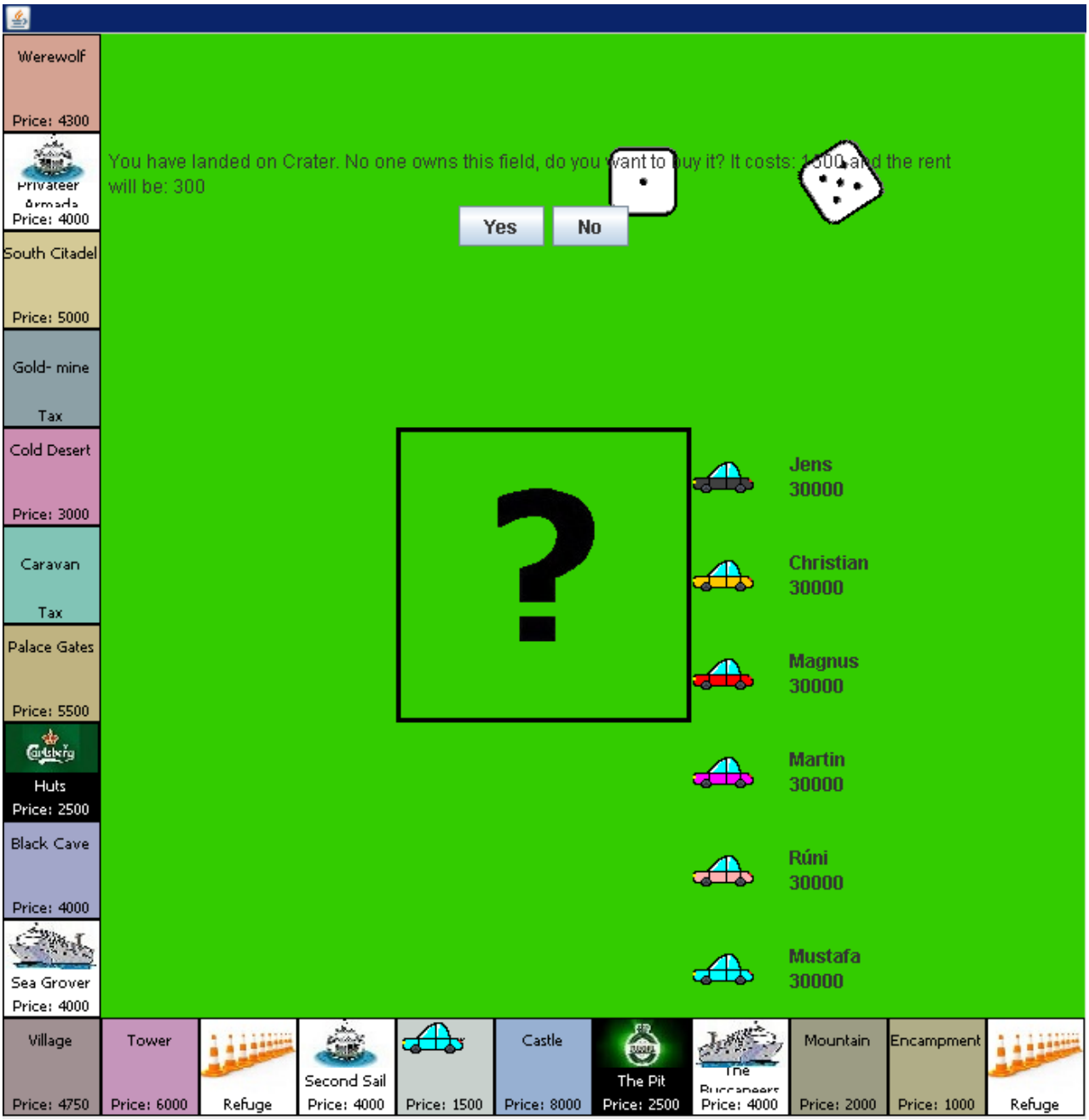

Figure 7: *Bilag 1*: Spilleren tilbydes at købe et felt.

Bilag 2

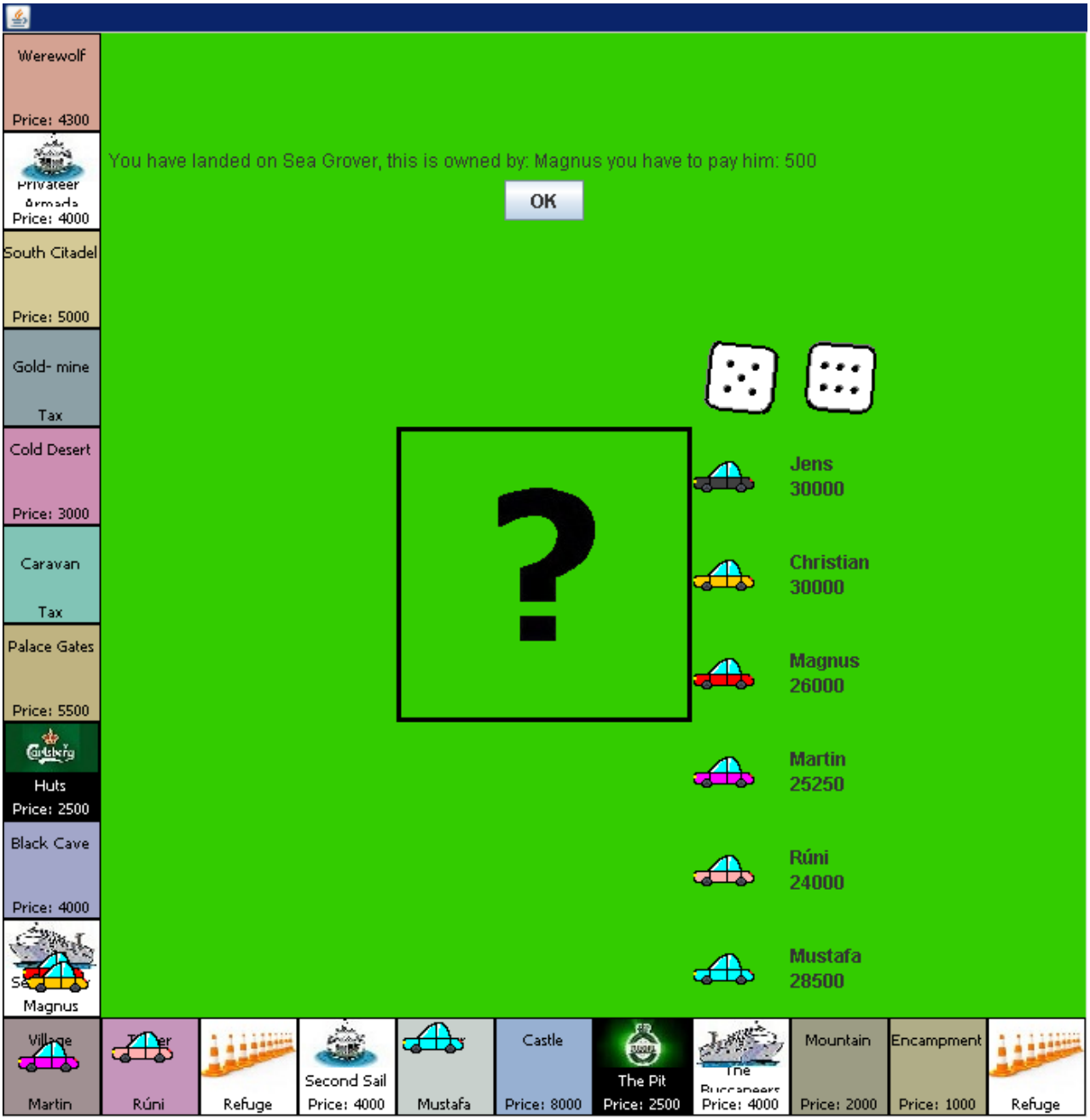

Figure 8: *Bilag 2*: Spilleren betaler leje.

Bilag 3

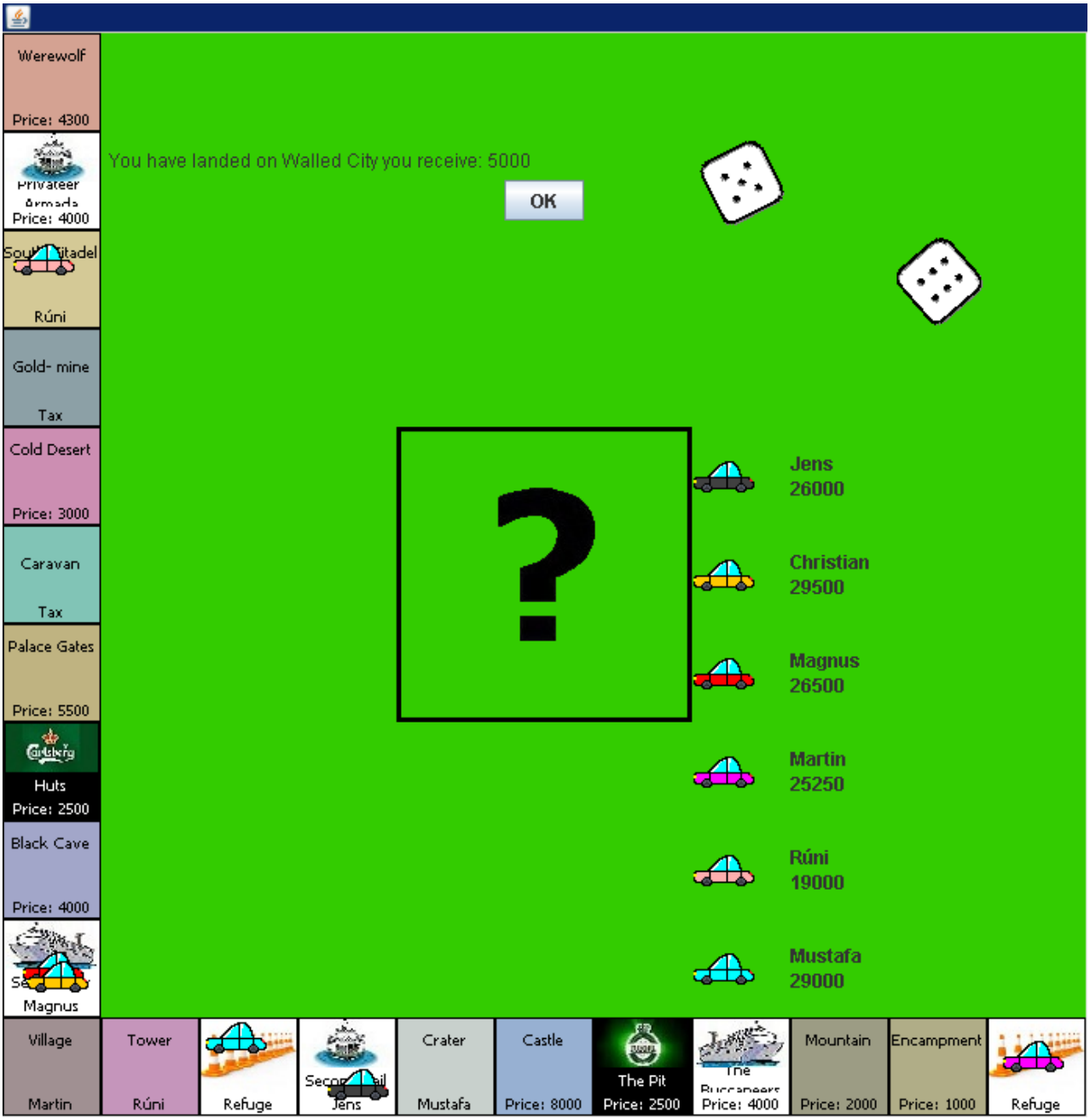

Figure 9: *Bilag 3*: Spilleren modtager penge på 'Refuge'.

Bilag 4

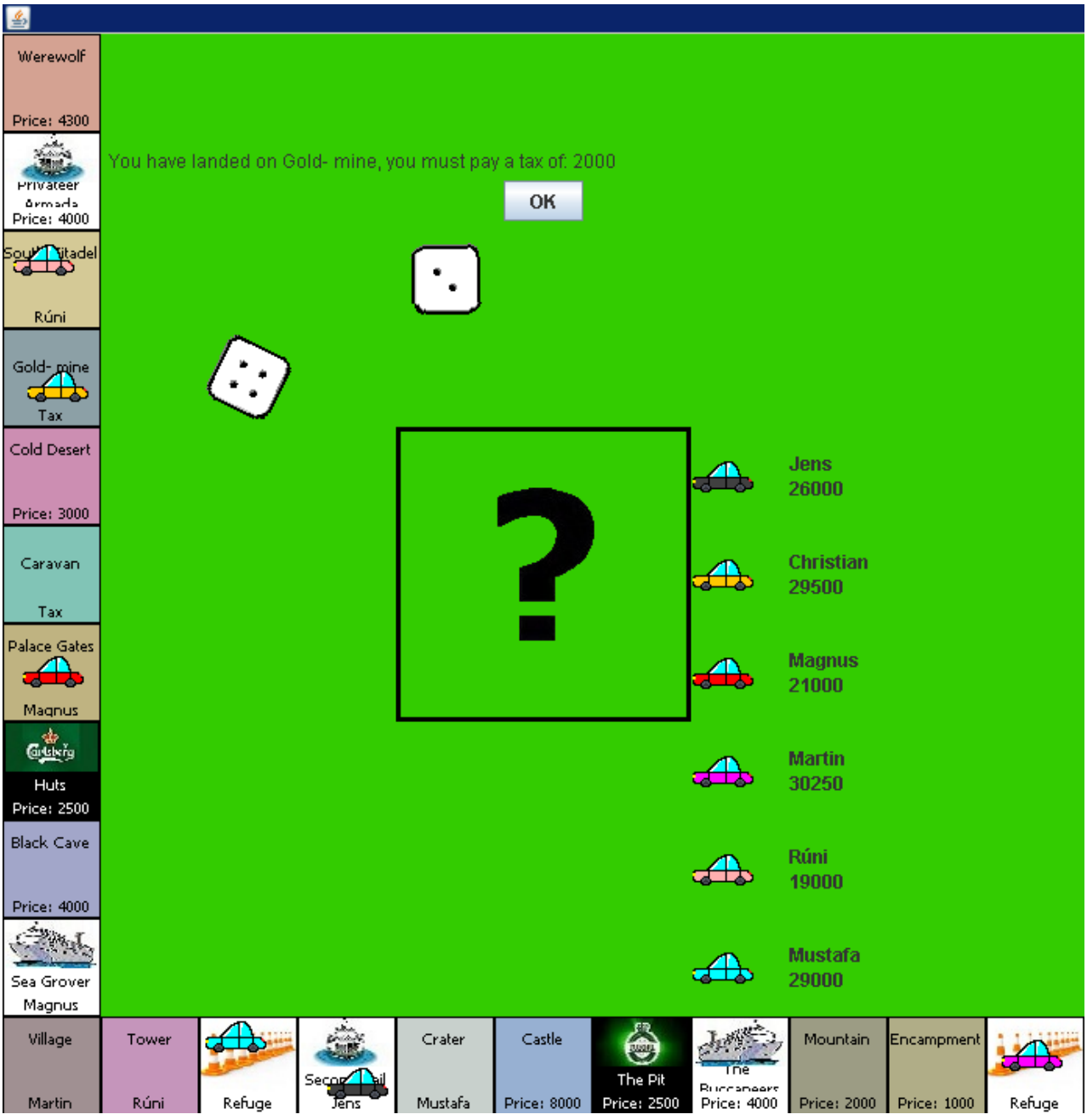

Figure 10: *Bilag 4*: Spilleren betaler fast skat på 'Tax'.

Bilag 5

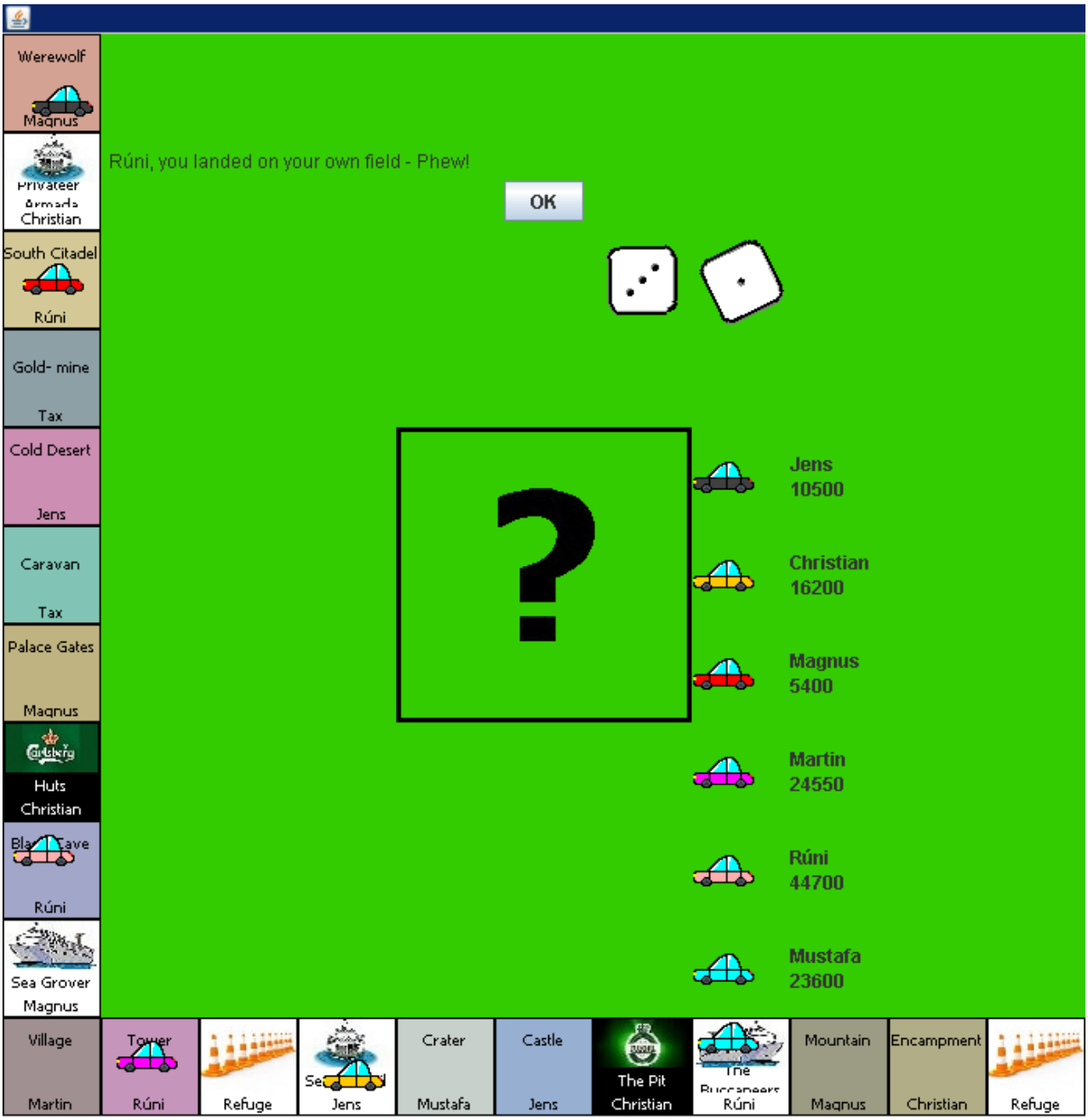

Figure 11: *Bilag 5*: Spilleren lander på eget felt.

Bilag 6

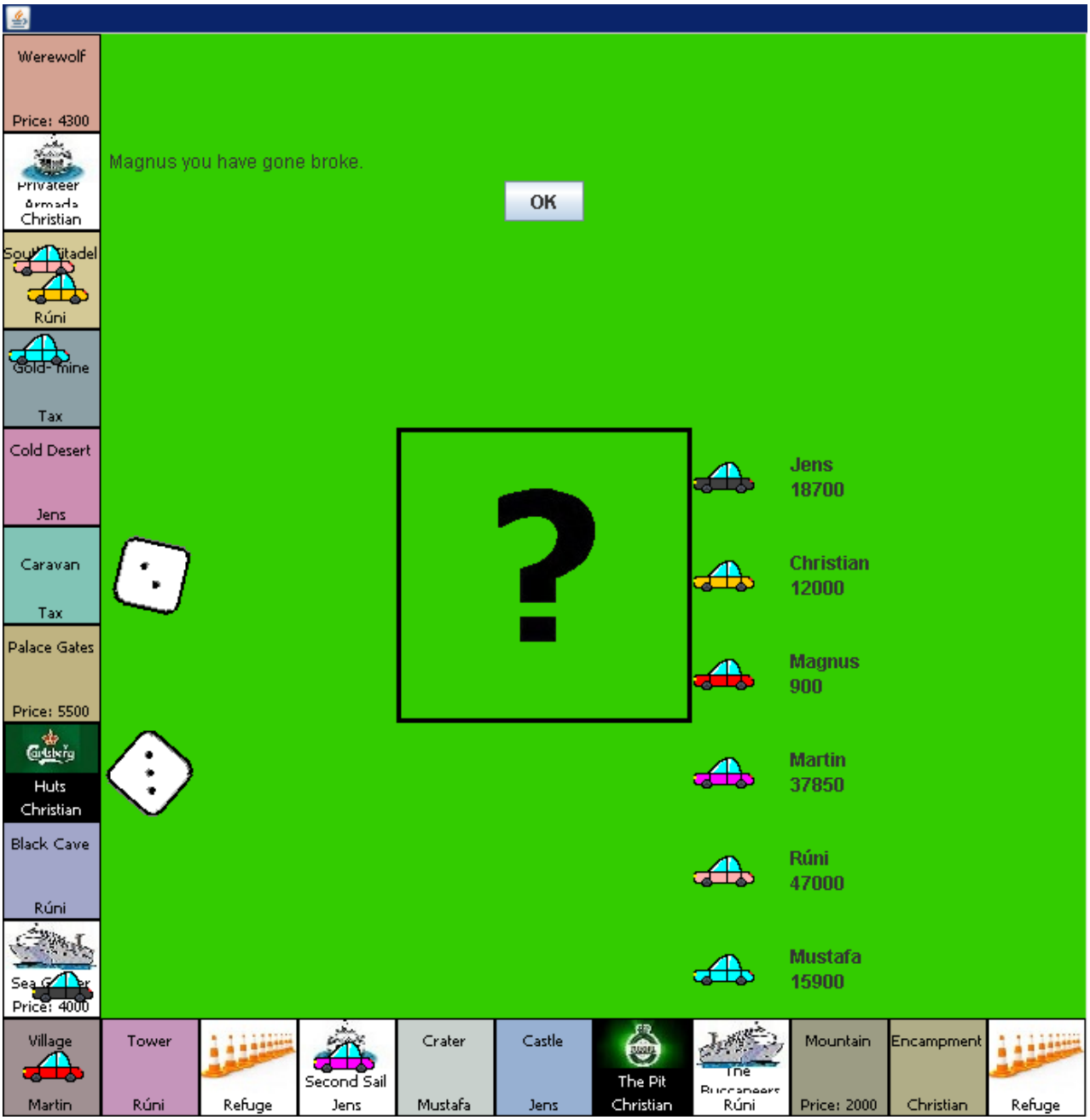

Figure 12: *Bilag 6*: Spilleren går fallit.

Bilag 7

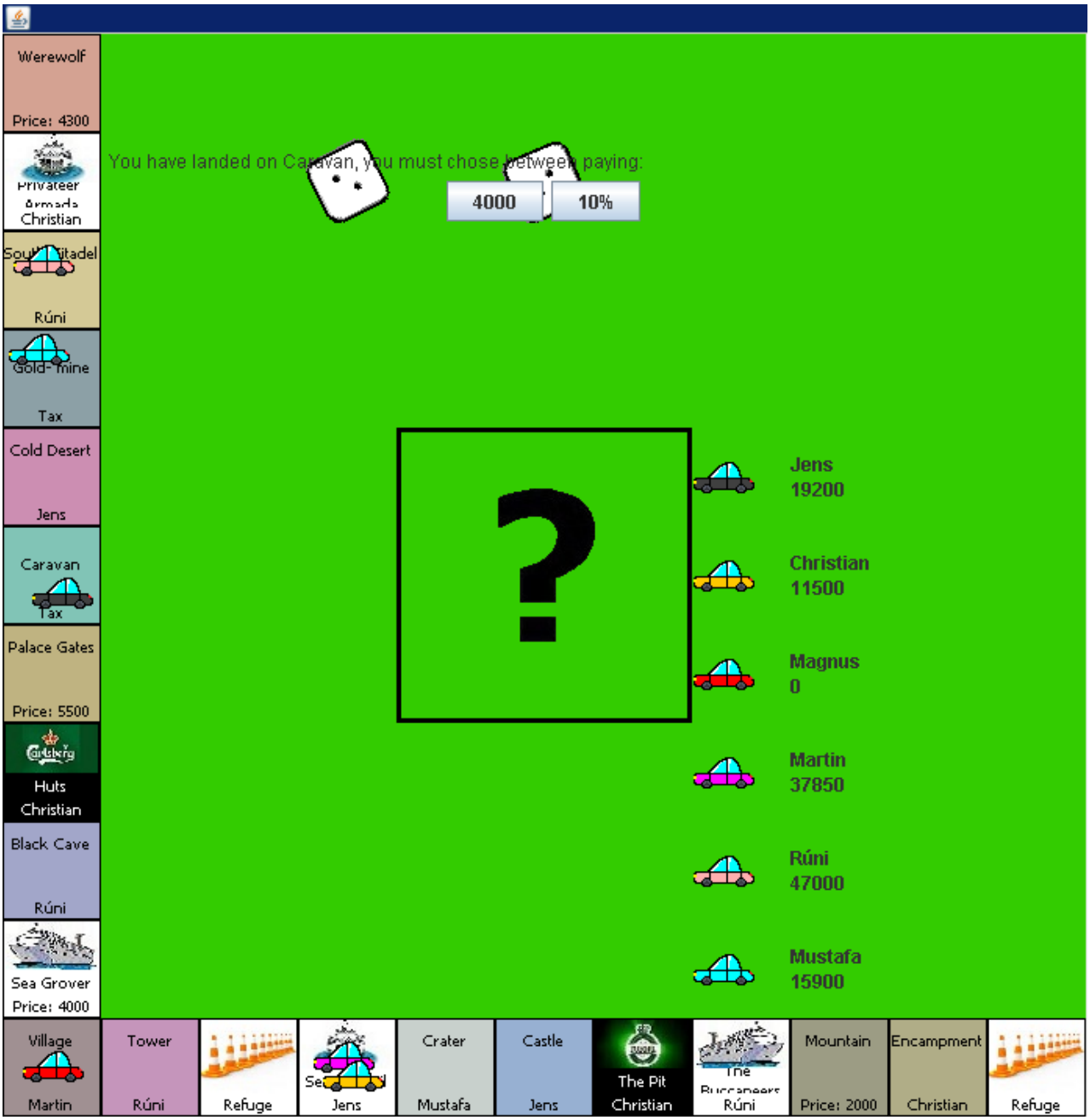

Figure 13: *Bilag 7*: Spilleren lander på et variabelt 'Tax' felt.

Bilag 8

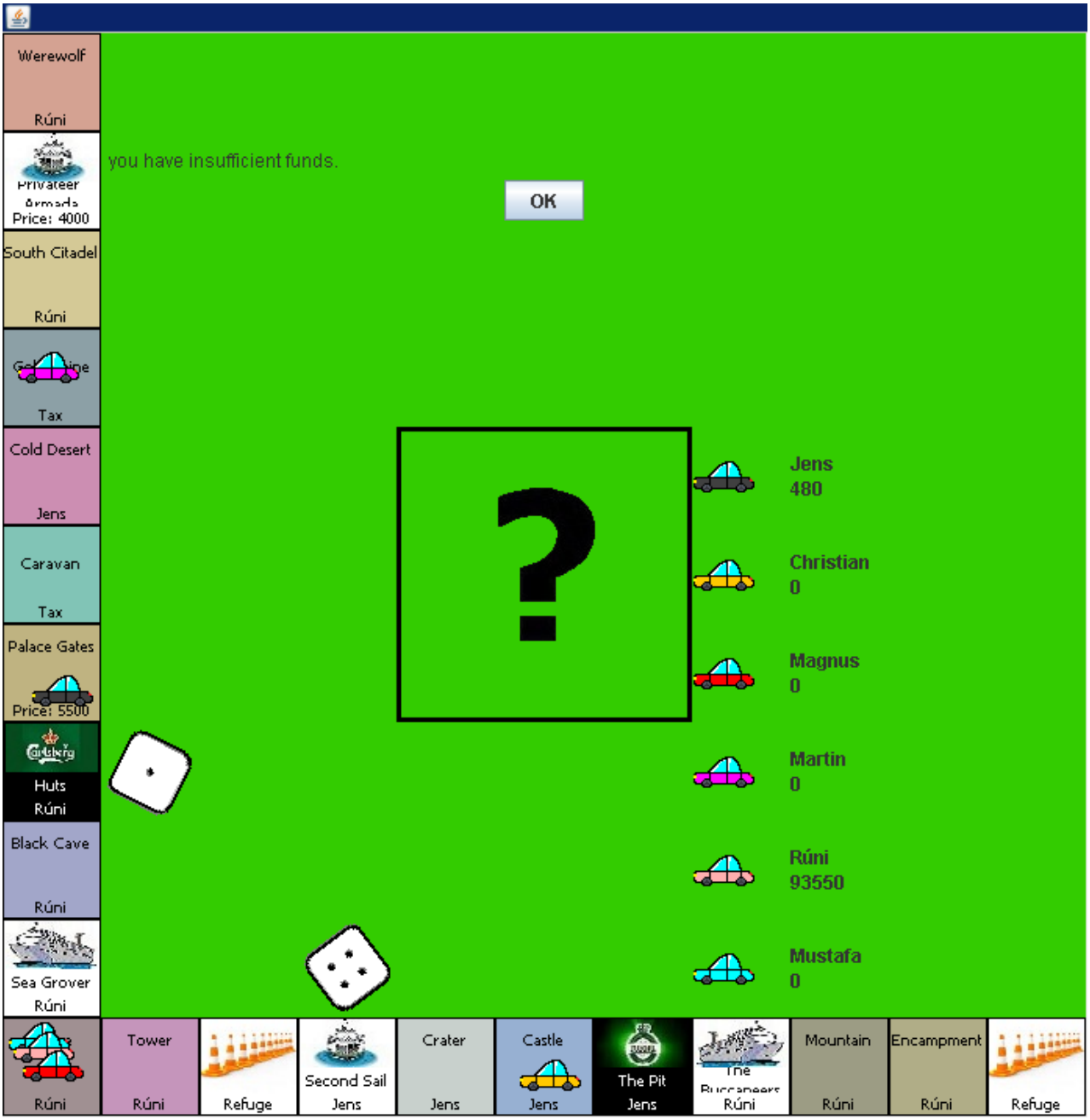

Figure 14: *Bilag 8*: Spilleren har ikke råd til at købe feltet.

Bilag 9

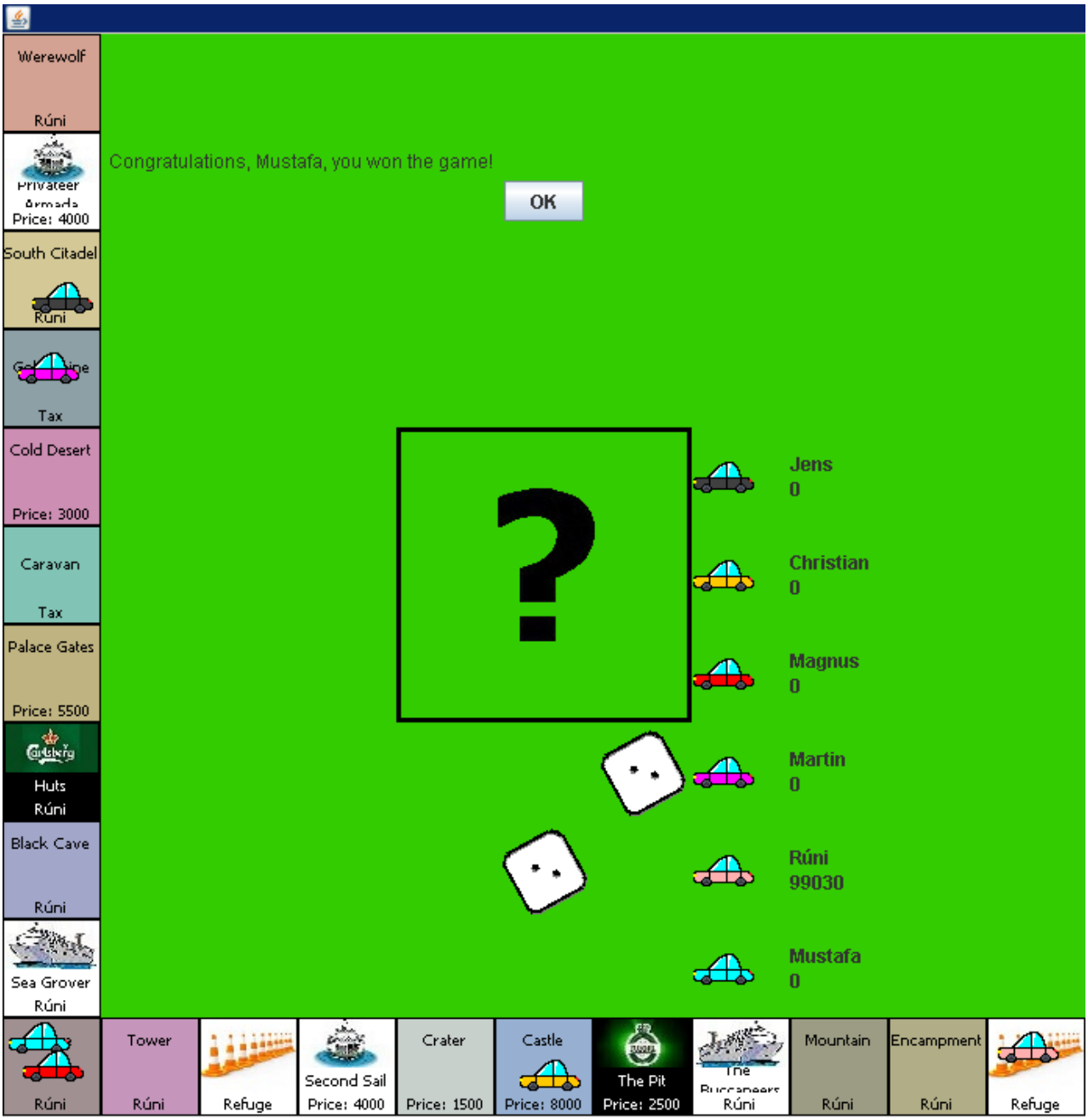

Figure 15: *Bilag 9*: Fejlagtig udnævnelse af vinderen - rettet i senere udgave.

Bilag 10

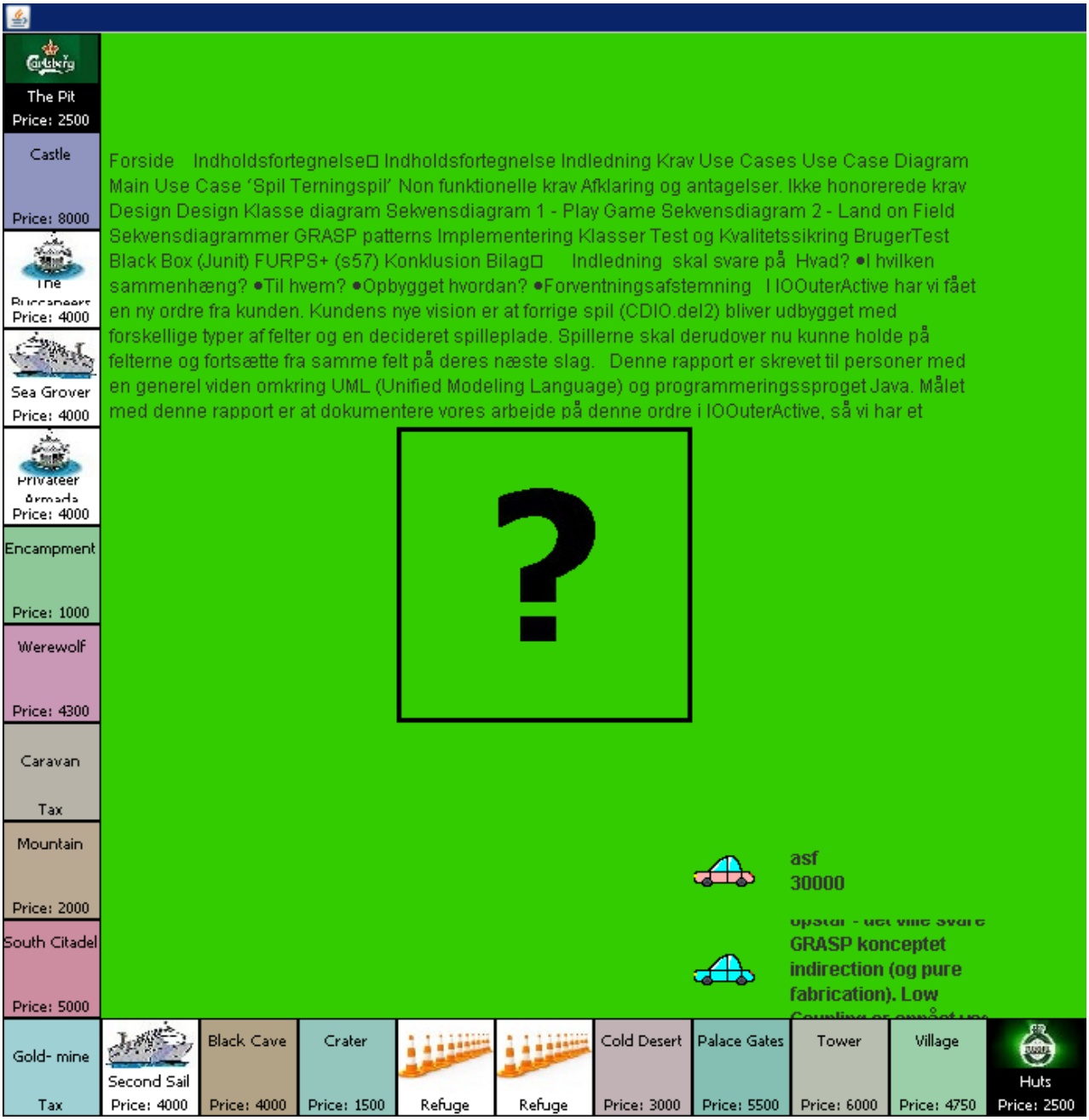

Figure 16: *Bilag 10*: Spilleren har valgt et meget langt navn - Hvilket får ok knappen til at 'forsvinde' og gør spillet uspilleligt.

Bilag 11

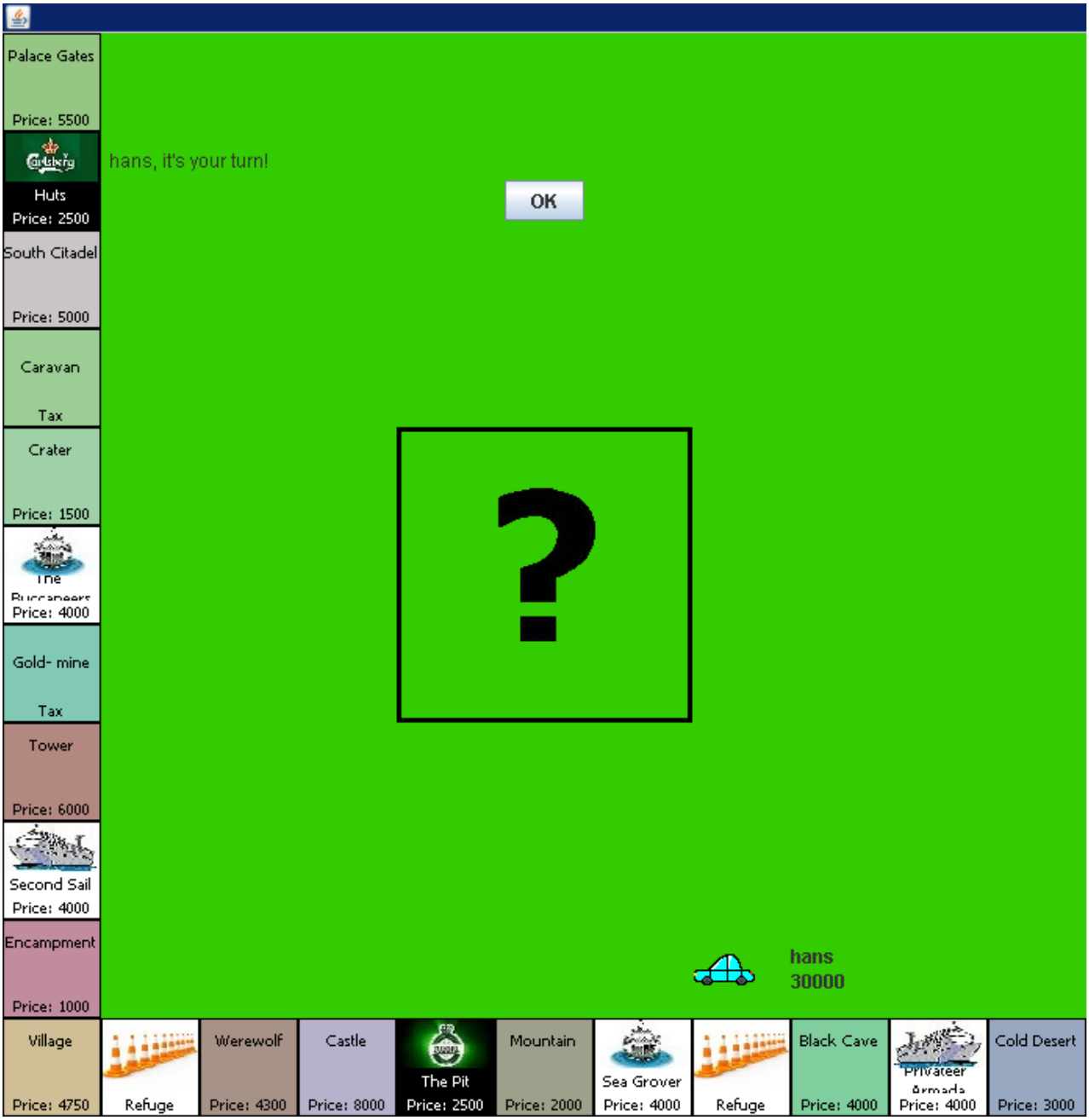

Figure 17: *Bilag 11*: Spillerne har valgt ens navne - Hvilket GUI'en ikke kan håndtere.