CDIO - 4

Et matador spil i java.

Af gruppe 33

Simon Engquist
s143233

Arvid Langsø
s144265

Mikkel Lund
s165238

Jeppe Nielsen
s093905

Mads Stege
s165243

Danmarks Tekniske Universitet
DTU Compute

16. Januar 2017 - Kl. 12:00

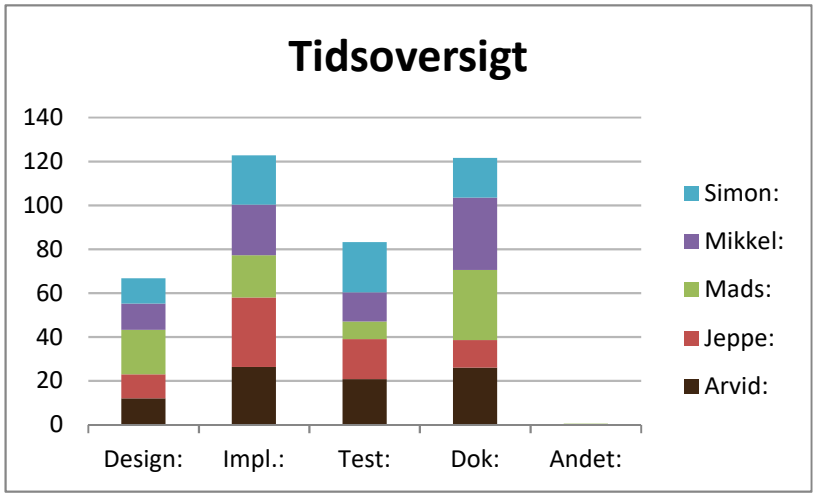
Side antal (med bilag): 147

Kurser: 02312

Time Regnskab:	Ver. 2017-01	Tidsplan for CDIO del 4 - 2017					
	Der findes et samlet overblik i bunden af tidsplanen!						
Dato	Deltager	Design:	Impl.:	Test	Dok.:	Andet:	I alt:
2017-01-02	Arvid	6	0	0	0	0	6
	Jeppe	6	0	0	0	0	6
	Mads	5,75	0	0	0,25	0	6
	Mikkel	6	0	0	0	0	6
	Simon	6	0	0	0	0	6
2017-01-03	Arvid	6	1,5	0	0	0	7,5
	Jeppe	5	0,5	0	0	0	5,5
	Mads	6	0	0	0	0	6
	Mikkel	6	1	0	0	0	7
	Simon	5,5	0	0	0	0	5,5
2017-01-04	Arvid	0	7,5	0	0	0	7,5
	Jeppe	0	7,5	0	0	0	7,5
	Mads	0	7,75	0	0,25	0	8
	Mikkel	0	7	0	0	0	7
	Simon	0	11	0	0	0	11
2017-01-05	Arvid	0	6,5	0	0	0	6,5
	Jeppe	0	4	0	0	0	4
	Mads	0	7	0	0	0	7
	Mikkel	0	9	0	0	0	9
	Simon	0	5	0	0	0	5
2017-01-06	Arvid	0	7	0	0	0	7
	Jeppe	0	7	0	0	0	7
	Mads	0	3,5	0	2	0	5,5
	Mikkel	0	3	0	0	0	3
	Simon	0	4	0	0	0	4
2017-01-07	Arvid	0	0	0	0	0	0
	Jeppe	0	0	0	0	0	0
	Mads	0	0	2	0	0	2
	Mikkel	0	2	0	0	0	2
	Simon	0	0	0	0	0	0
2017-01-08	Arvid	0	0	0	0	0	0
	Jeppe	0	0	0	0	0	0
	Mads	0	0	1	0	0	1
	Mikkel	0	1	0	0	0	1
	Simon	0	2,5	0	0	0	2,5
2017-01-09	Arvid	0	3,75	3,75	0	0	7,5
	Jeppe	0	0	5,5	0	0	5,5
	Mads	5,5	0	0	0	0,5	6
	Mikkel	0	0	0	0	0	0
	Simon	0	0	3	0	0	3
2017-01-10	Arvid	0	0	6	0	0	6
	Jeppe	0	3	3	0	0	6
	Mads	0	0	1	4,5	0	5,5
	Mikkel	0	0	6,25	0	0	6,25

	Simon	0	0	5,5	0	0	5,5
2017-01-11	Arvid	0	0	7	0	0	7
	Jeppe	0	6,5	0	0	0	6,5
	Mads	2	1	2	0	0	5
	Mikkel	0	0	0	7	0	7
	Simon	0	0	5,5	0	0	5,5
2017-01-12	Arvid	0	0	2	5,5	0	7,5
	Jeppe	0	3,25	3,25	0	0	6,5
	Mads	1	0	2	3,5	0	6,5
	Mikkel	0	0	7	0	0	7
	Simon	0	0	5,5	0	0	5,5
2017-01-12	Arvid	0	0	2	6	0	8
	Jeppe	0	0	6,5	0	0	6,5
	Mads	0	0	0	6	0	6
	Mikkel	0	0	0	7,5	0	7,5
	Simon	0	0	3,5	3,5	0	7
2017-01-13	Arvid	0	0	0	7	0	7
	Jeppe	0	0	0	6,5	0	6,5
	Mads	0	0	0	6,5	0	6,5
	Mikkel	0	0	0	7,5	0	7,5
	Simon	0	0	0	7,5	0	7,5
2017-01-14	Arvid	0	0	0	0	0	0
	Jeppe	0	0	0	0	0	0
	Mads	0	0	0	2	0	2
	Mikkel	0	0	0	4	0	4
	Simon	0	0	0	0	0	0
2017-01-15	Arvid	0	0	0	4	0	4
	Jeppe	0	0	0	2	0	2
	Mads	0	0	0	3	0	3
	Mikkel	0	0	0	3	0	3
	Simon	0	0	0	3	0	3
2017-01-16	Arvid	0	0	0	3,5	0	3,5
	Jeppe	0	0	0	4	0	4
	Mads	0	0	0	4	0	4
	Mikkel	0	0	0	4	0	4
	Simon	0	0	0	4	0	4
	Sum	66,75	122,8	83,3	121,5	0,5	333,25

Tidsskema (Gruppemedlem):	Design:	Impl.:	Test:	Dok:	Andet:	I alt:
Arvid:	12	26,25	20,8	26	0	85
Jeppe:	11	31,75	18,3	12,5	0	73,5
Mads:	20,25	19,25	8	32	0,5	80
Mikkel:	12	23	13,3	33	0	81,3
Simon:	11,5	22,5	23	18	0	75



Indholdfortegnelse

1 Abstract

This report describes the development process for a digital version of the Danish board game Matador. The goal is to program a computer game that allows players to play Matador on one computer. The game should maintain the feel of the board game, while also using the benefits of modern technology.

To achieve this, modern software development approaches such as OOAD and GRASP has been employed. Using the BCE model makes the software versatile, allowing for easy changes to the software in future development. The software has been written in the OO language Java, which allows it to be run on most systems independent of the operating system.

In conclusion the game is fully functional and some non critical features have been cut rather than poorly implemeneted. The code has been thoughtfully written, and should be easy to understand for experienced programmers. The program has been tested thoroughly and no major bugs remain.

2 Indledning

Gruppe 33, som arbejder for IOOuterActive, har fået tildelt en sidste opgave. Der skal laves et matadorspil der opfylder kundens vision. Kunden vil hellere have, at lidt er implementeret og alt virker, fremfor at alt er forsøgt implementeret og ingenting virker. Gruppen skal inddrage hvad gruppen har lært i de foregående opgaver hos kunden og genoverveje arkitekturen fra det samlede system.

Det er desuden valgt at lave matadorspillet efter det fysiske matadorspil fra Alga, som blev udgivet i 2002.

Spillet programmeres i Java ved hjælp af Eclipse og der anvendes Github og GitKraken til versionering og distribuering.

Der er i dette projekt lagt stort fokus på at skrive god og letlæselig kode. Dette medfører at mindre vigtige elementer såsom nogle UML-diagrammer og omfattende JUnit testing ikke er blevet prioriteret ligeså højt i dette projekt, som i de tidligere projekter. I forhold til testing har gruppen haft fokus på black-box-testing, med enkelte JUnit-tests på de vigtigste kodeområder i simple klasser.

3 Problemformulering

Kunden har stillet opgaven, at fremstille et program der simulerer spillet Matador. Det er dog ikke blevet specificeret hvilken udgave af spillet der skal implementeres. Det er derfor besluttet at tage udgangspunkt i et Matadorspil fra 2002 fremstillet af Alga. Kunden har ytret at de features der bliver implementeret i spillet skal være fuldt funktionelle. Der må altså ikke ligge features, som er fyldt med fejl der gør spillet uspilleligt.

Det blev fastlagt ved et møde, at følgende funktioner som minimum skulle implementeres:

1. Basale spil funktioner. Spiller bevægelse og win conditions.
2. De tre grundtyper: Ejendom, Tapperi og redderi.
3. Skattefelterne på pladen.
4. Fængslets funktioner.
5. Simple chancekort: Bevægelse af spillere, modtag/betal penge mellem de enkelte spillere.
6. Huse og Hoteller og deres ændring i grundenes leje.

Kunden forventer at disse features er implementeret og gennemtestet. Kunden ønsker desuden dokumentation over koden så fremtidige udviklere kan arbejde videre på systemet. Kunden forventer yderligere at systemet kan køre på en standard computer fra DTU's databarer.

4 Krav

Kravspecifikationen er lavet ud fra spillereglerne i det fysiske matadorspil. Den fulde kravspecifikation kan findes i bilag afsnit 12.1. I kravspecifikationen er nogle af kravene overstreget med grå farve. Disse krav er ikke implementeret i den udgave af spillet som denne rapport beskriver.

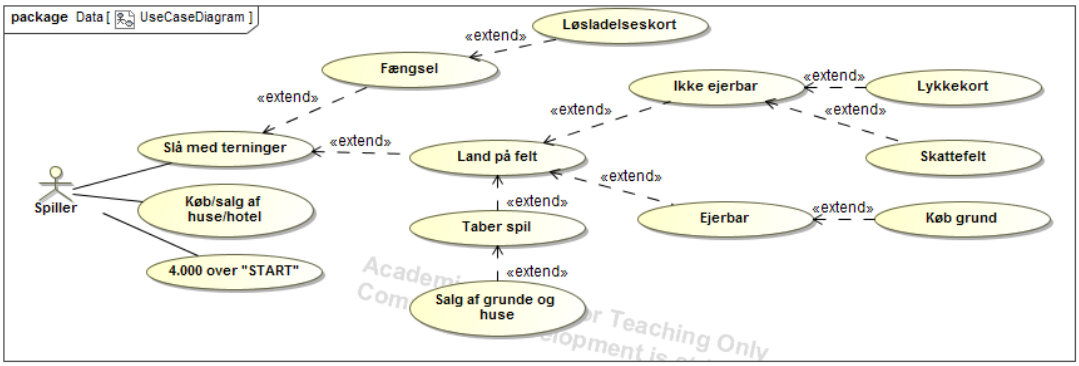
Nedenstående liste giver et kort overblik over de features der er implementeret i programmet.

1. Basale spil funktioner. Spiller bevægelse og win conditions.
2. De tre grundtyper: Ejendom, Tapperi og redderi.
3. Skattefelterne på pladen.
4. Fængslets funktioner.
5. Simple chancekort: Diverse typer bevægelse af spillere, modtag/betal penge for en enkelt spiller.
6. Huse og Hoteller.
7. Salg af bygninger og grunde ved betaling af gæld.

Ud fra listen kan det ses at de ønskede features er blevet implementeret. Yderligere er der implementeret salg af bygninger og grunde til banken. Dette træder i kraft når spilleren skal betale en gæld der er større end spillerens daværrende pengebeholdning.

5 Design

5.1 Use-case diagram

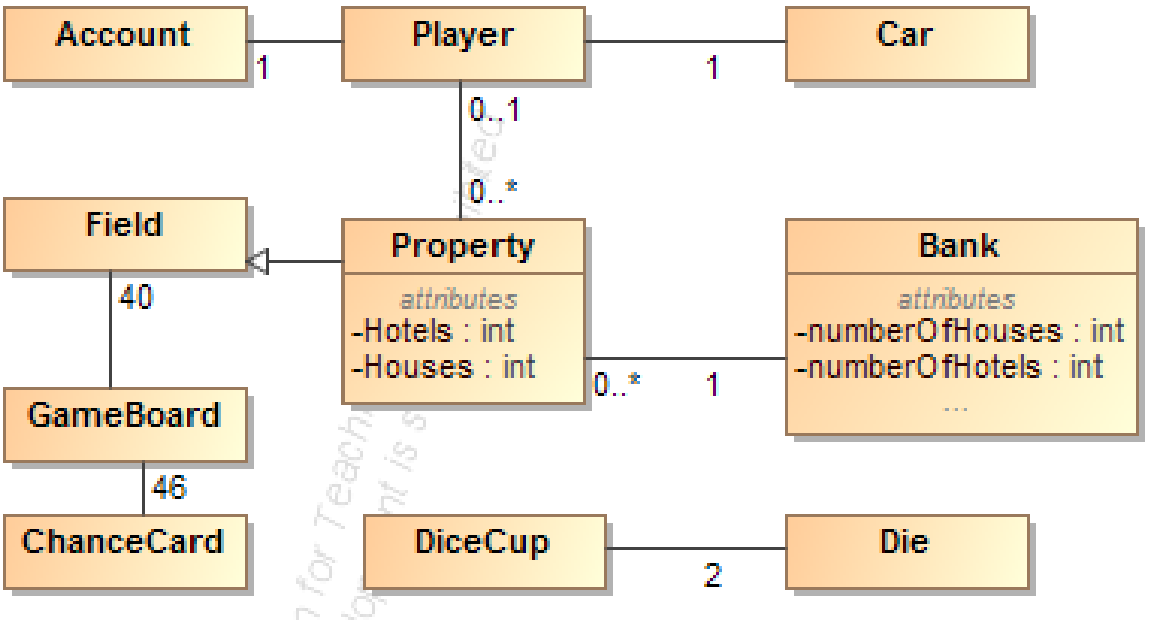


Figur 3: Use-Case diagram for programmet.

I starten af analyse fasen, er der blevet udarbejdet et detaljeret use-case diagram. Diagrammet skaber et overblik over de ting der skal laves, men der er dog mange af use-casene der gør meget lidt, eksempelvis ”slå med terningerne” use-caset. Dette use-case medfører dog at spilleren lander på et felt, og derfor forlænges dette use-case med et ”land på felt” use-case. ”Land på felt” use-casen forlænges så til flere use-cases, netop de type felter der kan landes på.

Use-casene forgrener sig altså ud til de use-cases som er nødvendige. Det kan ses på diagrammet hvordan dette sker flere steder i programmet. Ved at fremstille use-casen på denne måde gør det nemt at overskue hvilke dele programmet skal bestå af.

5.2 Domænemodel



Figur 4: Domænemodel

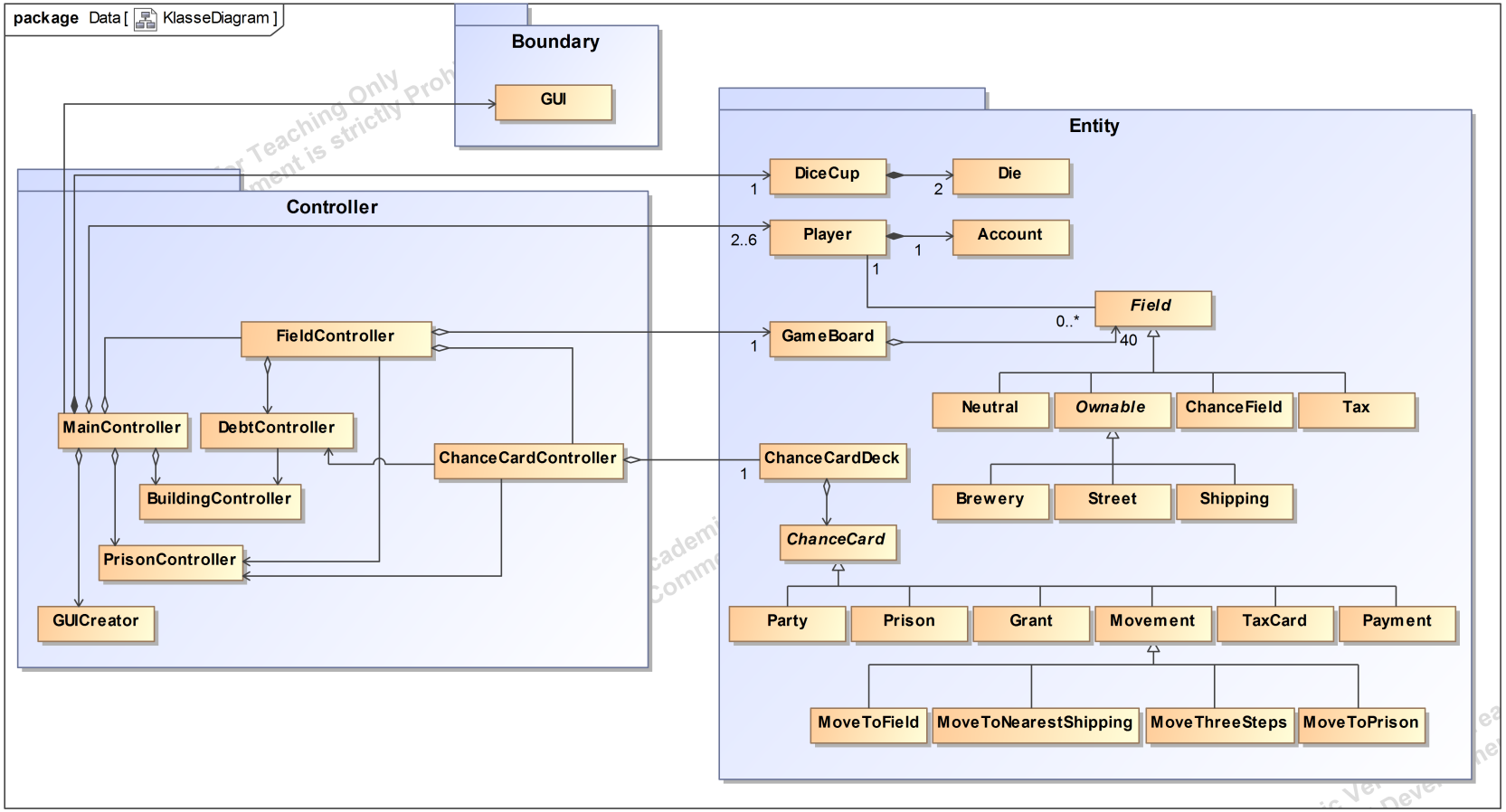
Domænemodellen beskriver det domæne, som der arbejdes med. Her er domænet matadorspillet. Matadorspillet har et spillebræt (GameBoard) med 40 felter (Fields) og ellers ligger der nogle 'Prøv-lykken'-kort (ChanceCard) på spillebrættet.

Nogle af felterne kan ejes (Ownable) af spilleren (Player). På nogle af de ejerbar felter, ejendommene (Property), kan der bygges bygninger (hotels/houses). Bygninger købes af spilleren (Player), som har en konto (Account) med penge (balance). Derudover har spilleren (Player) én brik (Car), som symboliserer hvor på spillepladen (GameBoard) spilleren befinder sig.

Spilleren køber bygningerne af banken (Bank). Banken (Bank) holder blandt andet styr på køb og salg af ejerbar grunde (Ownable) og antallet af huse og hoteller (numberOfHouses/numberOfHotels) i spillet. Spilleren bevæger sig rundt på spillebrættet ved at slå med terningerne (Die) i raflebægeret (DiceCup).

Når domænet er fastlagt er det lettere at danne sig et overblik over hvad og hvordan softwaren skal designes.

5.3 Klassediagram overblik



Figur 5: Simpelt klasse diagram.

Beskrivelse og BCE

På (figur 5) kan det ses at MainControlleren længst til venstre har de fleste af controllerne. Det skyldes at maincontrolleren er der hvor programmet starter. MainControllere aggerer derfor som opgavefordeler for de andre controllere. GUI er markeret forkert på dette diagram, da GUI'en er statisk og der er kun tale om en 'dependency'. Generelt snakker alle controllerne med GUI'en. Programmet er opbygget efter BCE modellen. Det er gjort tydeligt ved hjælp af pakke navnene. Det gør det nemt at udskift dele af programmet, eksempelvis kan man nemt erstatte GUI'en med en pænere GUI. Dette kan man gøre ved at lave en ny GUI med de samme metoder der gør det samme, men hvor GUI'en blot behandler informationen på en anden måde.

5.4 Design Klasse Diagram

Der er fremstillet et klassediagram som viser hvordan programmet er bygget op. Diagrammet kan ses senere i afsnit 5.4. Det er opbygget efter BCE, som var tydeligt på det forrige diagram. Klassernes funktioner bliver beskrevet yderligere i implementerings afsnittet. Overordnet kan man se på diagrammet at MainControlleren har ansvaret for oprette mange af de andre klasser, og den har ansvaret for at styre spillet. Hvis der skal ske en mere kompleks opgave, så kalder den de andre controllere.

Klassediagrammet mangler en masse dependency pile. De er blevet udeladt for at overskueliggøre diagrammet. I metode kaldene kan man generelt se hvilke metoder der kræver hvilke typer af klasser. Player klassen bliver desuden brugt af alle controllerne, bortset fra GUICreator. Dermed er alle controllerne dependent på player. Field klassen bliver også brugt i flere controllere og de er derfor også dependent på field.

Programmet er blevet designet til at passe til en GUI givet af DTU. Da GUI klassen har alle de funktioner der normalt bliver implementeret i boundary klasser, bliver GUI klassen brugt som boundary. GUI klassen er statisk. Derfor kan den kaldes af alle klasser. Derfor er det blevet valgt at de forskellige controller har ansvar for at outputte de ting de gør til GUI'en.

Der er et par ting der kan ændres på diagrammet. Fx er der mange klasser der indeholder hjælpe metoder. Eksempelvis MainController.addReturnToArray, og metoden i Buidlingcontroller der finder min/max af flere integer værdier. Disse burde have ligget i en hjælpe klasse og alle være statiske. På den måde kunnne alle klasserne kalde dem.

På næste side er indsat det fulde klassediagram i A2 format.

package Model [Designklasse diagram]

boundary

GUI

controller

GUICreator
attributes
-cars [*]
-fields : Field [*]
-fieldCounter : int
operations
+beginBoardBuilding() : void
+endBoardBuilding() : void
+addField(String[]) : void
-addPlayersToGUI(String[]) : void
-askNumberOfPlayers() : int
+getPlayerNames() : String [*]
-createCars()
-getFieldColor(String)

MainController
attributes
-players : Player [2..6]
-dice : DiceCup
-prisonController : PrisonController
-fieldController : FieldController
-buildingController : BuildingController
-turn : int
-numExtraTurn : int
-extraTurn : boolean
operations
+main(String[]) : void
+changeTurn() : void
+playTurn() : void
+playGame() : void
+checkForExtraTurn() : boolean
+givePlayer4000() : void
+movePlayerOnGUI() : void
+rollDice() : int
+removePlayerOnGui(String) : void
+movePlayer(int) : void
+movePlayerTo(int) : void
+playerTurnDecision() : void
+addReturnToArray(String[]) : String [*]
+checkForWinner() : void

BuildingController
attributes
-houses : int
-hotels : int
operations
+changeHouses(int) : void
+changeHotels(int) : void
+countStreetColours(Player) : int [*]
-canBuildOnColour(Player) : boolean [*]
-canBuildOnColoursString(Player) : String [*]
-streetsWithColour(String) : int
-minimum(int[]) : int
-maximum(int[]) : int
-streetsWithBuildings(Player, String, boolean) : String [*]
-setBuilding(Player, String) : void [*]
-setHotel(Player, Street) : void [*]
-setHouse(Player, Street) : void [*]
+sellHouse(Player, String) : void
+buyBuildingMenu(Player) : void
+sellBuildingMenu(Player) : void

PrisonController
attributes
-mainController : MainController
-debtController : DebtController
operations
+sendToPrison(Player) : void
+inPrison(Player) : boolean
+checkInPrison(Player) : boolean

ChanceCardController
attributes
-deck : ChanceCardDeck
-prison : PrisonController
-debt : DebtController
-main : MainController
operations
+draw (Player) : void

FieldController
attributes
-prisonController : PrisonController
-chanceCardController : ChanceCardController
-debtController : DebtController
-gameBoard : GameBoard
-doubleRent : boolean
operations
+landOnField(Player, int) : void
+setDoubleRent(boolean) : void

DebtController
attributes
-buildingController : BuildingController
operations
-canStreetBeSold(Street, Player) : boolean
-getFieldsWithoutBuildings(Player) : Ownable [*]
-fieldsToString(Ownable[]) : String [*]
+sellField(Player) : void
+playerAffordPayment(Player, int) : boolean
+playerHasLost(Player) : void
+handleDebt(Player, int) : boolean

entity

Player
attributes
-name : String
-account : Account
-hasLost : boolean
-inPrison : boolean
-position : int
-fields : Ownable [28]
-turnsInPrison : int
operations
+payRent(Player, int) : void
+payRent(Player, int) : void
+changeAccountBalance(int Value) : int
+setFields() : void
+removeField(Ownable) : void
+getBrewEriesOwned() : int
+getShippingsOwned() : int
+getStreetsOwned(String colour) : int
+buyField(Field) : boolean
+getFortune() : int
+getStreetFromName(String) : Street
+getHousePriceFromColour(String colour) : int
+changeTurnsInPrison(int) : void

DiceCup
attributes
-d1 : die
-d2 : die
operations
+shake() : void
+getDiceValue() : int [2]

Die
attributes
-MAX_VALUE : int
-value : int
operations
+rollDie() : int

GameBoard
attributes
-fields : Field [40]
-fieldcounter : int
operations
+addField(String[] information) : String
+getField(int fieldNum) : Field

Account
attributes
-balance : int
operations
+changeBalance(int value) : void

entity.chanceCard

ChanceCardDeck
attributes
-chanceCardDeck : ChanceCard [*]
operations
+draw () : ChanceField
+shuffle() : void
-swapTwoCards() : void
-createGrant(String[]) : Grant
-createTaxCard(String[]) : TaxCard
-createPayment(String[]) : Payment
-createParty(String[])
-createPrison(String[]) : Prison
-createMoveThreeSteps(String[]) : MoveThreeSteps
-createMoveToField(String[]) : MoveToField
-createMoveToNearestShipping(String []) : MoveToNearestShipping
-toInt(String) : int
-toBoolean(String) : boolean
-toIntArray(String[]) : int [*]

MoveToPrison
attributes
-sentToPrison : boolean

ChanceCard
attributes
-type : String
-description : String

Grant
attributes
-amount : int

Payment
attributes
-amount : int

MoveThreeSteps
attributes
-steps : int

Movement

MoveToField
attributes
-fieldPosition : int

MoveToNearestShipping
attributes
-shippingPositions : int [4]
-doubleRent : boolean

entity.fields

Field
attributes
-fieldNumber : int
-name : String
-type : String
-description : String
operations
+*getRent() : int*

ChanceField

Neutral

Tax
attributes
-hasTaxRate : boolean
-amount : int

Ownable
attributes
-price : int
-owner : Player
operations
+removeOwner() : void
+setOwner(Player) : void
+isFieldOwned(boolean condition) : boolean
+isFieldOwnedByAnotherPlayer(Player) : boolean
+*getRent() : int*
+getColour() : String
...

Brewery
attributes
-baseRent : int
-diceSum : int
operations
+getRent() : int

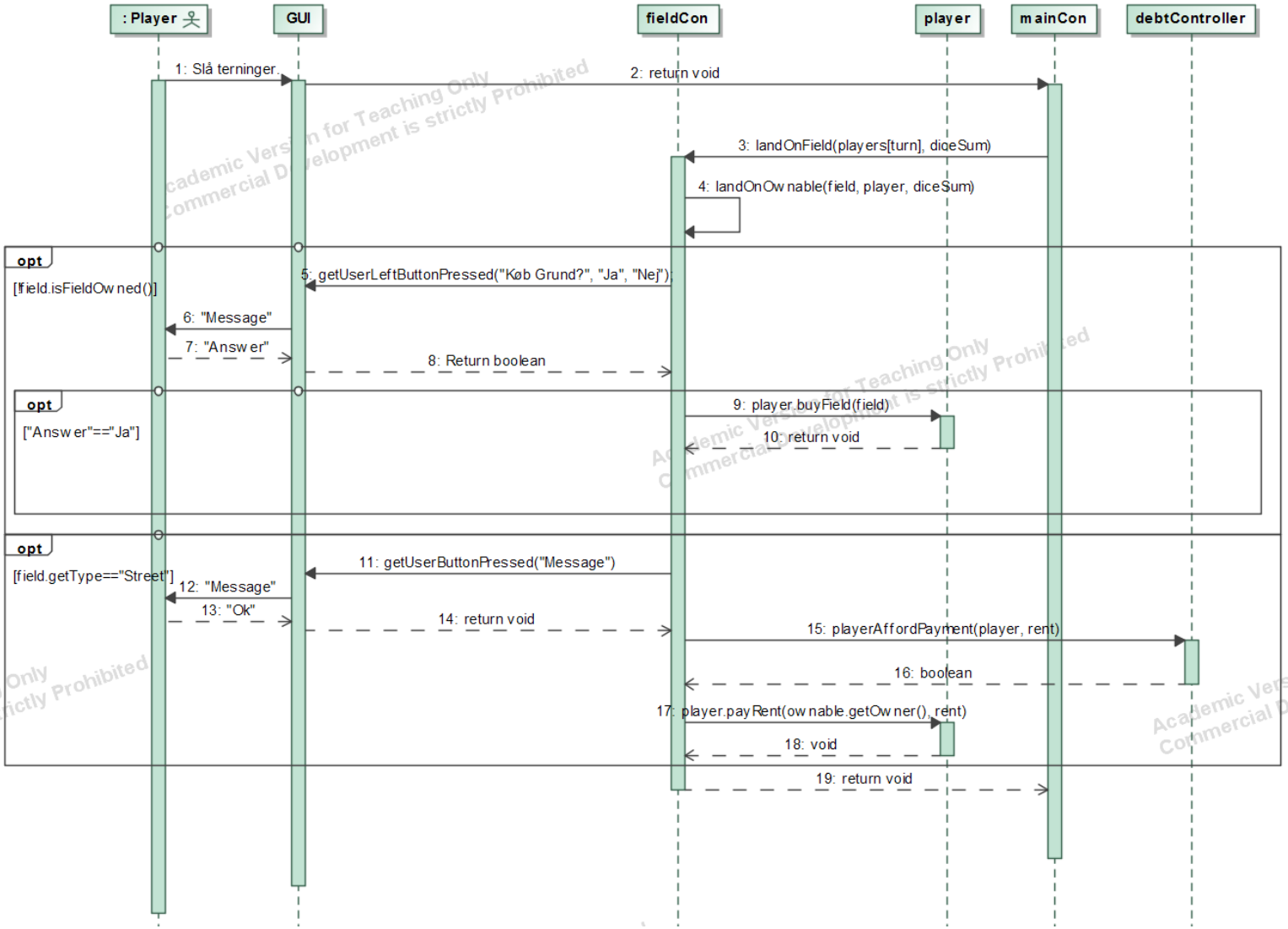
Shipping
attributes
-RENT : int
operations
+getRent() : int

Street
attributes
-colour : String
-baseRent : int
-housePrice : int
-houseRent : int [0..5]
-numbOfHouses : int
-pledge : int
operations
+getHouseRent(int numbOfHouses) : int
+getRent() : int
+getValue() : int
+changeNumbOfHouses(int amount) : int

5.5 Sekvens Diagram

Det er bestemt kun at lave ét sekvensdiagram, da disse tager lang tid at fremstille. Derfor er der her valgt at vise den mest centrale del af systemet. Diagrammet beskriver use-casen 'land på felt'. Sekvensdiagrammet viser kun tilfældet hvor spilleren landet på et felt der kan ejes og feltet er af typen 'ejendom' (kaldet 'Street' i koden).

Før spilleren lander på et felt har spilleren slået med terningerne(1-2). Dernæst rykker MainControlleren spilleren og fortæller FieldControlleren at den skal overtage (3). FieldControlleren ser at der er tale om et ejerbart felt. Derfor kalder den landOnOwnable metoden i sig selv(4). Nu er der to muligheder. Enten er feltet ejet, eller også er det ikke. Hvis det ikke er ejet kan spilleren købe feltet(5-10). Hvis feltet er ejet skal spilleren betale leje. Dette får han at vide (11-14) og dernæst kaldes DebtControlleren. DebtControlleren tjekker om spilleren har råd, eller kan få råd ved at sælge sine bygninger/grunde, til at betale sin leje(15-16). Hvis han havde råd så betales lejen til ejeren af feltet(17-18).



Figur 6: Sekvensdiagram af landOnField

6 Implementering

6.1 Kodestruktur

Projektet er struktureret efter BCE-modellen, hvilket betyder at programmets forskellige klasser er delt op i pakker: Boundaries, Entities, Controllers. Denne konstruktion gør at der let kan udskiftes dele af koden, uden at der skal ændres i de øvrige klasser.

6.1.1 Boundaries

Projektet indeholder ikke nogen boundary pakke, men det forestilles at alle GUI klasser havde ligget i denne pakke, hvis der også skulle udvikles en GUI til projektet. Projektet anvender en GUI, som importeres som en jar-fil. GUI jar-filen ligger i biblioteksmappen "lib".

GUI'en er udviklet af DTU, og er ikke relateret til nogen gruppemedlemmer i dette projekt.

6.1.2 Entities

Entitetspakken indeholder de klasser der repræsenterer de virkelige objekter i matadorspillet. Disse klasser indeholder igen data om objekterne og metoder der håndterer disse data.

Entity pakken har også to underpakker: chanceCards, som indeholder chancekortene, og field, som indeholder de forskellige felter på spillepladen.

6.1.3 Controllers

Controller pakken indeholder alle programmets controller klasser. Klasserne står for at håndtere spillogikken i spillet.

BuildingController

BuildingController står for køb og salg af bygninger. Controlleren har to variable: "houses"og "hotels", der holder styr på hvor mange huse og hoteller der er til rådighed. Controlleren oprettes fra starten med 32 huse og 12 hoteller. Hvis en spiller prøver at købe et hus, og der allerede står 32 på pladen, så får de en besked om at der ikke er flere huse til rådighed.
Når der bygges og sælges huse er det vigtigt i forhold til krav F4.1.4.1, at der ikke sælges eller bygges for mange huse på en grund. Metoden *streetsWithBuildings* står for at dette krav overholdes.
Her er et kodeeksempel fra *streetsWithBuildings* metoden i BuildingControlleren.

```
234     // An array to hold the name of the differents streets that you can
235     // build on.
236     String[] streetNamesNew = new String[amountOfStreets];
237     j = 0;
238     for (int i = 0; i < streetNames.length; i++) {
239       if (streetsWithEqualAmountHouses == houses[i]) {
```

```
240             streetNamesNew[j] = streetNames[i];
241             j++;
242           }
243         }
244         return streetNamesNew;
```

ChanceCardController

ChanceCardController står for at håndtere de forskellige scenarier der skal ske, når der trækkes et "Prøv lykken"-kort.
Trækkes der for eksempel et "Bevæg dig til nærmeste rederi"-kort, kaldes metoden *drawMoveToNearestShipping*. Metoden henter spillerens position og tjekker om spilleren har passeret et af rederi-felterne. Metoden placerer derefter spilleren på det næste i rækkefølgen.
Her er et kodeeksempel fra ChanceCardControlleren på metoden *drawMoveToNearestShipping*:

```
94    private void drawMoveToNearestShipping(ChanceCard currentCard, Player
          player) {
95      MoveToNearestShipping card = (MoveToNearestShipping) currentCard;
96      int[] shippingPos = card.getShippingPositions();
97      int currentPos = player.getPosition();
98
99      // Sets the position of the player to the first shipping field if you
100     // are passed the last shipping field.
101     player.setPosition(shippingPos[0]);
102     // Sets the position of the player to the next shipping field if you
103     // haven't passed the last shipping field.
104     for (int i = 0; i < shippingPos.length; i++) {
105       if (shippingPos[i] > currentPos) {
106         player.setPosition(shippingPos[i]);
107         break;
108       }
109     }
110     mainController.getLandOnFieldController().setDoubleRent(card.
          getDoubleRent());
111     mainController.movePlayerOnGUI();
112     // Hvis start passeres.
113     if (currentPos > player.getPosition()) {
114       mainController.givePlayer4000();
115     }
116     mainController.getLandOnFieldController().landOnField(player, 0);
117   }
```

DebtController

DebtController håndterer fremgangsmåderne når en spiller skylder penge. Hvis en spiller skylder penge, og har en balance mindre end hans gæld, men en formue der er større end gælden, så får han mulighed for at sælge de grunde eller bygninger han ejer. Metoden *playerAffordPayment* håndterer dette scenarie, og anvender metoden *handleDebt*, som viser spillerens muligheder på GUI'en og opererer de forskellige valgmuligheder.
Her er et kodeeksempel fra Debtcontrolleren på metoden *playerAffordPayment*:

```
181   public boolean playerAffordPayment(Player player, int payment) {
182     if (player.getAccountBalance() <= payment) {
```

```
183         return handleDebt(player, payment);
184
185       } else
186         return true;
187     }
```

FieldController

FieldController står for at oprette spillepladen (GameBoard) og hvad der sker når man lander på de forskellige felter. Metoden *landOnField* håndterer dette og kalder de forskellige metoder, alt efter hvilken type felt der landes på. For eksempel hvis der landes på en af grundene der kan ejes, kaldes metoden *landOnOwnable*. Metoden tjekker om feltet er ejet af en anden spiller. Hvis dette ikke er tilfældet, får spilleren der landede på feltet mulighed for at købe feltet, dog hvis han heller ikke selv ejer feltet. Er feltet derimod ejet af en anden spiller, så skal der betales leje til denne.
Her er et kodesempel fra *landOnOwnable* metoden i FieldControlleren:

```
121         // If the player can afford to pay rent.
122         if (bankController.playerAffordPayment(player, rent)) {
123           player.payRent(ownable.getOwner(), rent);
124           GUI.setBalance(ownable.getOwner().getName(), ownable.getOwner().
                getAccountBalance());
125           GUI.setBalance(player.getName(), player.getAccountBalance());
126         }
```

GUICreator

GUICreator opretter GUI'en. Her anvendes Reader klassen (som beskrevet i afsnit 6.1.4) til at oprette felterne på GUI'ens spilleplade. GUICreatoren har en *addField* metode, der kalder metoderne der tilføjer felterne til GUI'en. For eksempel kaldes metoden *addStreet* der tilføjer et ejendoms felt.
Her er et kodesempel fra GUICreator på metoden *addStreet*:

```
118   private void addStreet(String[] fieldData) {
119     Color color = getFieldColor(fieldData[1]);
120     fields[fieldCounter - 1] = new Street.Builder().setTitle(fieldData[0]).
          setSubText(fieldData[2])
121           .setDescription("").setBgColor(color).setRent(fieldData[3]).build()
                ;
122   }
```

PrisonController

PrisonController håndterer alt hvad der har med fængslet at gøre - når en spiller sendes i fængsel, og hvad der sker når spilleren er i fængsel.
Metoden *inPrison* håndterer hvad der sker når spilleren er i fængsel. Hvis en spiller er i fængsel og det er hans tur, så får han mulighederne for at betale sig ud af fængslet eller at slå sig ud med terningerne.
Her er et kodeeksempel fra *inPrison* metoden i PrisonController:

```
77     // If the player chooses to pay himself out, release him from jail.
78     if (userDecision.equals(payOut)) {
79       if (bankController.playerAffordPayment(player, 1000)) {
80         player.changeAccountBalance(-1000);
```

```
81            GUI.setBalance(player.getName(), player.getAccountBalance());
82            player.setInPrison(false);
83            boughtOut = true;
84          }
85        }
```

Det er blevet besluttet at spillere som har siddet i fængsel tre ture i træk, og som ikke kan betale sig ud af fængslet, bliver frigivet uden betaling. Man argumenterer for denne spilmekanik ved at sige at den fængslet spiller er gået glip af tre terningekast, og derfor er bag de andre spillere.

MainController

MainController opretter størstedelen af controllerne i programmet og fordeler opgaverne ud til disse. Det er også MainController der kører spillet. Klassens *main* metode står for at køre spillet. Metoden opretter MainController'en selv og kalder *playGame* metoden. *playGame* kalder *changeTurn* metoden der sørger for at skifte tur imellem spillerne og *playTurn* metoden der står for at spille en tur.
Her er et kodeeksempel fra MainControlleren på *playGame* metoden:

```
140   public void playGame() {
141     GUI.getUserButtonPressed("En tilfældig spiller er valgt til at starte",
            "Start spil");
142     // Keep changing turn until someone has won.
143     while (true) {
144       changeTurn();
145       // If a winner is found delcare him winner and close the game.
146       if (checkForWinner() != null) {
147         GUI.getUserButtonPressed("Tillykke " + checkForWinner().getName() +
                " har vundet.", "Sweet");
148         GUI.close();
149         break;
150       }
151
152       do {
153         playTurn();
154       } while (extraTurn && !players[turn].getHasLost());
155
156     }
157   }
```

6.1.4 Data

Programmet har også en data pakke. Data pakken indeholder en "Reader" klasse der står for at indlæse alle spillepladens feltinformationer fra en text fil (Feltliste.txt). Reader klassen opretter derefter et String array med disse data. Dette array bliver blandt andet anvendt af GameBoard klassen, som opretter alle felter som objekter.

6.2 GRASP

Det er en god idé at strukturere sit program efter BCE-modellen for at opnå god og letlæselig kode. En udviklingsmetode der er anvendt til dette er GRASP.
GRASP (General Responsibility Assignment Software Patterns) er en kollektion af objekt-orienteret design patterns. Der er i programmet fokus på anvende følgende patterns: Information Expert, Creator, Polymorphism, Controller, Low Coupling og High Cohesion.

Information Expert

'Information Expert' går ud på at gøre klassen fokuseret. Dette betyder, at klassen kun indeholder informationer inden for klassens ansvarsområde. Tager man udgangspunkt i 'Field'-klasserne, så ved de ikke noget om den spiller der står på det pågældende felt. Denne information indeholder 'Player'-klassen til gengæld. Det ville være ulogisk, hvis feltet eksempelvis kendte spillerens navn.

Creator

'Creator' går ud på at finde ud af hvilken klasse der skal oprette et specifikt objekt. For eksempel bliver alle 'Field'-objekterne oprettet i GameBoard, hvilket giver god mening, når man kigger på en virkelig spilleplade, da en spilleplade har felter.

Controller

'Controller' går ud på at finde ud af hvilke klasser der skal håndtere/kontrollere de forskellige hændelser der sker i spillet. Controllers er en vigtig del af BCE-modellen. De fleste af programmets controllere håndterer logikken bag hver deres use-case. For eksempel håndterer PrisonController hvad der sker i use-casen "Fængsel" (se figur 3)

Polymorphism

Polymorphism går ud på at udnytte arv. Tager man udgangspunkt i 'Field'-klasserne, så nedarver 'Field'-klassen alle sine egenskaber til alle andre felter. Hvilket gør, at man f.eks. kan smide et 'Street'-felt og et 'Shipping'-felt i et 'Field array'. Dette gør det muligt at benytte en metode, de tre klasser 'Field', 'Street' og 'Shipping' har tilfælles uden at skulle 'caste' fra en klasse til en anden. For eksempel kan man benytte 'getRent' metoden på et 'Ownable'-array indeholdende både 'Street' og 'Shipping'-felter, da alle tre klasser har 'getRent' metoden.

Low Coupling

'Low Coupling' går ud på at skille klasserne ad, så man opnår lavere afhængighed mellem de forskellige klasser. Dette betyder, at det er meget lettere at tilføje, fjerne og redigere features/klasser i programmet. Hvis man foretager ændringer i klasserne er der dermed også færre klasser der bliver påvirket. Det er derfor nemt at ændre i klasserne.

High cohesion

'High Cohesion' står for høj samhørighed. Hvilket betyder at hver enkelt klasse er fokuseret, håndterbar og forståelig. Klassen har altså ansvar for en bestemt del af koden og kender ikke til de dele af koden, som ikke er relevant for klassen.

Når man gør brug af low coupling og high cohesion gør det programmet nemt at ændre og lettere foreståeligt for fremtidige udviklere.

7 Minimumskrav

Softwaren kan køre på en hvilken som helst nyere pc, som har den korrekte software installeret. Da IOOuterActive forventer at programmet kan køres på computerne i DTU's databarer, er disse sat som minimumkravet til at kunne køre programmet.

8 Installationsguide

Installationsguiden beskriver hvordan programmet kan køres via Eclipse og den medfølgende .bat fil. Softwaren kan køre på andre IDE'er end Eclipse, men dette er ikke beskrevet her.

Uanset hvordan du vælger at køre programmet, skal du første hente den nyeste version af Java (hvis du vælger af bruge Eclipse skal du også hente Java JDK). Derudover skal der hentes zip-filen 33_CDIO4 fra campusnet.dtu.dk.

Kørsel via Eclipse

Hvis du gerne vil køre programmet ved hjælp af Eclipse skal du første hente den nyeste version af Eclipse. Projektet er skrevet i Eclipse v. 4.6.2.

Følg derefter instrukserne herunder:

1. Start Eclipse.
2. Tryk på 'File' i øverste venstre hjørne og vælg 'Import'.
3. I det nye vindue skal du indtaste 'Existing' i søgefeltet.
4. Vælg herefter 'Existing Projects into Workspace'.
5. Tryk 'Next'.
6. Markér 'Select archive file' og Tryk på browse ud for denne
7. Vælg '33_CDIO4'.
8. Tryk på 'Finish'. Du har nu importeret projektet i Eclipse.
9. Tryk på mappen '33_CDIO4' i 'Package Explorer', vælg derefter 'src' $\rightarrow$ 'controller'.

10. Dobbelt-klik på 'MainController.java'.
11. Højreklik herefter og vælg 'Run as' → 'Java application'.

Kørsel via .bat fil (Kun windows)

Følg instrukserne herunder:

1. Unzip mappen 33_CDIO4.
2. Åben 33_CDIO4
3. Åben mappen 'Installation'.
4. Dobbelt-klik derefter på filen der hedder "33_CDIO4.bat" for at starte spillet.

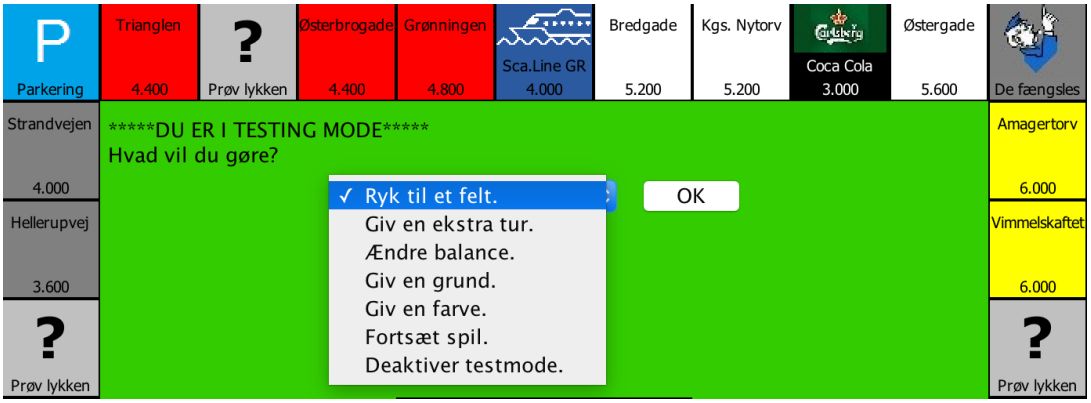
9 Test

9.1 TestModeController

I forbindelse med udviklingen af programmet, er det blevet besluttet også at udvikle en test klasse kaldet TestModeController. Denne klasse er lavet med henblik på at gøre det nemt og hurtigt at black-box-teste softwaren i forskellige test cases. Hvis man kalder main-metoden med argumentet 'testmode', bliver "Test Mode" aktiveret. Herunder er vist et billede af koden til aktivering af 'Test Mode' i MainControlleren.

```
35  public MainController(String[] args) {
36    // Used for testmode only.
37    if (args.length != 0) {
38      if (args[0].toLowerCase().equals("testmode"))
39        testMode = new TestModeController(true);
40    } else {
41      testMode = new TestModeController(false);
42    }
```

Når "Test Mode" bliver aktiveret kan spilleren manipulere spillet ved hjælp af en "Test Mode"-menu der kun kommer frem, når softwaren er i "Test Mode" og først efter spilleren har slået med terningerne. På billedet herunder kan man se hvilke muligheder man får i "Test Mode"-menuen.



Figur 7: Test mode-menuen

Ved hjælp af ”Test Mode”-menuen kan man slippe for at skulle bruge debuggeren til at lave simple tests af spillets logik.

Eksempel: For at købe et felt i debuggeren skal der indsættes et objekt i et array og en variable skal sættes til true - dette er ikke nemt at løse igennem debuggeren. Dette kan gøres simpelt i ”Test Mode”-menuen ved at vælge ”Giv grund” og indtaste feltnummeret på den grund man ønsker.

TestModeControlleren er lavet således at det er nemt, og hurtigt, at tilføje nye kommandoer til ”Test Mode”-menuen. Denne lethed opnås ved at der køres et ”Switch-statement” på kommandoen, som brugeren har valgt. ”Switchen” kalder i de fleste tilfælde blot en privat metode, som udføre den kommando brugeren har valgt. På denne måde skal man blot implementere den private metode i TestModeControlleren, som gør det ønskede, og derefter tilføje muligheden til ”Switchen”.

Eksempel: Blandt andet er der følgende muligheder i ”Test Mode”-menuen:

- ”Giv grund” som lader spilleren overtage et specifikt felt ud fra et udvalgt feltnummer.
- ”Giv farve” som lader spilleren overtage alle felter i en udvalgt farve.

Begge muligheder har deres egen private metode, som bliver kaldt, når disse muligheder bliver valgt. Disse private metoder benytter en privat hjælpe metode kaldet changeFieldOwnership. Denne metode ændrer ejeren af et ejerbart felt. Først checker den om det valgte felt overhovedet kan ejes. Dernæst tjekkes der om feltet er ejet. Hvis det er ejet så fjernes den forrige ejer. Derefter gives spilleren de penge som grunden koster, og dernæst køber han blot grunden på normalvis. Koden til changeFieldOwnership kan ses herunder:

```
135   private void changeFieldOwnership(GameBoard board, Player player, int
          fieldNum) {
136     Field testField = board.getField(fieldNum);
137     //If the field can be owned.
138     if ((testField instanceof entity.field.Ownable)) {
139       Ownable currentField = (Ownable) testField;
140       //if the field is owned.
141       if (currentField.getOwner() != null) {
142         currentField.getOwner().removeField(currentField);
143         currentField.removeOwner();
```

```
144         }
145         //Buy the field, and give the player what i payed for the field.
146         player.changeAccountBalance(currentField.getPrice());
147         player.buyField(currentField);
148         GUI.setOwner(fieldNum, currentField.getOwner().getName());
149 
150       } else {
151         GUI.getUserButtonPressed("Dette felt kan ikke købes", "Ok");
152       }
153     }
```

ChangeFieldOwnership kan dermed bruges til at overtage et hvilket som helst felt. Dette betyder at hvis man ønskede dette kunne man i princippet let udvide ”Test Mode”-menuen med en mulighed ”Giv type” som lader spilleren overtage alle felter i én bestemt type. Eksempelvis rederier, tapperier eller ejendomme. Man kunne lave en privat metode der løb alle felter igennem med den valgte type og samtidig kaldte changeFieldOwnership.

Denne test klasse er blevet brugt til at gennemteste det meste af spillet. Det har minimeret arbejdstiden for de tests der er blevet foretaget. Som konsekvens heraf er der blevet lavet mange flere test af softwaren end hvad der ellers ville være blevet lavet. TestModeControlleren har også gjort det væsentligt mere overskueligt at teste softwaren. Kort sagt, har TestModeControlleren været med til at effektivisere testningen af softwaren.

Eksempel: Et eksempel på en test, der kan blive lavet væsentligt hurtigere ved hjælp af TestModeControlleren er en test, hvor man vil se om en spiller taber ”korrekt”. I ”Test Mode” gøres dette blot ved at give spiller1 nogle grunde, sætte spiller2s pengebeholdning til 0, og dernæst flytte spiller2 til en af spiller1s grunde. Dermed taber spiller2 da han ikke kan betale lejen til spiller1. Desuden kan man også hurtigt tjekke om spiller2s grunde bliver frigivet rigtigt. Hvis spiller2 havde huse/hoteller på hans grunde kunne man også se om disse bliver fjernet korrekt.

9.2 Test-cases

For at teste programmet er der lavet nogle test-case beskrivelser, der dækker de store dele af programmet. Disse test udført ved hjælp af TestModeController-klassen. De fulde test-case beskrivelser kan findes i bilagene i afsnit 12.2, men her er et overblik:

1. Test af spillets start. Oprettes spillerne korrekt.
2. Test af en tur. Slås der med terningerne? Rykker spillerne? Sker der noget når spillerne lander på et felt?
3. Test af ejendomme. Tjek at køb af ejendom og huse fungerer. Tjek at lejen trækkes korrekt.
4. Test af rederier. Stiger lejen når man har flere rederier?
5. Test af tapperier. Bliver lejen sat efter terninge slaget?
6. Test af fængsel. Test alt hvad der har med fængslet at gøre.
7. Test af skatte felterne. Bliver spilleren fratrukket den rigtige skat?

8. Test af 'Prøv lykken'-kort. Gør 'Prøv lykken'-kortene de rigtige ting?
9. Test om man kan tabe og vinde spillet. Taber spillerne korrekt? Bliver den sidste spiller erklæret vinder?
10. Test af max grænse for huse og hoteller.
11. Test af salg af grunde og huse.

Det kan nu ses at hovedtrækkene i programmet er testet. Programmet er testet bredt i de fleste test-cases. Dette garanterer selvfølgelig ikke at programmet er fejlfrit, men det forventes dog at spillet fungerer, som forventet i langt de fleste spil gennemkørsler.

9.3 Junit

Grundet størrelsen af programmet og den begrænset mængde tid, er der valgt kun at teste klasser der ikke afhænger af mange andre klasse. Desuden har man i disse klasser kun valgt at teste de mere komplicerede metoder. Der er ikke fremstillet meget dokumentation for dette, da der i projektet har fokuseret på at få koden til at virke, fremfor at Unit-teste al kode.

Følgende er en liste over der klasser der er blevet testet i.

- JUnit af Shipping.
- JUnit af Brewery.
- JUnit af Player.
- JUnit af Die.
- JUnit af Account.

9.4 System status

Systemet er blevet gennemtestet og der er ikke fundet nogle alvorlige fejl, der ødelægger spillet (eller spiloplevelsen). Der er dog blevet fundet en mindre alvorlig fejl. Fejlen opstår når alle huse er blevet købt og én spiller forsøger at sælge et hotel. Det bør ikke være muligt at sælge et hotel, når der ikke er flere huse tilbage. Fejlen består i, at i programmet bliver hotellet stadig solgt og fire huse bliver sat på grunden. Dette gør at mængden af ledige huse bliver negativ. Dermed har spilleren i princippet snydt, da han har lavet flere huse end der er i spillet.

Dette har ikke nogen stor betydning, da spilleren stadig ikke kan købe huse før mængden af huse igen er blevet positiv. Derfor er det vurderet at spillet stadig kan spilles med en tilfredsstillende oplevelse. Fejlen er relativt kompliceret at rette, og for mange spillere vil fejlen ikke påvirke deres spil. Derfor er fejlen ikke blevet rettet i programmet.

10 Forbedringsforslag

Forslag til features.

I dette projekt blev der lagt vægt på at skabe et fungerende Matador-spil, med operationsdygtige funktioner. Man nåede desværre ikke at implementere alle funktioner i spillet. Et par features der er blevet overvejet, men som ikke er blevet implementeret kan findes herunder.

- Bytning af grunde mellem spillere.
- Pantsætning.
- Auktion af grunde.
- De resterende prøv-lykken kort.
- Mulighed for at gemme spillet.
- Mulighed for at få et overblik over ens grunde.
- Mulighed for at give op.
- Start op menu, så man kan starte nye spil uden at lukke programmet.
- Opsummering af resultater, ejendomme, o. lign. ved spillets afslutning

Bedre GUI

GUI'en er generelt ikke særlig brugervenlig. Man skal fx trykke på 'OK'-knappen for at gå videre. Her kunne det være rart også at have muligheden for at trykke på 'enter' eller 'space' for at gå videre. Desuden findes GUI'en kun i en lav opløsning, hvilket er irriterende på moderne computere hvor opløsning typisk er tre-fire gange større i bredden. Mange af felterne mangler derudover information man ofte godt vil kende, eksempelvis lejen på grunde ved forskellige antal huse. Der er også en fejl hvis man indtaster antal spillere sammentidig med man trykker ok, kan man risikere at man får trykket flere spillere end 6, hvilket skaber fejl i GUI'en, for der er et lille tidsområde hvor man kan trykke ok, før den bliver knappen bliver låst.

Sprog klasse

Det er i dette projekt blevet fravalgt at lave spillet nemt oversættelig. Det kan dog tilføjes i en separat statisk sprog klasse. Denne kunne indeholde alt tekst der udskrives i spillet. På den måde kan en oversætter blot oversætte dette dokument og så er hele spillet oversat. Dette kræver dog nogle ændringer i den måde tekst bliver udskrevet på. Da alle strings i controllerne skal erstattes med et kald til sprog klassen.

Bank klasse

En ting der kunne tilføjes til koden som ville gøre koden mere læselig ville være en 'Bank'-klasse. Denne klasse skulle arve fra 'Player' klassen. På denne måde kunne man kalde player med en payRent(Bank bank) metode. Ved at gøre dette kan man udnytte polymorfi. Derfor skal man ikke længere ændre spillerens balance manuelt, da payrent metoden gør dette. Nu kaldes blot payRent metoden med ejeren af feltet som input. Her ejer banken så alle felter der ikke kan ejes af spillerne. Dette gør det muligt at undgå nogle branches i koden helt eller delvist. Samtidig vil det også gøre det muligt at bruge playerAffordPayment metoden inde i payRent metoden, da payRent nu altid bliver kaldt når spilleren skylder penge.

11 Konklusion

I dette projekt er der blevet udviklet et computerspil baseret på brætspillet matador. De væsentligste funktioner fra brætspillet er blevet implementeret i spillet.

Der er stadig et par ting der kan tilføjes til programmet. Heriblandt er nogle små ting såsom pantsætning og de resterende chancekort. Et par større features der mangler er auktion af grunde og ejendomme, samt handle med grunde mellem spillere.

Spillet kan dog stadig spilles uden disse features, da de ikke er nødvendige for en god Matador oplevelse.

Spillet er blevet grundigt testet primært ved brug af blackbox testing. Mange forskellige testcases er oprettet og alle features er blevet gennemtestet. Derved har er det sikret at kvaliteten af produktet er på et acceptabelt niveau og kun få kendte fejl.

12 Bilag

12.1 KravSpecifikation

1 - Spillere

- F1.1: Der skal være mellem 2 og 6 spillere.
- F1.2: Spillerne skal slå med terningerne.
- F1.3: Spillerne skal rykke deres brik med summen af terningernes øjne.
- F1.4: Når spillerne passerer start, skal spilleren modtage 4000 kr.
- F1.5: Spillerne skal have en pengebeholdning.
  - F.1.5.1: Spillerne skal starte med 30000 kr.
- F1.6: Hvis en spiller slår to ens, skal han have en
- ekstra tur.
  - F.1.6.1: Hvis en spiller slår to ens tre gange i træk, skal spilleren i fængsel.
- F1.7: Spillerne skal tabe spillet, når deres samlede salgsværdi ikke kan betale deres gæld.

2 - Felttyper

- Felt-informationen (grundpriser mm.) kan findes i bilag.
- F2.1: Der skal være 40 felter i alt.
- F2.2: Der skal være 6 hovedtyper af felter.
  - F2.2.1: Der skal være felter der skal kunne ejes.
    - F2.2.1.1: Der skal være fire rederier.
    - F2.2.1.2: Der skal være to tapperier.
    - F2.2.1.3: Der skal være tyve ejendomme
  - F2.2.2: Der skal være skatte felter.
    - F2.2.2.1: Der skal være ét “ekstraordinær Statsskat”-felt.
    - F2.2.2.2: Der skal være ét “Indkomstskat”-felt.
  - F2.2.3: Der skal være fængsels felter.
    - F2.2.3.1: Der skal være ét “Gå i fængsel”-felt.
    - F2.2.3.2: Der skal være ét “På besøg i fængsel”-felt.
  - F2.2.4: Der skal være ét Parkeringsfelt.

- F2.2.5: Der skal være ét Startfelt.
- F2.2.6: Der skal være seks Prøv-lykken felter.

3 - Felter der kan ejes

- F3.1: Et felt der kan ejes skal have en grundpris.
- F3.2: Hvis en spiller lander på et felt benævnt i skal spilleren have mulighed for at købe feltet, hvis feltet ikke er ejet og spilleren har råd.
  - F3.2.1: Hvis spilleren ikke har råd til feltet, skal feltet på auktion.
- F3.3: Hvis en spiller lander på et felt benævnt i F2.2.1 og feltet er ejet, så skal spilleren betale leje til ejeren af feltet.
- F3.4: Lejen skal afhænge af felttypen.
  - F3.4.1: Hvis feltet er af typen rederi, så afhænger lejen af antallet af rederier, som ejeren har.
    - F3.4.1.1: Hvis ejeren har ét rederi, skal spilleren betale 500 kr i leje.
    - F3.4.1.2: Hvis ejeren har to rederier, skal spilleren betale 1000 kr i leje.
    - F3.4.1.3: Hvis ejeren har tre rederier, skal spilleren betale 2000 kr i leje.
    - F3.4.1.4: Hvis ejeren har fire rederier, skal spilleren betale 4000 kr i leje.
  - F3.4.2: Hvis feltet er af typen tapperi, så skal spilleren betale leje til ejeren baseret på spillerens terningesum og antallet af tapperier ejeren har.
    - F3.4.2.1: Hvis ejeren har ét tapperier er lejen 100 gange terninge summen.
    - F3.4.2.2: Hvis ejeren har to tapperier er lejen 200 gange terninge summen.
  - F3.4.3: Hvis feltet er af typen ejendom, så skal spilleren betale leje til ejeren. Lejen bestemmes ud fra hvor mange ejendomme af samme farve ejeren har og ud fra hvor mange bygninger han har på ejendommen. Der er 7 mulige lejer pr. ejendom:
    - F3.4.3.1: Hvis han ikke ejer alle ejendomme med samme farve.
      - F3.4.3.1.1: Lejen skal være basislejen.
  - F3.4.3.2: Hvis han ejer alle ejendomme med samme farve, men ingen bygninger på den ejendom spilleren landede på.
    - F3.4.3.2.1: Lejen skal være 2 x basislejen.
  - F3.4.3.3: Hvis han ejer alle ejendomme med samme farve og ét hus på den ejendom spilleren landede på.
    - F3.4.3.3.1: Lejen skal være den for ét hus.
  - F3.4.3.4: Hvis han ejer alle ejendomme med samme farve og to huse på den ejendom spilleren landede på.

        * F3.4.3.4.1: Lejen skal være den for to huse.
    - F3.4.3.5: Hvis han ejer alle ejendomme med samme farve og tre huse på den ejendom spilleren landede på.
        * F3.4.3.5.1: Lejen skal være den for tre huse.
    - F3.4.3.6: Hvis han ejer alle ejendomme med samme farve og fire huse på den ejendom spilleren landede på.
        * F3.4.3.6.1: Lejen skal være den for fire huse.
    - F3.4.3.7: Hvis han ejer alle ejendomme med samme farve og et hotel på den ejendom spilleren landede på.
        * F3.4.3.7.1: Lejen skal være den for ét hotel.
- F3.5: Et felt der kan ejes skal kunne sælges tilbage til banken for 50% af grundprisen.
    - F3.5.1: Feltet kan kun sælges, når gæld ikke kan betales på andre måder.
    - F3.5.2: Feltet kan altid sælges.

4 - Køb og salg af huse på ejendomme

- F4.1: Der skal være 32 huse og 12 hoteller tilgængeligt i spillet.
    - F4.1.1: Huse og hotel har én pris, som skal bestemmes af ejendommen.
    - F4.1.2: Hvis alle huse og hoteller er købt og en spiller gerne vil købe et hus/hotel, skal spilleren vente til der bliver et hus/hotel tilgængeligt.
    - F4.1.3: Vil flere spillere købe huse/hoteller og der er færre tilbage end det antal spillerne til sammen vil købe, skal der ske en auktion af de sidste huse/hoteller.
    - F4.1.4: En spiller skal kunne købe et hus på én ejendom, hvis han ejer alle ejendomme med den farve, som ejendommen han vil bygge på har.
        * F4.1.4.1: Spilleren skal bygge huse jævnt. Dvs der skal være 1 hus på alle ejendommene i ejendomskomplekset, før der kan komme 2 på en af ejendommene.
        * F4.1.4.2: Når der er 4 huse på alle ejendomme af samme farve og ejeren ønsker at købe endnu en bygning skal husene udskiftes med et hotel og det skal ikke længere være muligt at købe én bygning på denne ejendom.
        * F4.1.3.3: Spilleren skal altid kunne sælge huse/hoteller tilbage til banken for den halve købspris.

5 - Prøv-lykken felter og kort.

'Prøv-lykken'-kort information (beskrivelser mm.) kan findes i bilag.

- F5.1: Hvis spilleren lander på et 'Prøv-lykken'-felt, som benævnt i F2.3.6 skal han trække et 'Prøv-lykke'-kort.

- F5.2: Der skal være 6 hovedtyper af 'Prøv-lykken' kort.
  - F5.2.1: Der skal være to kort af typen: Kongens fødselsdag.
    * F5.2.1.1: Dette kort skal kunne beholdes af spilleren.
    * F5.2.1.2: Kortet skal kunne bruges til at komme ud af fængslet i starten af spillerens tur. Spilleren skal herefter kunne fortsætte sin tur.
    * F5.2.1.3: Kortet skal kunne sælges til andre spillere.
  - F5.2.2: Der skal være et kort af typen: Matador legat.
    * F5.2.2.1: Dette kort skal give spilleren 40000, hvis hans samlede formue er mindre end 15000.
  - F5.2.3: Der skal være to kort af typen: Skat.
    * F5.2.3.1: Her skal spilleren betale skat til banken.
      · F5.2.3.1.1: Størrelsen af skatten er baseret på mængden af huse og hoteller spilleren ejer.
  - F5.2.4: Der skal være 16 kort af typen 'Bevægelse'.
    * F5.2.4.1: Der er flere typer af 'Bevægelses'-kort.
      · F5.2.4.1.1: 'Ryk til felt'

        - F5.2.4.1.1.1: Her skal spilleren rykke til feltet skrevet på 'Prøv lykken'-kortet.

        - F5.2.4.1.1.2: Hvis spilleren passerer start skal spilleren indkasserer 4000 kr.-

      · F5.2.4.1.2: 'Ryk tre felter'

        - F5.2.4.1.2.1: Her skal spilleren rykke tre felter frem eller tilbage, specificeret af 'Prøv-lykke'-kortet.

        - F5.2.4.1.2.2: Hvis spilleren passerer start skal spilleren indkasserer 4000 kr.-

      · F5.2.4.1.3: 'Ryk i fængsel'

        - F5.2.4.1.3.1: Her skal spilleren rykke i fængsel.

        - F5.2.4.1.3.2: Hvis spilleren passerer start skal spilleren ikke indkassere 4000 kr.-

      · F5.2.4.1.4: 'Ryk frem til nærmeste rederi'

        - F5.2.4.1.4.1: Her skal spilleren rykke frem til nærmeste rederi på spillepladen.

- F5.2.4.1.4.2: Hvis specificeret af 'Prøv-lykken'-kortet skal spilleren betale ejeren af rederiet den dobbelte leje.

* F5.2.5: Der skal være 3 kort af typen: Fest. Her skal spilleren der trak Prøv-lykken-kortet modtage et beløb for eventet fra alle andre spillere.

* F5.2.6: Der skal være 21 kort af typen 'Betaling'. Her skal spilleren der trak 'Prøv-lykken'-kortet enten modtage eller betale et beløb fra/til banken.

6 - Skatte felter

- F6.1: Hvis en spiller lander på et felt benævnt i F2.2.2 skal spilleren betale skat.
  - F6.1.1: Hvis feltet er et "ekstraordinær Statsskat"-felt, skal skatten være 2000 kr.-
  - F6.1.2: Hvis feltet er ét "Indkomstskat"-felt, skal skatten vær e 10 % af den samlede spillerens samlede formue eller 4000 kr.-

7 - Fængselsfelter

- F7.1: Hvis en spiller lander på et 'Gå i fængsel'-felt skal spillerens tilstand skifte til 'I fængsel'- tilstand og spilleren skal rykkes til feltet 'På besøg i fængsel'-feltet.
  - F7.1.1: Selv om spilleren passerer 'Start' skal spilleren ikke indkassere 4000 kr.-
- F7.2: Hvis en spiller lander på feltet 'På besøg i fængsel' skal der ingenting ske. Feltet skal være neutralt.
- F7.3: Hvis en spiller trækker et 'Prøv-lykken'-kort af typen 'Ryk i fængsel' skal spillerens tilstand skifte til 'I fængsel'-tilstand og spilleren skal rykkes til feltet 'På besøg i fængsel'-feltet.
- F7.4: Når spillerens tilstand er 'I fængsel' skal spilleren
  - F7.4.1: Ikke kunne opkræve leje fra andre spillere.
  - F7.4.2: Kunne deltage i auktioner.
  - F7.4.3: Kunne deltage i handler mellem spillerne.
  - F7.4.4: Ikke kunne flytte sin brik før han er blevet løsladt.
- F7.5: Man skal løslades fra fængslet og skifte til spillerens tilstand til 'Normal'-tilstand, hvis
  - F7.5.1: Spilleren slår to ens.
    - F7.5.1.1: Spilleren skal rykker frem efter øjensummen, som normalt.
    - F7.5.1.2: Spilleren skal have ekstraslag ved to ens som normalt.
    - F7.5.1.3: De to ens skal tælle med i antallet af to-ens i træk, som normalt (benævnt i F1.6.1).
  - F7.5.2: Spilleren betaler 1000 kr.- i bøde i starten af spillerens tur.

    - F7.5.2.1: Spilleren skal da have sin tur som normalt.
  - F7.5.3: Spilleren har og bruger et 'Prøv-lykken'-kort af typen 'Kongens fødselsdag' i starten af sin tur.
    - F7.5.3.1: Spilleren skal da have sin tur som normalt.
  - F7.5.4: Spilleren har været i fængsel i 3 runder.
    - F7.5.4.1: Spilleren skal da tvinges til at betale 1000 kr.- i bøde i starten af sin 4. tur i fængslet
    - F7.5.4.2: Spilleren skal da have sin tur som normalt.

8 - Start felt

- F8.1: Spilleren skal starte spillet på 'Start'-feltet.
- F8.2: Hvis en spilleren lander på eller passerer 'Start'-feltet benævnt i F2.2.5 skal spilleren modtage 4000 kr.-

9 - Parkering felt

- F9.1: Hvis en spiller lander på 'Parkering'-feltet benævnt i F2.2.4 skal der ingenting ske. Feltet skal være neutralt.

10 - Tab af spillet

- F10.1: En spiller skal udgå fra spillet (tabe), hvis han skylder mere end han ejer.
  - F10.1.1: Spilleren skal overdrage alt til sin kreditor efter at have solgt eventuelle bygninger (huse og hoteller) tilbage til banken.
    - F10.1.1.1: Hvis kreditoren er en spiller modtager kreditoren alle spillerens grunde og resterende penge.
    - F10.1.1.2: Hvis kreditoren er banken så frigives alle spillerens grunde til banken.
      - F10.1.1.2.1: Bankøren sætter straks alle grundene på auktion.
    - F10.1.1.3: F10.1.1.1 og F10.1.1.2 gælder også ved træk af 'Prøv-lykken'-kort som ikke kan betales.

11 - Pantsætning

- F11.1: En spiller skal kunne pantsætte sine grunde.
  - F11.1.1: En spiller kan kun pantsætte sine ubebyggede grunde etc. til banken for det halve af grundens værdi.

- F11.1.2: Har spilleren bygninger på grunden skal bygningerne først sælges tilbage til banken.
- F11.1.3: Spilleren skal beholde grunden.
  - F11.1.3.1: Spilleren skal ikke længere kunne opkræve leje for grunden.
- F11.2: En spiller skal kunne købe sine pantsatte grunde tilbage.
  - F11.2.1: Prisen skal være pantsætnings værdien + 10% af pantsætnings værdien i rente.
- F11.3: Hvis en pantsat grund sælges til en anden spiller skal
  - F11.3.1: Køberen kun betale pantsætnings værdien, hvis køberen ophæver pantsætningen med det samme.
  - F11.3.2: Køberen betale pantsætnings værdien + 10% af pantsætnings værdien i rente, hvis køberen ikke ophæver pantsætningen med det samme.

12 - Handel og lån mellem spillere

- F12.1: Spillere skal ikke kunne låne penge af hinanden.
- F12.2: Spillere skal ikke kunne sælge/købe bygninger af hinanden.
- F12.3: Spillere skal kunne handle indbyrdes med ubebyggede grunde.

13 - Auktion

- F13.1: Ved en auktion skal alle spiller have lov til at byde på skift.
- F13.2: Auktionen skal være åben dvs. alle spillere ved hvad de andre spillere har budt indtil videre.
- F13.3: Auktionen skal slutte, når der ikke er flere spillere der vil byde.
- F13.4: En spillere skal ikke kunne byde mere end han har af kontanter.
- F13.5: Man skal ikke kunne handle med andre spillere under en auktion.

12.2 TestCases

Dette afsnit viser de testcases som specifikt er testet ved brug af testmodekontrolleren. Det kan dermed ses som blackboxtesting af systemet. Der er en en testcase på hver side.

Testcase 1 - Start spillet

Udførelse:

- Start spillet.
- Indtast antallet af spillere.
- Indtast navne på spillerne.
- Tjek at alle spillerne dukker op på startfeltet og i balance menuen.
- Check om det virker med flere spillere.

Resultat: Succes.
Arvid 10-01-2017

Testcase 2 - Spil en tur.

PreCondtion:
Spillet er startet.
Udførelse:

- Slå med terningerne og check at spillerne rykker rundt på pladen.
- Når en spiller lander på et felt, bekræft at der sker hvad det felt bør gøre.
- Check til slut om der kommer en menu frem hvor man kan slutte turen og gøre andre ting.

Resultat: Succes.
Af Mads Stege 10-01-2017

Kommentarer:

- Køb af grund(e) virker som forventet.
- Ejede grunde opkræver den korrekte leje og giver denne til den korrekte ejer af feltet.
- Bryggerier og rederier fratrækker den korrekte leje af spilleren, og giver denne korrekt til ejeren.
- "Prøv lykken", "De fængsles", "START" og "Skat" felterne virker også som de skal.
- Spillerne kan til sidst vælge enten at slutte turen, og sende den videre til den næste spiller. Spillerne kan også vælge at købe huse på deres grunde. Hvis spilleren ikke ejer det nødvendige antal grunde for at bygge, sendes han tilbage til den tidligere menu med en besked om at han ikke kan købe huse endnu.

Testcase 3 - Ejendomme

PreCondtion:
Spillet er startet.
Udførelse:

- Køb en ejendom for spiller 1.
- Lad en anden spiller lande på feltet, og tjek han betaler en leje.
- Køb 2 andre ejendomme i samme farve for spiller 1.
- Lad en anden spiller lande på samme felt og tjek han betaler dobbelt leje fra før.
- Køb et hus på alle tre grunde for spiller 1.
- Lad en anden spiller lande på et af felterne og tjek at lejen stiger.
- Gør dette for alle mængder af huse inklusiv hotel.

Resultat: Succes.
Af Mads Stege 10-01-2017

Kommentarer:

- Spiller 1 køber korrekt en tilfældig ejendom. Denne tilføres en farvet ramme og spiller 1 fratrækkes et beløb svarende til ejendommens pris.
- Spiller 2 sættes til at lande på feltet vha. Programmets testMode-funktion, og vi ser at spiller 2 korrekt fratrækkes leje, som overføres til spiller 1's pengebeholdning.
- Spiller 1 fratrækkes endnu en grunds pris fra vedkommendes balance.

Testcase 4 - Shipping

PreCondtion:
Spillet er startet.
Udførelse:

- Spiller 1 køber et shipping felt.
- En anden spiller lander på det og får fra trukket 500
- Køb en ekstra for spiller og tjek at beløbet bliver fordoblet.
- Gør dette indtil alle shipping felter er købt.

Resultat: Succes
Af Mads Stege 10-01-2017

Testcase 5 - Tapperi.

PreCondtion:
Spillet er startet.
Udførelse:

- Spiller 1 køber et Brewery felt.
- En anden spiller lander på det og får trukket 100 gange sit sidste terningeslag.
- Spiller 1 køber det andet brewery felt.
- En anden spiller lander på det og får trukket 200 gange sit sidste terningeslag.

Resultat: Succes
Af Mads Stege 10-01-2017

Testcase 6 - Prison

PreCondtion:
Spillet er startet.
Udførelse:

- Check at spilleren går i fængsel når han lander på gå i fængsel.
- Check at spilleren går i fængsel når han slår 3 ens tre ture i træk.
- Check at spillere kan betale sig ud.
- Check at spilleren kan slå to ens for at komme ud af fængslet.
- Check at spilleren bliver befriet fra fængslet 4 runde han er der inde.

Resultat: Succes
Testet af Arvid 11-01-17

Testcase 7 - Tax

PreCondtion:
Spillet er startet.
Udførelse:

- Check at spilleren betaler 2.000 kr. når der landes på felt nummer 39.
- Check at spilleren kan vælge at betale 4.000 kr. eller 10% på felt nummer 5.
- Check at begge ting virker.

Resultat: Succes
Testet af Arvid 11-01-17

Testcase 8 - Chance Kort

PreCondtion:
Spillet er startet.
Udførelse:

- Check at ryk til et felt kortet virker korrekt.
- Check at ryk 3 felter virker korrekt.
- Check at ryk til nærmeste rederi virker korrekt.
- Check at gå i fængsel kortet virker.
- Check at betal/modtag et beløb kort virker.

Resultat: Succes
Testet af Arvid og Simon - 11-01-2017

Testcase 9 - WinLose

PreCondtion:
Spillet er startet.
Udførelse:

- Check at spillere mister alle deres ting når de taber, og at huse bliver fjernet fra grundenene, check helst for spil med 3 spillere, og tjek at spillerne kan købe grundende igen, efter de er sat fri.
- Check at en spiller vinder når alle andre spillere har tabt.

Resultat: Success.
Testet af Arvid 11-01-2017

Testcase 10 - Edge cases.

Udførelse:

- Test at der ikke kan købes flere huse end der er i spillet.
- Test at der ikke kan købes flere hoteller end der er i spillet.

Resultat: Succes
Testet af Arvid og Simon 11-01-2017

Testcase 11 - Salg.

PreCondtion:
Spillet er startet.
Udførelse:

- Åbner den salgs menuen når man ikke har nok penge, men ens fortune er stor nok til at betale gælden.
- I ”Sælg grund”må der kun dukke grunde som kan sælges.
- I ”Sælg hus”må der kun dukke farver op hvor der alle af samme farve er eget.
- I ”Farve menuen”må den kun vises grunde man kan sælge huse på.
- Check at man kan kan sælge huse.
- Check at man kan sælge grund.
- Check at menuerne stadig virker når man har solgt ALT!
- Check at når man har råd til at betale lejen, skal man have mulighed for dette. Lejen skal også overføres korrekt til den rette ejer.

Resultat: Succes
Testet af Arvid og Jeppe 12-01-2017

12.3 Feltliste

Nr:	Felt navn:	Feltfarve:	Feltværdi:	Grundleje:	1hus:	2huse:	3huse:	4huse:	Hotel:	Pant:	Felttype:	Huspris:
1	**Start**	*Ingen*		Nej	X	X	X	X	X	X	Start	
2	**Rødovrevej**	**Blå**	1200	50	250	750	2250	4000	6000	600	Ejendom	1000
3	**Prøv lykken**	*Ingen*		Nej	X	X	X	X	X	X	Chancekort	
4	**Hvidovrevej**	**Blå**	1200	50	250	750	2250	4000	6000	600	Ejendom	1000
5	**Skat (10% eller 4.000)**	*Ingen*		Nej	X	X	X	X	X	X	Skat	
6	**Scandlines**	*Ingen*	4000	500/1000/2000/4000	X	X	X	X	X	2000	Rederi	
7	**Roskildevej**	**Orange**	2000	100	600	1800	5400	8000	11000	1000	Ejendom	1000
8	**Prøv lykken**	*Ingen*		Nej	X	X	X	X	X	X	Chancekort	
9	**Valby Langgade**	**Orange**	2000	100	600	1800	5400	8000	11000	1000	Ejendom	1000
10	**Allégade**	**Orange**	2400	150	800	2000	6000	9000	12000	1200	Ejendom	1000
11	**Fængsel - På besøg**	*Ingen*		Nej	X	X	X	X	X	X	Safe Haven	
12	**Frederiksberg Allé**	**Grøn**	2800	200	1000	3000	9000	12500	15000	1400	Ejendom	2000
13	**Tuborg Squash**	*Ingen*	3000	100*X eller 200*X	X	X	X	X	X	1500	Tapperi	
14	**Bülowsvej**	**Grøn**	2800	200	1000	3000	9000	12500	15000	1400	Ejendom	2000
15	**Gl. Kongevej**	**Grøn**	3200	250	1250	3750	10000	14000	18000	1600	Ejendom	2000
16	**Mols-Linien**	*Ingen*	4000	500/1000/2000/4000	X	X	X	X	X	2000	Rederi	
17	**Bernstorffsvej**	**Grå**	3600	300	1400	4000	11000	15000	19000	1800	Ejendom	2000
18	**Prøv lykken**	*Ingen*		Nej	X	X	X	X	X	X	Chancekort	
19	**Hellerupvej**	**Grå**	3600	300	1400	4000	11000	15000	19000	1800	Ejendom	2000
20	**Strandvejen**	**Grå**	4000	350	1600	4400	12000	16000	20000	2000	Ejendom	2000
21	**Parkering**	*Ingen*		Nej	X	X	X	X	X	X	Safe Haven	
22	**Trianglen**	**Rød**	4400	350	1800	5000	14000	17500	21000	2200	Ejendom	3000
23	**Prøv lykken**	*Ingen*		Nej	X	X	X	X	X	X	Chancekort	
24	**Østerbrogade**	**Rød**	4400	350	1800	5000	14000	17500	21000	2200	Ejendom	3000
25	**Grønningen**	**Rød**	4800	400	2000	6000	15000	18500	22000	2400	Ejendom	3000
26	**Scandlines**	*Ingen*	4000	500/1000/2000/4000	X	X	X	X	X	2000	Rederi	
27	**Bredgade**	**Hvid**	5200	450	2200	6600	16000	19500	23000	2600	Ejendom	3000
28	**Kgs. Nytorv**	**Hvid**	5200	450	2200	6600	16000	19500	23000	2600	Ejendom	3000
29	**Coca Cola**	*Ingen*	3000	100*X eller 200*X	X	X	X	X	X	1500	Tapperi	
30	**Østergade**	**Hvid**	5600	500	2400	7200	17000	20500	24000	2800	Ejendom	3000
31	**De fængsles**	*Ingen*		Nej	X	X	X	X	X	X	Fængsel	
32	**Amagertorv**	**Gul**	6000	550	2600	7800	18000	22000	25000	3000	Ejendom	4000
33	**Vimmelskaftet**	**Gul**	6000	550	2600	7800	18000	22000	25000	3000	Ejendom	4000
34	**Prøv lykken**	Ingen		Nej	X	X	X	X	X	X	Chancekort	
35	**Nygade**	**Gul**	6400	600	3000	9000	20000	24000	28000	3200	Ejendom	4000
36	**Scandlines**	Ingen	4000	500/1000/2000/4000	X	X	X	X	X	2000	Rederi	
37	**Prøv lykken**	Ingen		Nej	X	X	X	X	X	X	Chancekort	
38	**Frederiksberg Gade**	**Lilla**	7000	700	3500	10000	22000	26000	30000	3500	Ejendom	4000
39	**Skat (2.000)**	*Ingen*		Nej	X	X	X	X	X	X	Skat	
40	**Rådhuspladsen**	**Lilla**	8000	1000	4000	12000	28000	34000	40000	4000	Ejendom	4000

12.4 Løkke-kort

Løkke-kort listen kan findes i zip filen i Instalation/resources/Cards.csv

12.5 Kildekode.

12.5.1 Controller

12.5.1.1 MainController

```
1  package controller;
2
3  import desktop_resources.GUI;
4  import entity.DiceCup;
5  import entity.Player;
6  import testModeController.TestModeController;
7
8  /**
9   * This class is the main controller. It is responsible of handing out
10  * assigments to the different controllers and moving the player.
11  *
12  * @author Gruppe33
13  *
14  */
15 public class MainController {
16
17    // Instance variables.
18    private Player[] players;
19    private DiceCup dice;
20    private PrisonController prisonController;
21    private FieldController fieldController;
22    private BuildingController propertyController;
23    private DebtController bankController;
24    private TestModeController testMode;
25
26    private int turn;
27    private int numExtraTurn;
28    private boolean extraTurn;
29    private boolean testExtraTurn = false;
30
31    /**
32     * Constructor: Creates the needed variables for the main controller and
33     * construct new objects of some of the other classes.
34     */
35    public MainController(String[] args) {
36      // Used for testmode only.
37      if (args.length != 0) {
38        if (args[0].toLowerCase().equals("testmode"))
39          testMode = new TestModeController(true);
40      } else {
41        testMode = new TestModeController(false);
42      }
43
44      // Creates an objet that iniliazie the GUI gameboard.
45      GUICreator createGUI = new GUICreator();
46      // Initialise the class that reads fields from a text file.
47
48      // Ask the players their names on the GUI
```

```
49      String[] playerNames = createGUI.getPlayerNames();
50
51      // Create an array of players, and add their names.
52      this.players = new Player[playerNames.length];
53      for (int i = 0; i < players.length; i++) {
54        this.players[i] = new Player(playerNames[i]);
55      }
56
57      // Set a random player to start.
58      turn = (int) (Math.random() * players.length);
59
60      numExtraTurn = 0;
61
62      dice = new DiceCup();
63      propertyController = new BuildingController();
64      bankController = new DebtController(propertyController);
65      prisonController = new PrisonController(this, bankController);
66      fieldController = new FieldController(prisonController, this,
            propertyController, bankController);
67    }
68
69    /**
70     * Method main: This methods starts the program.
71     */
72    public static void main(String[] args) {
73      // Creates the main controller.
74      MainController game = new MainController(args);
75      // Starts the game
76      game.playGame();
77    }
78
79    /**
80     * Method changeTurn: Change the turn so that it is the next players turn
            .
81     * Ignores players that have lost.
82     */
83    public void changeTurn() {
84      // Chance the players turn until someone has lost.
85      do {
86        turn = (turn + 1) % players.length;
87      } while (players[turn].getHasLost());
88    }
89
90    /**
91     * Method playTurn: Plays one dice roll of a player, and do everything
92     * appropiate depending on where he lands.
93     */
94    public void playTurn() {
95      // Check if the player is in prison. If he is, hand over control of the
96      // turn to the prison.
97      if (prisonController.checkInPrison(players[turn])) {
98
99        if (!prisonController.inPrison(players[turn]))
100         return;
101     }
```

```
102     // Tell the player it is his turn on the GUI. Say something different
103     // depending on his last roll.
104     if (numExtraTurn < 1)
105       GUI.getUserButtonPressed(players[turn].getName() + " det er din tur."
              , "Slå med terninger");
106     else if (numExtraTurn == 1) {
107       GUI.getUserButtonPressed(players[turn].getName() + " det er din tur
              igen, fordi du har slået to ens ",
108           "Slå med terninger");
109     } else {
110       GUI.getUserButtonPressed(
111           players[turn].getName()
112               + " det er din tur igen, fordi du har slået to ens. Næste
                      gang bliver du sendt i fængsel.",
113           "Slå med terninger");
114     }
115
116     // Roll the dice.
117     int diceSum = rollDice();
118     // Check if he rolled the same on both dice.
119     checkForExtraTurn();
120     // If this is the third time he rolled the double something. Send him
          to
121     // prison.
122     if (numExtraTurn == 3) {
123       prisonController.sendToPrison(players[turn]);
124       extraTurn = false;
125       return;
126     }
127     // Move the player depending on the dice roll
128     movePlayer(diceSum);
129
130     // add landOnField methods.
131     fieldController.landOnField(players[turn], diceSum);
132
133     // Ask the player what he wants to do now.
134     playerTurnDecision();
135   }
136
137   /**
138    * Method playGame: Plays the game until someone has won.
139    */
140   public void playGame() {
141     GUI.getUserButtonPressed("En tilfældig spiller er valgt til at starte",
            "Start spil");
142     // Keep changing turn until someone has won.
143     while (true) {
144       changeTurn();
145       // If a winner is found delcare him winner and close the game.
146       if (checkForWinner() != null) {
147         GUI.getUserButtonPressed("Tillykke " + checkForWinner().getName() +
                " har vundet.", "Sweet");
148         GUI.close();
149         break;
150       }
```

```
151
152         do {
153           playTurn();
154         } while (extraTurn && !players[turn].getHasLost());
155
156       }
157     }
158
159     /**
160      * Method checkforExtraTurn: Checks if the player rolled the same on both
161      * dice. Returns true if that was the case.
162      *
163      * @return boolean
164      */
165     public boolean checkForExtraTurn() {
166       // Check if the dice have equals value, if it does give it an extra
167       // turn.
168       // testExtraTurn is used by the testmode controller to force an extra
169       // turn.
170       if (dice.getDiceValue()[0] == dice.getDiceValue()[1] || testExtraTurn)
              {
171         extraTurn = true;
172         numExtraTurn++;
173       }
174       // If not end the turn.
175       else {
176         extraTurn = false;
177         numExtraTurn = 0;
178       }
179
180       // Testing stuff:
181       testExtraTurn = false;
182       return extraTurn;
183     }
184
185     /**
186      * Method givePlayer4000: Gives the player 4000. Should be used when they
187      * pass the start field.
188      */
189     public void givePlayer4000() {
190       players[turn].changeAccountBalance(4000);
191       GUI.setBalance(players[turn].getName(), players[turn].getAccountBalance
              ());
192       GUI.getUserButtonPressed("Du passerede start og modtager 4.000.", "Ok")
              ;
193     }
194
195     /**
196      * Method movePlayerOnGui: Moves the players car to a position.
197      *
198      * @param playerName
199      *            Name of the player to be moved.
200      * @param position
201      *            New position of the player.
202      */
```

```
203   public void movePlayerOnGUI() {
204     // Remove all the cars of the player
205     GUI.removeAllCars(players[turn].getName());
206     // Place a new car on the new position.
207     GUI.setCar(players[turn].getPosition(), players[turn].getName());
208   }
209
210   /**
211    * Method rollDice: Roll the dice and return their sum.
212    *
213    * @return Sum of the dice.
214    */
215   public int rollDice() {
216     // Roll the dice.
217     dice.shake();
218     // Set the dice on the GUI
219     GUI.setDice(dice.getDiceValue()[0], 2, (int) (2 * (Math.random() - 0.5)
          + 7), dice.getDiceValue()[1], 3,
220         (int) (2 * (Math.random() - 0.5) + 7));
221
222     // If testmode is active open the testmode menu.
223     if (testMode.isActive()) {
224       int newRoll = testMode.options(players[turn], this, fieldController);
225       // Change the diceroll if the player chose to.
226       if (newRoll >= 0)
227         return newRoll;
228     }
229
230     return dice.getDiceValue()[0] + dice.getDiceValue()[1];
231
232   }
233
234   /**
235    * Method removePlayerOnGUI: Remove all cars of a player.
236    *
237    * @param playerName
238    *            Name of the player.
239    */
240   public void removePlayerOnGUI(String playerName) {
241     // Remove all the cars of the player
242     GUI.removeAllCars(playerName);
243   }
244
245   /**
246    * Method movePlayer: Moves the player forward on the board.
247    *
248    * @param diceSum
249    *            The sum of the dice.
250    */
251   public void movePlayer(int diceSum) {
252
253     // Check if the player can move to the next field problem free.
254     if (players[turn].getPosition() + diceSum <= 40 && players[turn].
          getPosition() + diceSum > 0) {
255       players[turn].setPosition(players[turn].getPosition() + diceSum);
```

```
256       }
257       // If the dicesum is negative, move backwards.
258       else if (players[turn].getPosition() + diceSum < 1) {
259         players[turn].setPosition(40 + diceSum + players[turn].getPosition())
              ;
260       }
261       // If the player passes 40. Calculate his new position based on his
262       // roll.
263       else {
264         int difference = 40 - players[turn].getPosition();
265         players[turn].setPosition(diceSum - difference);
266         movePlayerOnGUI();
267         givePlayer4000();
268       }
269       // Moves the player to his new position on the GUI.
270       movePlayerOnGUI();
271     }
272
273     /**
274      * Method movePlayerTo: Move the player to a specific location.
275      *
276      * @param newPos
277      *            The new position of the player.
278      */
279     public void movePlayerTo(int newPos) {
280       int currentPos = players[turn].getPosition();
281       players[turn].setPosition(newPos);
282       if (newPos < currentPos) {
283         givePlayer4000();
284       }
285       movePlayerOnGUI();
286
287     }
288
289     /**
290      * Method playerTurnDecision: Gives the player some options of what he
           wants
291      * to do.
292      */
293     public void playerTurnDecision() {
294       // Boolean that holds the decision of if the player want to end his
295       // turn.
296       boolean endTurn = false;
297       if (players[turn].getInPrison() || players[turn].getHasLost()) {
298         return;
299       }
300
301       String output = "Hvad vil du foretage dig?";
302       final String END_YOUR_TURN = "Slut din tur";
303       final String HOUSES_AND_HOTELS = "Køb huse";
304
305       // Keep asking for what the player wants to do until he chooses to add
306       // the player.
307       String userSelection;
308       while (!endTurn) {
```

```
309         userSelection = GUI.getUserSelection(output, END_YOUR_TURN,
                HOUSES_AND_HOTELS);
310         switch (userSelection) {
311         case END_YOUR_TURN:
312           endTurn = true;
313           break;
314         case HOUSES_AND_HOTELS:
315           propertyController.buyBuildingMenu(players[turn]);
316           break;
317
318         }
319       }
320
321     }
322
323     /**
324      * Method TESTsetExtraTurn: This method is only for testing. But it makes
325      * sure the player gets an extra turn.
326      *
327      * @param input
328      *            Whether or not to give an extra turn.
329      */
330     public void TESTsetExtraTurn(boolean input) {
331       testExtraTurn = input;
332     }
333
334     /**
335      * Method getPlayer(): Returns the array.
336      *
337      * @return Return a Player array.
338      */
339     public Player[] getPlayers() {
340       return players;
341     }
342
343     /**
344      * Static Method addReturnArray: Add the String "Gå tilbage" at the end
            of a
345      * String[]. Can be used for menus.
346      *
347      * @param input
348      *            The Array input.
349      * @return The array with "Gå tilbage" at the end.
350      */
351     public static String[] addReturnToArray(String[] input) {
352       String[] output;
353       if (input == null) {
354         output = new String[1];
355       } else {
356         output = new String[input.length + 1];
357         for (int i = 0; i < input.length; i++) {
358           output[i] = input[i];
359         }
360       }
361
```

```
362       output[output.length - 1] = "Gå tilbage";
363       return output;
364     }
365
366     /**
367      * Method that checks if all players except one has lost the game.
368      *
369      * @return Player winner.
370      */
371     public Player checkForWinner() {
372       Player winningPlayer = null;
373       for (int i = 0; i < players.length; i++) {
374         if (!players[i].getHasLost())
375           if (winningPlayer == null)
376             winningPlayer = players[i];
377           else
378             return null;
379
380       }
381       return winningPlayer;
382     }
383
384     /**
385      * Method FieldController: Return the field controller from the Main.
386      *
387      * @return FieldController.
388      */
389     public FieldController getLandOnFieldController() {
390       return fieldController;
391     }
392 }
```

12.5.1.2 BuildingController

```java
package controller;

import desktop_resources.GUI;
import entity.Player;
import controller.MainController;
import entity.field.*;

/**
 * This class is responsible for buynig and selling houses on Streets.
 *
 * @author Gruppe33
 *
 */
public class BuildingController {

  // Instance variables
  private int houses;
  private int hotels;

  /**
   * Constructor: Constructs a PropertyController.
   */
  public BuildingController() {
    this.houses = 32;
    this.hotels = 12;
  }

  /**
   * Method getHouses: Returns the number of houses available.
   *
   * @return Amount of houses available.
   */
  public int getHouses() {
    return houses;
  }

  /**
   * Method getHotels: Returns the number of hotels available.
   *
   * @return Amount of hotels available.
   */
  public int getHotels() {
    return hotels;
  }

  /**
   * Method changeHouses: Changes the amount of houses available.
   *
   * @param amount
   *            The amount to change the houses available with.
   */
  public void changeHouses(int amount) {
    houses = houses + amount;
```

```
54    }
55
56    /**
57     * Method changeHotels: Changes the amount of hotels available.
58     *
59     * @param amount
60     *            The amount to change the hotels available with.
61     */
62    public void changeHotels(int amount) {
63      hotels = hotels + amount;
64    }
65
66    /**
67     * Method countStreetColours: Returns the amount of each coloured streets
68     * the player owns as a integer array.
69     *
70     * @param player
71     *            The player who owns the fields.
72     * @return int array containing the information. fx int[0] = amount of
           blue
73     *         coloured streets owned.
74     */
75    public int[] countStreetColours(Player player) {
76      int[] countedStreetColours = new int[8];
77      String[] streetColours = { "Blå", "Orange", "Grøn", "Grå", "Rød", "Hvid
          ", "Gul", "Lilla" };
78
79      for (int i = 0; i < countedStreetColours.length; i++) {
80        countedStreetColours[i] = player.getStreetsOwned(streetColours[i]);
81      }
82      return countedStreetColours;
83    }
84
85    /**
86     * Method canBuildOnColour: Returns which street colours you can build at
87     * represented as a boolean array.
88     *
89     * @param player
90     *            The player who owns the streets.
91     * @return The boolean array which contains which streets you can buy
92     *         buildings at.
93     */
94    private boolean[] canBuildOnColour(Player player) {
95      // required amount of streets with the same colours to build a house.
96      int[] requiredStreets = { 2, 3, 3, 3, 3, 3, 3, 2 };
97      boolean[] canBuildOnColour = new boolean[8];
98
99      for (int i = 0; i < canBuildOnColour.length; i++) {
100       canBuildOnColour[i] = countStreetColours(player)[i] ==
            requiredStreets[i];
101     }
102     return canBuildOnColour;
103   }
104
105   /**
```

```
106     * Method canBuildOnColourString: Returns which street colours you can
            buy
107     * buildings at.
108     *
109     * @param player
110     *            The player that wants to buy a house.
111     * @return The String array which contains which colours you can buy
112     *         buildings at.
113     */
114    private String[] canBuildOnColourString(Player player) {
115      String[] colours = { "Blå", "Orange", "Grøn", "Grå", "Rød", "Hvid", "
           Gul", "Lilla" };
116      int trueCount = 0;
117      // Counts how many different colours you can build buildings at.
118      for (int i = 0; i < canBuildOnColour(player).length; i++) {
119        if (canBuildOnColour(player)[i]) {
120          trueCount++;
121        }
122      }
123      // New String array containing only the colours that you can build
124      // buildings at.
125      int j = 0;
126      String[] canBuildOn = new String[trueCount];
127      for (int i = 0; i < canBuildOnColour(player).length; i++) {
128
129        if (canBuildOnColour(player)[i] == true) {
130          canBuildOn[j] = colours[i];
131          j++;
132        }
133      }
134      return canBuildOn;
135    }
136
137    /**
138     * Method numbOfStreetsFromColour: Returns how many streets that is
            required
139     * to be able to build a building depending on the colour.
140     *
141     * @param colour
142     *            The colour of the street.
143     * @return 2 if the street is blue 3 otherwise.
144     */
145    private int streetsWithColour(String colour) {
146      int streetAmountWithColour = 0;
147      if (colour.equals("Blå") || colour.equals("Lilla")) {
148        streetAmountWithColour = 2;
149      } else {
150        streetAmountWithColour = 3;
151      }
152      return streetAmountWithColour;
153    }
154
155    /**
156     * Method minimum: Returns the minimum of an array.
157     *
```

```
158     * @param numbs
159     *            The array to find the minimum from.
160     * @return The smallest element of the array.
161     */
162    private int minimum(int[] numbs) {
163      // Set an abretary high value
164      int min = 420;
165
166      for (int i = 0; i < numbs.length; i++) {
167        min = Math.min(min, numbs[i]);
168      }
169      return min;
170    }
171
172    private int maximum(int[] numbers) {
173      // Set an abretary high value
174      int max = 0;
175
176      for (int i = 0; i < numbers.length; i++) {
177        max = Math.max(max, numbers[i]);
178      }
179      return max;
180    }
181
182    /**
183     * Gets the streets of a given colour with most or fewest houses, owned
          by a player.
184     *
185     * @param player
186     *            The player that owns the streets.
187     * @param colour
188     *            The colour of the street.
189     * @param mostHouses
190     *            With the most houses or fewest
191     * @return Returns a String array with street names of a given colour
          with most or fewest houses, owned by a player.
192     */
193    private String[] streetsWithBuildings(Player player, String colour,
          boolean mostHouses) {
194      // An array to hold the amount of houses on the different streets.
195      int[] houses = new int[streetsWithColour(colour)];
196      // An array to hold the names of the different streets.
197      String[] streetNames = new String[streetsWithColour(colour)];
198
199      // Fills the arrays with informations.
200      int j = 0;
201      for (int i = 0; i < player.getFields().length; i++) {
202
203        if (player.getFields()[i].getColour().equals(colour)) {
204          streetNames[j] = player.getFields()[i].getName();
205          houses[j] = ((Street) (player.getFields()[i])).getNumbOfHouses();
206          j++;
207        }
208      }
209      int streetsWithEqualAmountHouses = 0;
```

```
210
211       // true if the method is needed for sellHouses otherwise false.
212       if (mostHouses) {
213         streetsWithEqualAmountHouses = maximum(houses);
214         if (streetsWithEqualAmountHouses == 0) {
215           return null;
216         }
217       }
218       else {
219         streetsWithEqualAmountHouses = minimum(houses);
220         if (streetsWithEqualAmountHouses == 5) {
221           return null;
222         }
223
224       }
225
226       int amountOfStreets = 0;
227
228       // Finds how many streets needed.
229       for (int i = 0; i < streetNames.length; i++) {
230         if (houses[i] == streetsWithEqualAmountHouses) {
231           amountOfStreets++;
232         }
233       }
234       // An array to hold the name of the differents streets that you can
235       // build on.
236       String[] streetNamesNew = new String[amountOfStreets];
237       j = 0;
238       for (int i = 0; i < streetNames.length; i++) {
239         if (streetsWithEqualAmountHouses == houses[i]) {
240           streetNamesNew[j] = streetNames[i];
241           j++;
242         }
243       }
244       return streetNamesNew;
245     }
246
247     /**
248      * Method setBuilding: Sets a house or a hotel on the street on the
           street
249      * owned by the player.
250      *
251      * @param player
252      *            The player who wants to build a house/hotel.
253      * @param streetName
254      *            The name of the street.
255      */
256     private void setBuilding(Player player, String streetName) {
257       Street street = player.getStreetFromName(streetName);
258       int numbOfHouses = street.getNumbOfHouses();
259
260       if (numbOfHouses == 4) {
261         setHotel(player, street);
262       } else if (numbOfHouses < 4 && numbOfHouses >= 0) {
263         setHouse(player, street);
```

```
264         }
265         GUI.setBalance(player.getName(), player.getAccountBalance());
266     }
267
268     /**
269      * Method setHotel: Sets a hotel and removes houses on a street owned by
            the
270      * player and updates relevant information.
271      *
272      * @param player
273      *            The player who wants to build a hotel.
274      * @param street
275      *            The street the player wants to build on.
276      */
277
278     private void setHotel(Player player, Street street) {
279       if (getHotels() > 0) {
280         if (player.getAccountBalance() >= street.getHousePrice()) {
281           GUI.setHotel(street.getFieldNumber(), true);
282
283           street.changeNumbOfHouses(1);
284           player.changeAccountBalance(-street.getHousePrice());
285           changeHouses(4);
286           changeHotels(-1);
287         } else {
288           GUI.getUserButtonPressed("Du har ikke råd til at købe et hotel på
                denne grund", "Ok");
289         }
290       } else {
291         GUI.getUserButtonPressed("Der er ikke flere hoteller tilbage i banken
              ", "Ok");
292       }
293     }
294
295     /**
296      * Method setHouse: Sets a house on a street owned by the player and
            updates
297      * relevant information.
298      *
299      * @param player
300      *            The player who wants to build a house.
301      * @param street
302      *            The street the player wants to build on.
303      */
304     private void setHouse(Player player, Street street) {
305       if (getHouses() > 0) {
306         if (player.getAccountBalance() >= street.getHousePrice()) {
307           // Adds an amount of houses to the GUI and changes the value of
308           // numbOfHouses.
309           GUI.setHouses(street.getFieldNumber(), street.changeNumbOfHouses(1)
                );
310           player.changeAccountBalance(-street.getHousePrice());
311           changeHouses(-1);
312         } else {
313           GUI.getUserButtonPressed("Du har ikke råd til at købe et hus på
```

```
            denne grund.", "Ok");
      }
    } else {
      GUI.getUserButtonPressed("Der er ikke flere huse tilbage i banken", "
          Ok");
    }
  }

  public void sellHouse(Player player, String streetName) {

    Street street = player.getStreetFromName(streetName);

    if (street.getNumbOfHouses() == 0) {
      GUI.getUserButtonPressed("Der står ingen huse på " + street.getName()
          , "Ok");
    } else {
      if (street.getNumbOfHouses() == 5) {
        changeHotels(1);
        changeHouses(-4);
        GUI.setHotel(street.getFieldNumber(), false);
        GUI.setHouses(street.getFieldNumber(), street.changeNumbOfHouses
            (-1));
        // Der er en fejl her hvor spillerne kan få huse selvom der ikke
        // er nok huse.
      } else {
        changeHouses(1);
        GUI.setHouses(street.getFieldNumber(), street.changeNumbOfHouses
            (-1));
      }

      player.changeAccountBalance(street.getHousePrice() / 2);
      GUI.setBalance(player.getName(), player.getAccountBalance());
    }
  }

  /**
   * Method showHouseMenu: Shows on the GUI which options the player has <
       br>
   * when he wants to buy a building on a street
   *
   * @param player
   *             The player who wants to buy a building
   */
  public void buyBuildingMenu(Player player) {
    while (true) {
      String[] options = MainController.addReturnToArray(
          canBuildOnColourString(player));
      String out = "Hvilken farve ejendom vil du købe huse på?";
      if (options.length == 1) {
        out = "Du har ikke nogle grunde at købe huse på.";
      }
      String answer = GUI.getUserSelection(out, options);
      if (answer.equals("Gå tilbage")) {
        return;
      } else {
```

```
362           while (true) {
363             String[] options2 = MainController.addReturnToArray(
                    streetsWithBuildings(player, answer, false));
364             String answer2;
365             if (options2.length == 1) {
366               answer2 = GUI.getUserSelection("Du kan ikke bygge flere
                      bygninger på denne farve grunde.",
367                   options2);
368             } else {
369               answer2 = GUI.getUserSelection("Du har valgt " + answer + ".
                      Bygningerne på " + answer
370                   + " koster " + player.getHousePriceFromColour(answer) + "
                        kr.", options2);
371             }
372             if (answer2.equals("Gå tilbage")) {
373               break;
374             }
375             setBuilding(player, answer2);
376           }
377         }
378       }
379     }
380
381     public void sellBuildingMenu(Player player) {
382       while (true) {
383         String[] options = MainController.addReturnToArray(
              canBuildOnColourString(player));
384         String out = "Hvilken farve grunde vil du sælge huse på?";
385         if (options.length == 1) {
386           out = "Du har ikke nogle grunde at sælge huse på.";
387         }
388         String answer = GUI.getUserSelection(out, options);
389         if (answer.equals("Gå tilbage")) {
390           return;
391         } else {
392           while (true) {
393             String[] options2 = MainController.addReturnToArray(
                    streetsWithBuildings(player, answer, true));
394             String answer2;
395             if (options2.length == 1) {
396               answer2 = GUI.getUserSelection("Du har ikke flere bygninger på
                      denne farve grunde.", options2);
397             } else {
398               answer2 = GUI.getUserSelection("Du har valgt " + answer + ".
                      Bygningerne sælges for "
399                   + (player.getHousePriceFromColour(answer) / 2) + " kr.",
                        options2);
400             }
401             if (answer2.equals("Gå tilbage")) {
402               break;
403             }
404             sellHouse(player, answer2);
405           }
406         }
407       }
```

```
408         }
409     }
```

12.5.1.3 ChanceCardController

```java
package controller;

import entity.ChanceCardDeck;
import entity.Player;
import entity.chanceCard.*;
import controller.PrisonController;
import desktop_resources.GUI;
import controller.DebtController;

/**
 * This class is responsible for handling the chance cards when the player draws
 * them..
 *
 * @author Gruppe33
 *
 */
public class ChanceCardController {

  // Instance variables.
  private ChanceCardDeck deck;
  private PrisonController prisonController;
  private DebtController bank;
  private MainController mainController;

  /**
   * Constructor: Conctructs a ChanceCardController.
   *
   * @param prisonController
   *            The PrisonController.
   * @param bank
   * @param mainController
   *            The MainController.
   */
  ChanceCardController(PrisonController prisonController, DebtController bank, MainController mainController) {
    deck = new ChanceCardDeck();
    this.prisonController = prisonController;
    this.bank = bank;
    this.mainController = mainController;

  }

  /**
   * Method draw: Decides what happens when a player draws a card from the
   * chance card deck.
   *
   * @param player
   *            The player who draws a chance card.
   */
  public void draw(Player player)

  {
```

```
52      ChanceCard currentCard = deck.draw();
53
54      String type = currentCard.getType();
55      GUI.displayChanceCard(currentCard.getDescription());
56      GUI.getUserButtonPressed("Du har trukket et Prøv-lykken kort \n" +
            currentCard.getDescription(), "Ok");
57      switch (type) {
58      case "Grant":
59        drawGrant(currentCard, player);
60        break;
61      case "Payment":
62        drawPayment(currentCard, player);
63        break;
64      case "MoveToNearestShipping":
65        drawMoveToNearestShipping(currentCard, player);
66        break;
67      case "MoveToPrison":
68        drawMoveToPrison(currentCard, player);
69        break;
70      case "MoveToField":
71        drawMoveToField(currentCard, player);
72        break;
73      case "MoveThreeSteps":
74        drawMoveThreeSteps(currentCard, player);
75        break;
76      // case "Prison": drawPrison(currentCard, player);
77      // break;
78      // case "Tax": drawTaxCard(currentCard, player);
79      // break;
80      // case "Party": drawParty(currentCard, player);
81      // break;
82      }
83    }
84
85    /**
86     * Method DrawMoveToNearestShipping: Decides what happens when a player
87     * draws a chance card of the type 'move to nearest shipping'.
88     *
89     * @param currentCard
90     *            The card drawn.
91     * @param player
92     *            The player who draws the chance card.
93     */
94    private void drawMoveToNearestShipping(ChanceCard currentCard, Player
          player) {
95      MoveToNearestShipping card = (MoveToNearestShipping) currentCard;
96      int[] shippingPos = card.getShippingPositions();
97      int currentPos = player.getPosition();
98
99      // Sets the position of the player to the first shipping field if you
100     // are passed the last shipping field.
101     player.setPosition(shippingPos[0]);
102     // Sets the position of the player to the next shipping field if you
103     // haven't passed the last shipping field.
104     for (int i = 0; i < shippingPos.length; i++) {
```

```
105         if (shippingPos[i] > currentPos) {
106           player.setPosition(shippingPos[i]);
107           break;
108         }
109       }
110       mainController.getLandOnFieldController().setDoubleRent(card.
              getDoubleRent());
111       mainController.movePlayerOnGUI();
112       // Hvis start passeres.
113       if (currentPos > player.getPosition()) {
114         mainController.givePlayer4000();
115       }
116       mainController.getLandOnFieldController().landOnField(player, 0);
117     }
118
119     /**
120      * Method drawMoveToPrison: Decides what happens when a player draws a
121      * chance card of the type 'move to prison'.
122      *
123      * @param currentCard
124      *            The card drawn.
125      * @param player
126      *            The player who draws the chance card.
127      */
128     private void drawMoveToPrison(ChanceCard currentCard, Player player) {
129       prisonController.sendToPrison(player);
130     }
131
132     /**
133      * Method drawMoveToField: Decides what happens when a player draws a
              chance
134      * card of the type 'move to field'.
135      *
136      * @param currentCard
137      *            The card drawn.
138      * @param player
139      *            The player who draws the chance card.
140      */
141     private void drawMoveToField(ChanceCard currentCard, Player player) {
142       mainController.movePlayerTo(((MoveToField) currentCard).getMoveTo());
143       mainController.getLandOnFieldController().landOnField(player, 0);
144
145     }
146
147     /**
148      * Method drawGrant: Decides what happens when a player draws a chance
              card
149      * of the type 'grant'.
150      *
151      * @param currentCard
152      *            The card drawn.
153      * @param player
154      *            The player who draws the chance card.
155      */
156     private void drawGrant(ChanceCard currentCard, Player player) {
```

```
157         Grant grant = (Grant) currentCard;
158         if (player.getFortune() <= 15000) {
159           player.changeAccountBalance(grant.getAmount());
160           GUI.setBalance(player.getName(), player.getAccountBalance());
161         }
162       }
163
164       /**
165        * Method drawMoveThreeSteps: Decides what happens when a player draws a
166        * chance card of the type 'move three steps'.
167        *
168        * @param currentCard
169        *            The card drawn.
170        * @param player
171        *            The player who draws the chance card.
172        */
173       private void drawMoveThreeSteps(ChanceCard currentCard, Player player) {
174         MoveThreeSteps move = (MoveThreeSteps) currentCard;
175         mainController.movePlayer(move.getSteps());
176         mainController.getLandOnFieldController().landOnField(player, 0);
177       }
178
179       /**
180        * Method drawPayment: Decides what happens when a player draws a chance
181        * card of the type 'payment'.
182        *
183        * @param currentCard
184        *            The card drawn.
185        * @param player
186        *            The player who draws the chance card.
187        */
188       private void drawPayment(ChanceCard currentCard, Player player) {
189
190         if (bank.playerAffordPayment(player, ((Payment) currentCard).getAmount
               ())) {
191           player.changeAccountBalance(((Payment) currentCard).getAmount());
192           GUI.setBalance(player.getName(), player.getAccountBalance());
193         }
194       }
195
196   }
```

12.5.1.4 DebtController

```java
package controller;

import desktop_resources.GUI;
import entity.field.Ownable;
import entity.field.Street;
import entity.Player;

/**
 * This class is resposible for handling the debt of players when they owe the
 * bank or another player money.
 *
 * @author Gruppe33
 *
 */
public class DebtController {

  // Instance variables
  private BuildingController buildingController;

  /**
   * Constructor: Constructs a DebtController.
   *
   * @param buildingController
   *            The BuildingController.
   */
  public DebtController(BuildingController buildingController) {
    this.buildingController = buildingController;
  }

  /**
   * Checks if a Street field owned by a player can be sold. The field can not
   * be sold if there are buildings on any of his fields with the same color.
   *
   * @param street
   *            The Street field to check for.
   * @param player
   *            The owner of the Street field.
   * @return Whether the field can be sold or not.
   */
  private boolean canStreetBeSold(Street street, Player player) {
    if (street.getNumbOfHouses() > 0) {
      return false;
    }

    for (int i = 0; i < player.getFields().length; i++) {
      if (player.getFields()[i].getColour().equals(street.getColour())) {
        if (((Street) player.getFields()[i]).getNumbOfHouses() > 0)
          return false;
      }
    }
```

```
51
52      return true;
53
54    }
55
56    /**
57     * Gets the player's fields without buildings on them.
58     *
59     * @param player
60     *            The player to get the fields from.
61     * @return an array of the player's fields.
62     */
63    private Ownable[] getFieldsWithoutBuildings(Player player) {
64      // All the player's fields.
65      Ownable[] playerFields = player.getFields();
66
67      if (playerFields == null) {
68        return null;
69      }
70      // Array to hold the fields without buildings on them.
71      Ownable[] fieldsWithoutBuildings = new Ownable[playerFields.length];
72
73      // Counts the fields of the object type Street
74      int j = 0;
75
76      // Go through the player's fields.
77      for (int i = 0; i < playerFields.length; i++) {
78        // If a field is a Street object
79        if (playerFields[i] instanceof Street) {
80
81          if (canStreetBeSold((Street) playerFields[i], player)) {
82            fieldsWithoutBuildings[j] = playerFields[i];
83            j++;
84          }
85
86        } else {
87          fieldsWithoutBuildings[j] = playerFields[i];
88          j++;
89        }
90      }
91      return fieldsWithoutBuildings;
92    }
93
94    /**
95     * Creates a String array with the fields names and price divided by 2.
96     *
97     * @param fields
98     *            The fields to be operated.
99     * @return String array of the fields names and price.
100    */
101   private String[] fieldsToString(Ownable[] fields) {
102
103     if (fields == null) {
104       return null;
105     }
```

```
106       String[] fieldsNames = new String[fields.length];
107       int fieldsNamesLength = 0;
108       // Go through the fields array
109       for (int i = 0; i < fields.length; i++) {
110         if (fields[i] != null) {
111           // Type the fields name and price into the String array.
112           fieldsNames[i] = fields[i].getName() + " " + "(" + (fields[i].
                getPrice() / 2) + " kr.)";
113           fieldsNamesLength++;
114         }
115       }
116
117       if (fieldsNamesLength == 0) {
118         return null;
119       }
120       String[] fieldsNamesNew = new String[fieldsNamesLength];
121       for (int i = 0; i < fieldsNamesLength; i++) {
122         fieldsNamesNew[i] = fieldsNames[i];
123       }
124
125       return fieldsNamesNew;
126     }
127
128     /**
129      * Sells the field owned by the player. The field can only be sold if the
130      * field itself and its color siblings does not have any buildings on
             them.
131      *
132      * @param player
133      *            Owner of the field.
134      */
135     public void sellField(Player player) {
136       Ownable[] fieldsWithoutBuildings = getFieldsWithoutBuildings(player);
137       String[] fieldsToString = fieldsToString(fieldsWithoutBuildings);
138       String message = "Hvilken grund vil du sælge?";
139       String failMessage = "Du har ikke nogle grunde at sælge";
140       String[] options = MainController.addReturnToArray(fieldsToString);
141
142       if (options.length == 1) {
143         GUI.getUserSelection(failMessage, options);
144         return;
145       }
146
147       String answer = GUI.getUserSelection(message, options);
148
149       if (answer.equals("Gå tilbage")) {
150         return;
151       } else {
152         // Index of the selected field in the list.
153         int index = -1;
154         for (int i = 0; i < fieldsToString.length; i++) {
155           if (answer.equals(fieldsToString[i])) {
156             index = i;
157             break;
158           }
```

```
159         }
160
161         player.changeAccountBalance(fieldsWithoutBuildings[index].getPrice()
                / 2);
162         player.removeField(fieldsWithoutBuildings[index]);
163         GUI.setBalance(player.getName(), player.getAccountBalance());
164         GUI.removeOwner(fieldsWithoutBuildings[index].getFieldNumber());
165       }
166     }
167
168     /**
169      * The method playerAffordPayment checks to see if a player can afford a
170      * payment. If the player can not afford the required payment, the player
171      * loses the game.
172      *
173      * @param player
174      *            The player to be checked.
175      * @param payment
176      *            The payment to be withdrawn from the player's account.
177      * @return the current state of the player.setLost() condition. If true,
             the
178      *         player will lose. If false, the player may pay the payment,
             and
179      *         continue playing.
180      */
181     public boolean playerAffordPayment(Player player, int payment) {
182       if (player.getAccountBalance() <= payment) {
183         return handleDebt(player, payment);
184
185       } else
186         return true;
187     }
188
189     /**
190      * The method playerHasLost is a method which, when a player loses, sets
             all
191      * of the player's owned fields free for other players to purchase again.
192      *
193      * @param player
194      *            The affected player.
195      *
196      */
197     public void playerHasLost(Player player) {
198       player.setHasLost(true);
199
200       if (player.getFields() != null) {
201
202         Ownable[] fields = new Ownable[player.getFields().length];
203         fields = player.getFields();
204
205         // Go through the fields array
206         for (int i = 0; i < fields.length; i++) {
207           // If the field is a Street field
208           if (fields[i] instanceof Street) {
209             Street streetField = (Street) fields[i];
```

```
210             if (streetField.getNumbOfHouses() == 5) {
211               buildingController.changeHotels(1);
212             } else {
213               buildingController.changeHouses(streetField.getNumbOfHouses());
214             }
215             streetField.changeNumbOfHouses(-streetField.getNumbOfHouses());
216             GUI.setHouses(streetField.getFieldNumber(), streetField.
                    getNumbOfHouses());
217             GUI.setHotel(streetField.getFieldNumber(), false);
218             GUI.removeOwner(streetField.getFieldNumber());
219           }
220           player.removeField(fields[i]);
221         }
222       }
223
224       player.changeAccountBalance(-player.getAccountBalance() - 1);
225       GUI.setBalance(player.getName(), player.getAccountBalance());
226       GUI.getUserButtonPressed("Du tabte spillet.", "Ok");
227     }
228
229     /**
230      * Checks if a player can handle the debt he owes.
231      *
232      * @param player
233      *            The player that owes debt.
234      * @param debt
235      *            The debt that is owed.
236      * @return Whether the player can handle the debt or not.
237      */
238     public boolean handleDebt(Player player, int debt) {
239       // If the player's debt is bigger than the players (fortune minus his
240       // balance) divided by 2.
241       if (((player.getFortune() - player.getAccountBalance()) / 2) + player.
              getAccountBalance() >= debt) {
242         while (true) {
243           String sellField = "Sælg grund";
244           String sellHouse = "Sælg hus";
245           String payDebt = "Betal gæld";
246           String message = "Du har ikke råd til at betale din gæld, hvad vil
                 du gøre?";
247           String[] options = { sellField, sellHouse };
248
249           if (player.getAccountBalance() >= debt) {
250             String[] expandedOptions = { payDebt, options[0], options[1] };
251             options = expandedOptions;
252             message = "Du kan nu betale din gæld. Vil du sælge yderligere
                   grunde eller bygninger?";
253           }
254           String userSelection = GUI.getUserSelection(message, options);
255
256           if (userSelection.equals(sellHouse))
257             buildingController.sellBuildingMenu(player);
258           else if (userSelection.equals(sellField))
259             sellField(player);
260           else if (userSelection.equals(payDebt))
```

```
261                 break;
262             }
263
264             return true;
265         }
266
267         else {
268             playerHasLost(player);
269             return false;
270         }
271     }
272 }
```

12.5.1.5 FieldController

```
1  package controller;
2
3  import desktop_resources.GUI;
4  import entity.Player;
5  import entity.field.*;
6  import entity.GameBoard;
7
8  /**
9   * This class is responsible for the fields. It controls what happens when
        the
10  * players land on a field.
11  *
12  * @author Gruppe33
13  *
14  */
15 public class FieldController {
16
17    // Instance variables
18    private PrisonController prisonController;
19    private ChanceCardController chanceCardController;
20    private DebtController bankController;
21    private GameBoard gameBoard;
22    private boolean doubleRent = false;
23
24    /**
25     * Constructor: Constructs a FieldController
26     *
27     * @param prisonController
28     *            The PrisonController.
29     * @param mainController
30     *            The MainController.
31     * @param propertyController
32     *            The PropertyController.
33     */
34    public FieldController(PrisonController prisonController, MainController
          mainController,
35        BuildingController propertyController, DebtController bankController)
              {
36      this.bankController = bankController;
37      this.prisonController = prisonController;
38      this.chanceCardController = new ChanceCardController(prisonController,
            bankController, mainController);
39      this.gameBoard = new GameBoard();
40    }
41
42    /**
43     * Method landOnField: Decides what has to be done when a player lands on
            a
44     * field.
45     *
46     * @param player
47     *            The player to land on the field.
48     * @param diceSum
```

```
49      *            The dice sum of the player's dice roll.
50      */
51     public void landOnField(Player player, int diceSum) {
52
53       Field field = gameBoard.getField(player.getPosition());
54       String type = field.getType();
55       switch (type) {
56       case "Ejendom":
57       case "Rederi":
58       case "Tapperi":
59         landOnOwnable(field, player, diceSum);
60         break;
61       case "Chancekort":
62         landOnChanceField(player);
63         break;
64       case "Skat":
65         landOnTax(field, player);
66         break;
67       default:
68         landOnNeutral(player);
69         break;
70       }
71       // Reset the doubleRent, this variable is used in one type of
             chanceCard
72       doubleRent = false;
73     }
74
75     /**
76      * Method landOnOwnable: Decides what has to be done when a player lands
           on
77      * an ownable field.
78      *
79      * @param field
80      *            The field the player landed on.
81      * @param player
82      *            The player to land on the field.
83      * @param diceSum
84      *            The dice sum of the player's dice roll
85      */
86     public void landOnOwnable(Field field, Player player, int diceSum) {
87       Ownable ownable = (Ownable) field;
88       // If the field is not owned.
89       if (!ownable.isFieldOwned()) {
90         boolean bought = GUI.getUserLeftButtonPressed(
91             "Du landte på " + ownable.getName() + ", vil du købe grunden?", "
                 Ja", "Nej");
92         // Buy the field if the player wants to and he ca afford it.
93         if (bought) {
94           if (player.buyField(ownable)) {
95             GUI.setOwner(player.getPosition(), player.getName());
96             GUI.setBalance(player.getName(), player.getAccountBalance());
97           } else {
98             GUI.getUserButtonPressed("Du har ikke råd til at købe " + ownable
                   .getName(), "Ok");
99           }
```

```
100         }
101       }
102       // If the field is owned.
103       else {
104         // If the field is owned by another player, and he is not in prison.
105         if (ownable.isFieldOwnedByAnotherPlayer(player) && !ownable.getOwner
              ().getInPrison()) {
106           // If the field is a Brewery.
107           if (ownable.getType().equals("Tapperi")) {
108             Brewery brewery = (Brewery) (ownable);
109             brewery.setDiceSum(diceSum);
110           }
111           int rent = ownable.getRent();
112           // Only used for some specific chancecards
113           if (doubleRent) {
114             rent = rent * 2;
115           }
116           GUI.getUserButtonPressed(
117               "Du landte på " + ownable.getName() + ". Grunden er ejet af " +
                    ownable.getOwner().getName()
118                   + ", og du skal betale en leje på " + rent + " kr.",
119               "Betal " + rent + " kr. til " + ownable.getOwner().getName());
120
121           // If the player can afford to pay rent.
122           if (bankController.playerAffordPayment(player, rent)) {
123             player.payRent(ownable.getOwner(), rent);
124             GUI.setBalance(ownable.getOwner().getName(), ownable.getOwner().
                  getAccountBalance());
125             GUI.setBalance(player.getName(), player.getAccountBalance());
126           }
127         }
128       }
129     }
130
131     /**
132      * Method landOnField: Decides what has to be done when a player lands on
             a
133      * Tax field.
134      *
135      * @param field
136      *            The field the player landed on.
137      * @param player
138      *            The player to land on the tax field.
139      */
140     public void landOnTax(Field field, Player player) {
141       int rentAmount = 0;
142       Tax tax = (Tax) (field);
143       if (tax.getHasTaxRate() == true) {
144
145         // The tax rate rent to be paid.
146         int rentTaxRate = (int) (0.1 * player.getFortune());
147
148         // The rent amount to be paid.
149         rentAmount = 4000;
150
```

```
151         boolean payTaxRate = GUI.getUserLeftButtonPressed("Du skal betale
                indkomstskat.",
152             "Betal 10% (" + rentTaxRate + " kr.)", "Betal 4.000 kr.");
153         if (payTaxRate) {
154           // Subtract the tax rate rent from the player's balance.
155           if (bankController.playerAffordPayment(player, rentTaxRate))
156             player.changeAccountBalance(-rentTaxRate);
157
158         } else {
159           // Subtract the rent amount from the player's balance.
160           if (bankController.playerAffordPayment(player, rentAmount))
161             player.changeAccountBalance(-rentAmount);
162         }
163       } else {
164         GUI.getUserButtonPressed("Du skal betale ekstraordinærstatsskat.", "
                Betal 2.000 kr.");
165
166         // The rent amount to be paid.
167         rentAmount = 2000;
168
169         // Subtract the rent amount from the player's balance.
170         if (bankController.playerAffordPayment(player, rentAmount))
171           player.changeAccountBalance(-rentAmount);
172       }
173       GUI.setBalance(player.getName(), player.getAccountBalance());
174     }
175
176     /**
177      * Method landOnNeutral: Decides what has to be done when a player lands
                on
178      * the following fields: <br>
179      * - Sent to prison <br>
180      * - Start <br>
181      * - Parking <br>
182      * - Visiting prison. <br>
183      *
184      * @param player
185      *            The player who landed on the neutral field.
186      */
187     public void landOnNeutral(Player player) {
188       int prisonFieldNum = 31;
189       if (player.getPosition() == prisonFieldNum) {
190         prisonController.sendToPrison(player);
191       }
192     }
193
194     /**
195      * Method landOnChanceField: Decides what has to be done when a player
                lands
196      * on a chance field.
197      *
198      * @param player
199      *            The player who landed on the chance field.
200      */
201     public void landOnChanceField(Player player) {
```

```
202         GUI.getUserButtonPressed("Du landte på prøv lykken.", "Træk et kort");
203         chanceCardController.draw(player);
204     }
205 
206     /**
207      * Method setDoubleRent: Sets the value of doubleRent.
208      *
209      * @param doubleRent
210      *             The value to be set.
211      */
212     public void setDoubleRent(boolean doubleRent) {
213         this.doubleRent = doubleRent;
214 
215     }
216 
217     /**
218      * Method TESTgetGameBoard: Used to get the Gameboard in the testmode
219      * controller.
220      *
221      * @return Gameboard
222      */
223     public GameBoard TESTgetGameBoard() {
224         return gameBoard;
225     }
226 }
```

12.5.1.6 GUICreator

```java
package controller;

import java.awt.Color;

import data.Reader;
import desktop_codebehind.Car;
import desktop_fields.Brewery;
import desktop_fields.Chance;
import desktop_fields.Field;
import desktop_fields.Jail;
import desktop_fields.Refuge;
import desktop_fields.Shipping;
import desktop_fields.Start;
import desktop_fields.Street;
import desktop_fields.Tax;
import desktop_resources.GUI;

/**
 * This class is responsible for creating the GUI.
 *
 * @author Gruppe33
 *
 */
public class GUICreator {

  // Instance variables
  private Car[] cars;
  private Field[] fields;
  private int fieldCounter;

  /**
   * GUICreator constructor
   */
  public GUICreator() {
    Reader dataReader = new Reader("src/data/Feltliste.txt");
    String[][] data = dataReader.readFile();
    beginBoardBuilding();
    for (int i = 0; i < 40; i++) {
      addField(data[i]);
    }
    endBoardBuilding();
  }

  /**
   * Method beginBoardbuidling: Create a list of GUI Fields. This should be
   * used before the addField Method.
   *
   */
  public void beginBoardBuilding() {
    fields = new Field[40];
    fieldCounter = 1;
  }

```

```
54    /**
55     * This method constructs the fields on the GUI, after they have been
          added
56     * by the addField method.
57     */
58    public void endBoardBuilding() {
59      GUI.create(fields);
60      cars = createCars();
61    }
62
63    /**
64     * Method addField: Adds a field to the GUI Game board
65     *
66     * @param information
67     *             Array of all the field information. This information
          depends
68     *             on the field in question. This method calls many different
69     *             submethods using a switch. These methods construct the
70     *             different types of fields.
71     */
72    public void addField(String[] information) {
73      // Gets the type of the fields.
74      String fieldType = information[information.length - 2];
75
76      // Calls the respective method that handles specific types of fields.
77      switch (fieldType) {
78      case "Ejendom":
79        addStreet(information);
80        break;
81      case "Rederi":
82        addShipping(information);
83        break;
84      case "Tapperi":
85        addBrewery(information);
86        break;
87      case "Chancekort":
88        addChance(information);
89        break;
90      case "Skat":
91        addTax(information);
92        break;
93      case "Start":
94        addStart(information);
95        break;
96      case "Fængsel":
97        addJail(information);
98        break;
99      case "Besøg":
100       addVisit(information);
101       break;
102     case "Parkering":
103       addParking(information);
104       break;
105     default:
106       // Throws an error message when the field type is not regonised.
```

```
107             System.out.println("GUICreator: Not a valid field at field number: "
                    + fieldCounter);
108         }
109         fieldCounter++;
110     }
111
112     /**
113      * Method addStreet: Adds a Street field to the GUI's fields array.
114      *
115      * @param fieldData
116      *             String array of all the fields data.
117      */
118     private void addStreet(String[] fieldData) {
119         Color color = getFieldColor(fieldData[1]);
120         fields[fieldCounter - 1] = new Street.Builder().setTitle(fieldData[0]).
                setSubText(fieldData[2])
121                 .setDescription("").setBgColor(color).setRent(fieldData[3]).build()
                        ;
122     }
123
124     /**
125      * Method addShipping: Adds a Shipping field to the GUI's fields array.
126      *
127      * @param fieldData
128      *             String array of all the fields data.
129      */
130     private void addShipping(String[] fieldData) {
131         fields[fieldCounter - 1] = new Shipping.Builder().setTitle(fieldData
                [0]).setSubText(fieldData[2])
132                 .setDescription(fieldData[1])
133                 .setBgColor(
134                         Color.getHSBColor((float) (216.21 / 360.0), (float) (72.5 /
                                100.0), (float) (62.75 / 100.0)))
135                 .setPicture("src/data/pictures/Ferry.png").build();
136     }
137
138     /**
139      * Method addBrewery: Adds a Brewery field to the GUI's fields array.
140      *
141      * @param fieldData
142      *             String array of all the fields data.
143      */
144     private void addBrewery(String[] fieldData) {
145         fields[fieldCounter - 1] = new Brewery.Builder().setTitle(fieldData[0])
                .setSubText(fieldData[2])
146                 .setDescription(fieldData[1]).setBgColor(Color.BLACK).build();
147     }
148
149     /**
150      * Method addChance: Adds a chanceCard field to the GUI's fields array.
151      *
152      * @param fieldData
153      *             String array of all the fields data.
154      */
155     private void addChance(String[] fieldData) {
```

```
156         fields[fieldCounter - 1] = new Chance.Builder().setBgColor(Color.
                LIGHT_GRAY).build();
157     }
158 
159     /**
160      * Method addTax: Adds a Tax field to the GUI's fields array.
161      *
162      * @param fieldData
163      *             String array of all the fields data.
164      */
165     private void addTax(String[] fieldData) {
166         fields[fieldCounter - 1] = new Tax.Builder().setTitle(fieldData[0]).
                setSubText(fieldData[2])
167                 .setDescription(fieldData[1]).setBgColor(Color.LIGHT_GRAY).build();
168     }
169 
170     /**
171      * Method addStart: Adds a Start field to the GUI's fields array.
172      *
173      * @param fieldData
174      *             String array of all the fields data.
175      */
176     private void addStart(String[] fieldData) {
177         fields[fieldCounter - 1] = new Start.Builder().setTitle(fieldData[0]).
                setSubText(fieldData[2])
178                 .setDescription(fieldData[1]).setBgColor(Color.RED).build();
179     }
180 
181     /**
182      * Method addJail: Adds a Go To Jail field to the GUI's fields array.
183      *
184      * @param fieldData
185      *             String array of all the fields data.
186      */
187     private void addJail(String[] fieldData) {
188         fields[fieldCounter - 1] = new Jail.Builder().setSubText(fieldData[0]).
                setDescription(fieldData[1])
189                 .setBgColor(Color.GRAY).build();
190     }
191 
192     /**
193      * Method addVisit: Adds a Visit Jail field to the GUI's fields array.
194      *
195      * @param fieldData
196      *             String array of all the fields data.
197      */
198     private void addVisit(String[] fieldsData) {
199         fields[fieldCounter - 1] = new Jail.Builder().setSubText(fieldsData[0])
                .setDescription(fieldsData[1])
200                 .setBgColor(Color.GRAY).build();
201     }
202 
203     /**
204      * Method addParking: Adds a Parking field to the GUI's fields array.
205      *
```

```
206     * @param fieldData
207     *            String array of all the fields data.
208     */
209    private void addParking(String[] fieldsData) {
210      fields[fieldCounter - 1] = new Refuge.Builder().setSubText(fieldsData
           [0]).setDescription(fieldsData[1])
211          .setBgColor(
212              Color.getHSBColor((float) (198.1 / 360.0), (float) (100.0 /
                   100.0), (float) (90.98 / 100.0)))
213          .setPicture("src/data/pictures/Parkeringslogo.png").build();
214    }
215
216    /**
217     * Method addPlayersToGUI: Adds the players to the GUI.
218     *
219     * @param players
220     *            String array of player names
221     */
222    private void addPlayersToGUI(String[] players) {
223      for (int i = 0; i < players.length; i++) {
224        GUI.addPlayer(players[i], 30000, cars[i]);
225        GUI.setCar(1, players[i]);
226      }
227    }
228
229    /**
230     * Method askNumberOfPlayers: Lets the player insert how many players are
231     * participating in the game.
232     *
233     * @return (int) The number of players that are playing.
234     */
235    private int askNumberOfPlayers() {
236      // Ask the players how many are playing
237      int numberOfPlayers = GUI.getUserInteger("Indtast hvor mange spillere
           der vil spille (2-6)", 2, 6);
238      return numberOfPlayers;
239    }
240
241    /**
242     * Method getPlayerNames: Let all the players enter their player names.
243     *
244     * @return String array of the player names.
245     */
246    public String[] getPlayerNames() {
247      int numberOfPlayers = askNumberOfPlayers();
248      String[] playerNames = new String[numberOfPlayers];
249
250      // Lets player 1 enter a name that is not an empty String.
251      do {
252        playerNames[0] = GUI.getUserString("Indtast navn for spiller nummer
             1:");
253      } while (playerNames[0].equals(""));
254
255      // Lets all the users insert their player names
256      for (int i = 1; i < numberOfPlayers; i++) {
```

```
257         boolean nameEqual = true;
258         playerNames[i] = null;
259
260         while (nameEqual) {
261           // Lets player 2-6 enter a name that is not an empty String.
262           do {
263             playerNames[i] = GUI.getUserString("Indtast navn for spiller
                  nummer: " + (i + 1));
264           } while (playerNames[i].equals(""));
265
266           // Goes through the already added player names
267           for (int j = 0; j < i; j++) {
268
269             // If the entered player name matches an already existing
270             // name
271             if (playerNames[j].equals(playerNames[i])) {
272               GUI.getUserButtonPressed("Navnet findes allerede. Vælg venligst
                    et andet navn.", "Ok");
273               nameEqual = true;
274               break;
275             } else {
276               nameEqual = false;
277             }
278           }
279         }
280       }
281       addPlayersToGUI(playerNames);
282       return playerNames;
283     }
284
285     /**
286      * Method createCars builds all the available cars in the game.
287      *
288      * @return Car array of all the cars.
289      */
290     private Car[] createCars() {
291       Car[] carArray = new Car[6];
292       carArray[0] = new Car.Builder().primaryColor(Color.BLUE).secondaryColor
              (Color.BLACK).typeUfo().patternFill()
293           .build();
294       carArray[1] = new Car.Builder().primaryColor(Color.RED).secondaryColor(
              Color.BLACK).typeUfo().patternFill()
295           .build();
296       carArray[2] = new Car.Builder().primaryColor(Color.GREEN).
              secondaryColor(Color.BLACK).typeTractor()
297           .patternFill().build();
298       carArray[3] = new Car.Builder().primaryColor(Color.YELLOW).
              secondaryColor(Color.BLACK).typeTractor()
299           .patternFill().build();
300       carArray[4] = new Car.Builder().primaryColor(Color.PINK).secondaryColor
              (Color.BLACK).typeRacecar().patternFill()
301           .build();
302       carArray[5] = new Car.Builder().primaryColor(Color.WHITE).
              secondaryColor(Color.BLACK).typeRacecar()
303           .patternFill().build();
```

```
304       return carArray;
305     }
306
307     /**
308      * Method getFieldColor: Gets the color of the field
309      *
310      * @param fieldColor
311      *            String describing the color in danish.
312      * @return The corresponding Color object.
313      */
314     private Color getFieldColor(String fieldColor) {
315       Color color;
316       switch (fieldColor) {
317       case "Rød":
318         color = Color.RED;
319         break;
320       case "Blå":
321         color = Color.BLUE;
322         break;
323       case "Orange":
324         color = Color.ORANGE;
325         break;
326       case "Grøn":
327         color = Color.GREEN;
328         break;
329       case "Grå":
330         color = Color.GRAY;
331         break;
332       case "Hvid":
333         color = Color.WHITE;
334         break;
335       case "Gul":
336         color = Color.YELLOW;
337         break;
338       case "Lilla":
339         color = Color.MAGENTA;
340         break;
341       default:
342         color = Color.GRAY;
343         break;
344       }
345       return color;
346     }
347 }
```

12.5.1.7 PrisonController

```java
package controller;

import desktop_resources.GUI;
import entity.Player;

/**
 * This class is responsible for anything to do with the prison.
 *
 * @author Gruppe33
 *
 */
public class PrisonController {

  // Instance variables
  private MainController mainController;
  private DebtController bankController;

  /**
   * Constructor: Constructs a PrisonController.
   *
   * @param mainController
   *            MainController.
   * @param bankController
   */
  public PrisonController(MainController mainController, DebtController
      bankController) {
    this.mainController = mainController;
    this.bankController = bankController;
  }

  /**
   * Method sendToPrison: Sends a player to the prison.
   *
   * @param player
   *            Player to be send to prison.
   */
  public void sendToPrison(Player player) {
    player.setInPrison(true);
    GUI.getUserButtonPressed("De fængsles.", "Afslut tur");
    player.setPosition(11);
    mainController.movePlayerOnGUI();

  }

  /**
   * Method inPrison: Controls the prison turn.
   *
   * @param player
   *            Player in prison.
   * @return boolean True if the player can roll after he gets out of
      prison
   *         otherwise false.
   */
```

```
52  public boolean inPrison(Player player) {
53    boolean boughtOut = false;
54    String payOut = "Betal dig ud: 1.000 kr";
55    String rollOut = "Slå dig ud fængslet med terningerne";
56    String userDecision = null;
57    // If this is your fourth round in jail.
58    if (player.getTurnsInPrison() >= 3) {
59      player.setInPrison(false);
60      player.setTurnsInPrison(0);
61      GUI.getUserButtonPressed("Du har nu siddet i fængsel i 3 runder, og
            bliver derfor sat fri.", "Ok");
62      return true;
63    }
64    // If you can afford to pay yourself out of jail.
65    if (player.getAccountBalance() > 1000) {
66      userDecision = GUI.getUserButtonPressed("Du er i fængsel, hvad vil du
            gøre?", payOut, rollOut);
67    } else if (((player.getFortune() - player.getAccountBalance()) / 2) +
         player.getAccountBalance() >= 1000) {
68      userDecision = GUI.getUserButtonPressed(
69          "Du er i fængsel, hvad vil du gøre?\n"
70              + "Du har dog ikke råd til at betale dig ud, så hvis du vil g
                  øre det, er du nødt til at sælge bygninger eller grunde.",
71          payOut, rollOut);
72    } else {
73      userDecision = GUI.getUserButtonPressed(
74          "Du er i fængsel og har ikke mulighed for at betale dig ud, da du
              ikke har råd.", rollOut);
75    }
76
77    // If the player chooses to pay himself out, release him from jail.
78    if (userDecision.equals(payOut)) {
79      if (bankController.playerAffordPayment(player, 1000)) {
80        player.changeAccountBalance(-1000);
81        GUI.setBalance(player.getName(), player.getAccountBalance());
82        player.setInPrison(false);
83        boughtOut = true;
84      }
85    }
86    // Roll the dice and do whats appropiate.
87    else {
88      int diceSum = mainController.rollDice();
89      player.changeTurnsInPrison(1);
90      if (mainController.checkForExtraTurn()) {
91        GUI.getUserButtonPressed("Du slog to ens! Du er nu fri og må slå
              igen.", "Slå med terningerne");
92        player.setInPrison(false);
93        mainController.movePlayer(diceSum);
94        mainController.getLandOnFieldController().landOnField(player,
              diceSum);
95        mainController.playerTurnDecision();
96      } else {
97        GUI.getUserButtonPressed("Du slog dig ikke ud af fængslet", "Afslut
              tur");
98      }
```

```
99        }
100       return boughtOut;
101     }
102
103     /**
104      * Method checkInPrison: Checks whether a player is in prison or not.
105      *
106      * @param player
107      *             Player to check for.
108      * @return True if the player is in prison.
109      */
110     public boolean checkInPrison(Player player) {
111       if (player.getInPrison())
112         return true;
113       else
114         return false;
115     }
116 }
```

12.5.2 data

12.5.2.1 Reader

```
1  package data;
2
3  import java.io.File;
4  import java.io.FileNotFoundException;
5  import java.util.Scanner;
6
7  /**
8   * This class is responsible for reading the fields from a file.
9   * @author Gruppe33
10  *
11  */
12 public class Reader {
13
14    //Instance variables
15    File file;
16
17    /**
18     * Constructor: Constructs a Reader.
19     * @param fileName The file to be read.
20     */
21    public Reader(String fileName)
22    {
23      file = new File(fileName);
24    }
25
26    /**
27     * Method readFile: Returns the information in the file as a 2-
          dimensional String array.
28     * @return The 2-dimensional String array.
29     */
30    public String[][] readFile()
31    {
32      String[][] fileData = new String[40][12];
33      try {
34        Scanner scanner = new Scanner(file);
35
36        for(int i = 0; i < 40; i++)
37        {
38          fileData[i] = scanner.nextLine().split(";");
39        }
40        scanner.close();
41      }
42      catch(FileNotFoundException e) {
43        e.printStackTrace();
44      }
45      return fileData;
46    }
47
48    /**
49     * Method formatFileData: Returns the data without '.'.
50     * @param fieldData The data to be formatted.
```

```
51      * @return Returns the formatted data.
52      */
53     public String[][] formatFileData(String[][] fieldData)
54     {
55       String[][] formattedFieldData = new String[fieldData.length][fieldData
             [1].length];
56       for(int i = 0; i < fieldData.length; i++)
57       {
58         for(int j = 0; j < fieldData[i].length; j++)
59         {
60           formattedFieldData[i][j] = fieldData[i][j].replaceAll("\\.", "");
61         }
62       }
63       return formattedFieldData;
64     }
65 }
```

12.5.3 entity

12.5.3.1 Account

```
1  package entity;
2
3  /**
4   * This class is responsible for handling the players currency balance.
5   *
6   * @author Gruppe33
7   *
8   */
9  public class Account {
10
11   // Instance variables
12   private int balance;
13
14   /**
15    * Constructor: Constructs an Account.
16    *
17    * @param balance
18    *             The start balance of the account..
19    */
20   public Account(int balance) {
21     this.balance = balance;
22   }
23
24   /**
25    * Method getBalance: Returns the balance of the account.
26    *
27    * @return balance The Balance of the account.
28    */
29   public int getBalance() {
30     return balance;
31   }
32
33   /**
34    * Method changeBalance: Calculates the new balance based on the
          parameter
35    * value.
36    *
37    * @param value
38    *             The value that the balance of the account should be changed
39    *             with. If the parameter is positive the method adds value to
40    *             the balance. If the parameter is negative the method
          subtracts
41    *             value from balance.
42    * @return The method return true if the change of balance succeeded.
          False
43    *         otherwise.
44    */
45   public void changeBalance(int value) {
46     // Checks if the balance overflows
47     if (balance + value < balance && value > 0) {
48     }
```

```
49      // Checks if the balance underflows
50      else if (balance + value > balance && value < 0) {
51      }
52      // Changes balance
53      else {
54        balance = balance + value;
55      }
56
57    }
58
59    /**
60     * OBS: This method is only used for testing. Method setBalance sets the
61     * balance of the Account.
62     */
63    public void setBalance(int value) {
64      balance = value;
65    }
66  }
```

12.5.3.2 ChanceCardDeck

```
1  package entity;
2
3  import java.io.File;
4  import java.io.FileNotFoundException;
5  import java.util.Random;
6  import java.util.Scanner;
7
8  import entity.chanceCard.*;
9
10 /**
11  * This class is responsible for creating, holding and giving class cards.
        This
12  * is primarily used by the chancecard controller.
13  *
14  * @author Gruppe33
15  *
16  */
17 public class ChanceCardDeck {
18   // Instance variables
19   private ChanceCard[] chanceCardDeck;
20
21   /**
22    * Constructor: Constructs a chanceCard deck.
23    */
24   public ChanceCardDeck() {
25     chanceCardDeck = new ChanceCard[37];
26     int cardNumber = 0;
27
28     // Imports file
29     String fileName = "data.csv";
30     File file = new File(fileName);
31
32     // Scans the file and splits it into an array based on the position of
          ,
33     try {
34       Scanner inputStream = new Scanner(file);
35       String data = inputStream.nextLine();
36       for (int i = 0; i < chanceCardDeck.length; i++) {
37         data = inputStream.nextLine();
38         String[] fromInfo = data.split(",");
39
40         switch (fromInfo[0]) {
41         case "Grant":
42           chanceCardDeck[cardNumber] = createGrant(fromInfo);
43           break;
44         case "Party":
45           chanceCardDeck[cardNumber] = createParty(fromInfo);
46           break;
47         case "Payment":
48           chanceCardDeck[cardNumber] = createPayment(fromInfo);
49           break;
50         case "Prison":
51           chanceCardDeck[cardNumber] = createPrison(fromInfo);
```

```
52            break;
53          case "TaxCard":
54            chanceCardDeck[cardNumber] = createTaxCard(fromInfo);
55            break;
56          case "MoveToNearestShipping":
57            chanceCardDeck[cardNumber] = createMoveToNearestShipping(fromInfo
                  );
58            break;
59          case "MoveToPrison":
60            chanceCardDeck[cardNumber] = createMoveToPrison(fromInfo);
61            break;
62          case "MoveToField":
63            chanceCardDeck[cardNumber] = createMoveToField(fromInfo);
64            break;
65          case "MoveThreeSteps":
66            chanceCardDeck[cardNumber] = createMoveThreeSteps(fromInfo);
67            break;
68          default:
69            System.out.println("Error: This card is unknown");
70            System.out.println(chanceCardDeck[cardNumber].getType());
71            break;
72          }
73
74          cardNumber++;
75
76        }
77        shuffle();
78        inputStream.close();
79      } catch (FileNotFoundException e) {
80        e.printStackTrace();
81      }
82    }
83
84    /**
85     * Method draw: Returns the first card in the deck and afterwards puts it
              in
86     * the bottom of the deck
87     *
88     * @return The first card of the deck.
89     */
90    public ChanceCard draw() {
91      ChanceCard first = chanceCardDeck[0];
92
93      for (int i = 0; i < chanceCardDeck.length - 1; i++) {
94        chanceCardDeck[i] = chanceCardDeck[i + 1];
95      }
96
97      chanceCardDeck[chanceCardDeck.length - 1] = first;
98      return first;
99    }
100
101   /**
102    * Method shuffle: Shuffles the chance card deck.
103    */
104   public void shuffle() {
```

```
105       for (int i = 0; i < 10000; i++) {
106         swapTwoCards();
107       }
108     }
109
110     /**
111      * Method swapTwoCards: Swaps two chance cards in the deck.
112      */
113     private void swapTwoCards() {
114       Random generator = new Random();
115       int cardNumber1 = generator.nextInt(chanceCardDeck.length);
116       ChanceCard temp = chanceCardDeck[cardNumber1];
117       int cardNumber2 = generator.nextInt(chanceCardDeck.length);
118       chanceCardDeck[cardNumber1] = chanceCardDeck[cardNumber2];
119       chanceCardDeck[cardNumber2] = temp;
120     }
121
122     /**
123      * Method creatGrant: Returns a Grant chanceCard from the given
          information.
124      *
125      * @param i
126      *             The information given.
127      * @return The Grant chanceCard.
128      */
129     private Grant createGrant(String[] i) {
130       Grant grant = new Grant(i[0], i[1], toInt(i[2]));
131       return grant;
132     }
133
134     /**
135      * Method createTaxCard: Returns a Tax chanceCard from the given
136      * information.
137      *
138      * @param i
139      *             The information given.
140      * @return The Tax chanceCard.
141      */
142     private TaxCard createTaxCard(String[] i) {
143       String[] stringArray = { i[2], i[3] };
144       TaxCard taxCard = new TaxCard(i[0], i[1], toIntArray(stringArray));
145       return taxCard;
146     }
147
148     /**
149      * Method createPayment: Returns a Payment chanceCard from the given
150      * information.
151      *
152      * @param i
153      *             The information given.
154      * @return The Payment chanceCard.
155      */
156     private Payment createPayment(String[] i) {
157       Payment payment = new Payment(i[0], i[1], toInt(i[2]));
158       return payment;
```

```
159     }
160
161     /**
162      * Method createParty: Returns a Party chanceCard from the given
163      * information.
164      *
165      * @param i
166      *             The information given.
167      * @return The Party chanceCard.
168      */
169     private Party createParty(String[] i) {
170       Party party = new Party(i[0], i[1], toInt(i[2]));
171       return party;
172     }
173
174     /**
175      * Method createPrison: Returns a Prison chanceCard from the given
176      * information.
177      *
178      * @param i
179      *             The information given.
180      * @return The Prison chanceCard.
181      */
182     private Prison createPrison(String[] i) {
183       Prison prison = new Prison(i[0], i[1]);
184       return prison;
185     }
186
187     /**
188      * Method createMoveThreeSteps: Returns a MoveThreeSteps chanceCard from
           the
189      * given information.
190      *
191      * @param i
192      *             The given information.
193      * @return The MoveThreeSteps chanceCard.
194      */
195     private MoveThreeSteps createMoveThreeSteps(String[] i) {
196       MoveThreeSteps moveThreeSteps = new MoveThreeSteps(i[0], i[1], toInt(i
              [2]));
197       return moveThreeSteps;
198     }
199
200     /**
201      * Method createMoveToField: Returns a MoveToField chanceCard from the
           given
202      * information.
203      *
204      * @param i
205      *             The given information.
206      * @return The MoveToField chanceCard.
207      */
208     private MoveToField createMoveToField(String[] i) {
209       MoveToField moveToField = new MoveToField(i[0], i[1], toInt(i[2]));
210       return moveToField;
```

```
211     }
212
213     /**
214      * Method createMoveToPrison: Returns a MoveToPrison chanceCard from the
215      * given information.
216      *
217      * @param i
218      *             The given information.
219      * @return The MoveToPrison chanceCard.
220      */
221     private MoveToPrison createMoveToPrison(String[] i) {
222       MoveToPrison moveToPrison = new MoveToPrison(i[0], i[1]);
223       return moveToPrison;
224     }
225
226     /**
227      * Method createMoveToNearestShipping: Returns a MoveToNearestShipping
228      * chanceCard from the given information.
229      *
230      * @param i
231      *             The given information.
232      * @return The MoveToNearestShipping chanceCard.
233      */
234     private MoveToNearestShipping createMoveToNearestShipping(String[] i) {
235       String[] stringArray = { i[2], i[3], i[4], i[5] };
236       MoveToNearestShipping moveToNearestShipping = new MoveToNearestShipping
          (i[0], i[1], toIntArray(stringArray),
237           toBoolean(i[6]));
238       return moveToNearestShipping;
239     }
240
241     /**
242      * Method toInt: Converts a string to an integer.
243      *
244      * @param s
245      *             The given String
246      * @return The returned integer.
247      */
248     private int toInt(String s) {
249       return Integer.parseInt(s);
250     }
251
252     /**
253      * Method toBoolean: Converts a string to a boolean.
254      *
255      * @param s
256      *             The given String.
257      * @return The returned boolean.
258      */
259     private boolean toBoolean(String s) {
260       return Boolean.parseBoolean(s);
261     }
262
263     /**
264      * Method toIntArray: Converts a String array to an integer array.
```

```
265     *
266     * @param s
267     *             The String array.
268     * @return The returned integer array.
269     */
270    private int[] toIntArray(String... s) {
271      int[] intArray = new int[s.length];
272      for (int i = 0; i < s.length; i++) {
273        intArray[i] = toInt(s[i]);
274      }
275      return intArray;
276    }
277  }
```

12.5.3.3 DiceCup

```java
package entity;

import entity.Die;

/**
 * This class is responsible for rolling 2 object of type Die, and
    returning that information to the main controller.
 * @author Gruppe33
 *
 */
public class DiceCup
{
  //Instance variables
  private Die d1;
  private Die d2;

  /**
   * Constructor: Constructs a DiceCup.
   * The diceCup object contains two die Objects.
   */
  public DiceCup()
  {
    d1 = new Die();
    d2 = new Die();
  }

  /**
   * Method rollDice: Rolls the dice in the DiceCup.
   */
  public void shake()
  {
    d1.rollDie();
    d2.rollDie();
  }

  /**
   * Method getDiceValue: returns the value of the rolled dice as an
      integer array.
   * @return Returns the array with the DiceValues.
   */
  public int[] getDiceValue()
  {
    int[] array = {d1.getValue(), d2.getValue()};
    return array;
  }
}
```

12.5.3.4 Die

```java
package entity;

/**
 * This class holds a representation of a die.
 * @author Gruppe33
 *
 */
public class Die
{
  //Constants
  public final int MAX_VALUE; //The max value of a die.

  //Instance variables
  private int value;

  /**
   * Constructor: Constructs a 6-sided die.
   */
  public Die()
  {
    MAX_VALUE = 6;
    value = rollDie();
  }

  /**
   * Constructor: Constructs a n-sided die.
   * If the parameter given is less than 1, the constructor constructs a
   * 6-sided die. If the parameter overflows, the constructor constructs a
   * 6-sided die.
   * @param sides The amount of sides you want the die to have.
   */
  public Die(int sides)
  {
    if (sides < 1)
    {
      MAX_VALUE = 6;
      value = rollDie();
    }
    else
    {
      MAX_VALUE = sides;
      value = rollDie();
    }
  }

  /**
   * Method rollDie: Sets the face value of the die to a random integer
      between 1-MAX_VALUE.
   * @return Returns the newly rolled value of the die.
   */
  public int rollDie()
  {
    value = (int) (Math.random() * MAX_VALUE + 1); // generates a random
```

```
         value between 1-MAX_VALUE.
53       return value;
54     }
55
56     /**
57      * Method getValue: Returns the face value of the die.
58      * @return Returns the current face value of the die.
59      */
60     public int getValue()
61     {
62       return value;
63     }
64
65     /**
66      * Method toString: Returns a string representation of the die.
67      */
68     public String toString()
69     {
70       return "The value of the die is: " + value;
71     }
72
73     /**
74      * Method setValue: Sets the face value of the die. NB: This method is
           only
75      * used when testing the Die class and is therefore not seen in the
76      * diagrams.
77      * @param value The face value to be set.
78      */
79     public void setValue(int value)
80     {
81       this.value = value;
82     }
83 }
```

12.5.3.5 GameBoard

```java
package entity;

import entity.field.*;
import data.Reader;

/**
 * This class is responsible for creating, holding and giving fields from the board.
 * @author Gruppe33
 *
 */
public class GameBoard
{
  // Instance variables.
  private Field[] fields;
  private int fieldCounter;

  /**
   * Constructor: Constructs a Gameboard from the data file "src/data/Feltliste.txt".
   */
  public GameBoard(){
    fields = new Field[40];
    fieldCounter = 1;
    Reader reader = new Reader("src/data/Feltliste.txt");
    String[][] fieldData = reader.readFile();
    fieldData = reader.formatFileData(fieldData);
    //Opretter felterne
    for(int i = 0; i < fields.length; i++)
    {
      addField(fieldData[i]);
    }
  }

  /**
   * Method addField: Adds a field to the GameBoard from the information given.
   * @param information The information given.
   */
  public void addField(String[] information) {
    String fieldType = information[10];
    switch (fieldType) {
      case "Ejendom":
        addStreet(information);
        break;
      case "Rederi":
        addShipping(information);
        break;
      case "Tapperi":
        addBrewery(information);
        break;
      case "Chancekort":
        addChance(information);
```

```
51            break;
52          case "Skat":
53            addTax(information);
54            break;
55          case "Start":
56          case "Besøg":
57          case "Parkering":
58          case "Fængsel":
59            addNeutral(information);
60            break;
61          default:
62            System.out.println("GameBoard: Not a valid field at field number: "
                  + fieldCounter);
63        }
64        fieldCounter++;
65      }
66
67      /**
68       * Method addStreet: Adds a field of the type Street to the GameBoard
             from the given information.
69       * @param information The information given.
70       */
71      private void addStreet(String[] information)
72      {
73        String name = information[0];
74        String type = information[10];
75        String description = information[3];
76        int price = Integer.parseInt(information[2]);
77        String colour = information[1];
78        int baseRent = Integer.parseInt(information[3]);
79        int housePrice = Integer.parseInt(information[11]);
80        int[] houseRent = { Integer.parseInt(information[4]), Integer.parseInt(
              information[5]), Integer.parseInt(information[6]), Integer.parseInt(
              information[7]), Integer.parseInt(information[8]) };
81        int pledge = Integer.parseInt(information[9]);
82        fields[fieldCounter - 1] = new Street(fieldCounter,name, type,
              description, price, colour, baseRent, housePrice, houseRent, pledge)
              ;
83      }
84
85      /**
86       * Method addShipping: Adds a field of the type Shipping to the GameBoard
             from the given information.
87       * @param information The information given.
88       */
89      private void addShipping(String[] information)
90      {
91        String name = information[0];
92        String type = information[10];
93        String description = information[3];
94        int price = Integer.parseInt(information[2]);
95        fields[fieldCounter - 1] = new Shipping(fieldCounter,name, type,
              description, price);
96      }
97
```

```
98      /**
99       * Method addBrewery: Adds a field of the type Brewery to the GameBoard
             from the given information.
100      * @param information The information given.
101      */
102     private void addBrewery(String[] information)
103     {
104       String name = information[0];
105       String type = information[10];
106       String description = information[3];
107       int price = Integer.parseInt(information[2]);
108       fields[fieldCounter - 1] = new Brewery(fieldCounter,name, type,
              description, price);
109     }
110
111     /**
112      * Method addChance: Adds a field of the type ChanceField to the
             GameBoard from the given information.
113      * @param information The given information.
114      */
115     private void addChance(String[] information)
116     {
117       String name = information[0];
118       String type = information[10];
119       String description = information[3];
120       fields[fieldCounter - 1] = new ChanceField(fieldCounter,name, type,
              description);
121     }
122
123     /**
124      * Method addTax: Adds a field of the type TaxField to the GameBoard from
             the given information.
125      * @param information The given information.
126      */
127     private void addTax(String[] information)
128     {
129       String name = information[0];
130       String type = information[10];
131       String description = "";
132       boolean rate = Boolean.parseBoolean(information[5]);
133       int amount = Integer.parseInt(information[4]);
134       fields[fieldCounter - 1] = new Tax(fieldCounter,name, type, description
              , rate, amount);
135     }
136
137     /**
138      * Method addNeutral: Adds a field of the type Neutral to the GameBoard
             from the given information.
139      * @param information The given information.
140      */
141     private void addNeutral(String[] information)
142     {
143       String name = information[0];
144       String type = information[10];
145       String description = "";
```

```
146         fields[fieldCounter - 1] = new Neutral(fieldCounter,name, description,
                type);
147     }
148 
149     /**
150      * Method getField: Returns a field from the GameBoard.
151      * @param fieldNum The number of the field you want to be returned.
152      * @return The field.
153      */
154     public Field getField(int fieldNum)
155     {
156       return fields[fieldNum-1];
157     }
158 }
```

12.5.3.6 Player

```java
package entity;

import entity.field.*;

/**
 * This class is responsible for the players, and everything they can do.
 * @author Gruppe33
 *
 */
public class Player {

  // Instance variables
  private String name; // The name of the player.
  private Account account; // The account of the player.
  private boolean hasLost; // Tells if the player has lost the game.
  private boolean inPrison; // Tells if the player is in prison.
  private int position; // The board position of the player.
  private Ownable[] fields; // The fields owned by the player
  private int turnsInPrison;

  /**
   * Constructor: Constructs a player.
   *
   * @param The
   *            name of the player.
   */
  public Player(String name) {
    this.name = name;
    account = new Account(30000);
    hasLost = false;
    inPrison = false;
    position = 1;
    fields = null;
    setTurnsInPrison(0);
  }

  /**
   * Method payRent: The object pays the rent to the owner.
   *
   * @param owner
   *            The owner to be paid.
   * @param rent
   *            The rent to be paid.
   */
  public void payRent(Player owner, int rent) {
    // Checks if the player has enough money to pay the rent.
    if (account.getBalance() >= rent) {
      // Adds the rent to the balance of the owner.
      owner.changeAccountBalance(rent);
      // Subtracts the rent from the objects balance.
      account.changeBalance(-rent);
    }
  }
```

```
54
55    /**
56     * Method getPlayerName: Returns the name of the player.
57     *
58     * @return The name of the player.
59     */
60    public String getName() {
61      return name;
62    }
63
64    /**
65     * Method getAccountBalance: Returns the balance of the player's account.
66     *
67     * @return The balance of the player's account.
68     */
69    public int getAccountBalance() {
70      return account.getBalance();
71    }
72
73    /**
74     * Method changeAccountBalance: Changes balance of the player's account
          with
75     * the parameter value.
76     *
77     * @param value
78     *             The balance should be changed with.
79     */
80    public void changeAccountBalance(int value) {
81      account.changeBalance(value);
82    }
83
84    /**
85     * Method getLost: Returns the player's lost status.
86     *
87     * @return The player's lost status. If true then the player has lost the
88     *         game.
89     */
90    public boolean getHasLost() {
91      return hasLost;
92    }
93
94    /**
95     * Method setLost: Sets the player's lost status.
96     *
97     * @param condition
98     *             The condition to be set. If condition is true then the
          player
99     *             has lost.
100    */
101   public void setHasLost(boolean condition) {
102     hasLost = condition;
103   }
104
105   /**
106    * Method getPrison: Returns the player's prison status.
```

```
107     *
108     * @return The player's prison status. If true then the player is in
            prison.
109     */
110    public boolean getInPrison() {
111      return inPrison;
112    }
113
114    /**
115     * Method setPrison: Sets the player's prison status.
116     *
117     * @param condition
118     *             The condition to be set. If condition is true then the
            player
119     *             is in prison.
120     */
121    public void setInPrison(boolean condition) {
122      inPrison = condition;
123    }
124
125    /**
126     * Method getPosition: Returns the position of the player
127     *
128     * @return The position of the player
129     */
130    public int getPosition() {
131      return position;
132    }
133
134    /**
135     * Method setPosition sets the position of the player.
136     *
137     * @param position
138     *             The position to be set.
139     */
140    public void setPosition(int position) {
141      this.position = position;
142    }
143
144    /**
145     * Method getFields: Returns the fields owned by the player.
146     *
147     * @return The fields owned by the player.
148     */
149    public Ownable[] getFields() {
150      return fields;
151    }
152
153    /**
154     * Method setFields: Sets the fields owned by the player.
155     *
156     * @param street
157     *             The street to be added to the player's street list.
158     */
159    public void setFields(Field field) {
```

```
160       Ownable ownable = (Ownable) (field);
161
162       int length;
163       if (fields == null) {
164         length = 1;
165       } else {
166         length = fields.length + 1;
167       }
168       // Creates a new fields array with 1 more space than the original.
169       Ownable[] newFields = new Ownable[length];
170
171       // Go through the original fields array and add its values to the new
172       // fields array.
173       for (int i = 0; i < newFields.length - 1; i++) {
174         newFields[i] = fields[i];
175       }
176
177       // Add the newly bought street to the new fields array.
178       newFields[newFields.length - 1] = ownable;
179
180       // Sets the original fields array to the new fields array.
181       fields = newFields;
182     }
183
184     /**
185      * Method loseField: Removes a Ownable field from the player's field list
             .
186      *
187      * @param field
188      *             The specific field to be removed.
189      * @param player
190      *             The player affected by this removal.
191      */
192     public void removeField(Ownable field) {
193       // Check that the owner of the field and this player is the same.
194       if (field.getOwner().getName().equals(this.getName())) {
195         field.setOwner(null);
196
197         String removedField = field.getName();
198
199
200         Ownable[] fewerFields = new Ownable[fields.length - 1];
201
202         if (fields.length == 1) {
203           fields = null;
204           return;
205         }
206
207         int i=0;
208         for (int j = 0; j < fields.length; j++) {
209           if (!removedField.equals(fields[j].getName())) {
210             fewerFields[i] = fields[j];
211             i++;
212           }
213         }
```

```
214
215         fields = fewerFields;
216       }
217     }
218
219     /**
220      * Method getBottlersOwned: Returns the amount of bottler fields owned by
221      * the player.
222      *
223      *
224      * @return The amount of bottler fields owned.
225      */
226     public int getBreweriesOwned() {
227       int amountOfBottlers = 0;
228       for (int i = 0; i < fields.length; i++) {
229         if ("Tapperi".equals(fields[i].getType())) {
230           amountOfBottlers++;
231         }
232       }
233       return amountOfBottlers;
234     }
235
236     /**
237      * Method getFleetsOwned: Returns the amount of fleet fields owned by the
238      * player.
239      *
240      * @return The amount of fleet fields owned.
241      */
242     public int getShippingsOwned() {
243       int amountOfShippings = 0;
244       for (int i = 0; i < fields.length; i++) {
245         if (fields[i].getType().equals("Rederi")) {
246           amountOfShippings++;
247         }
248       }
249       return amountOfShippings;
250     }
251
252     /**
253      * Method getPropertiesOwned: Returns the amount of properties owned by
           the
254      * player with the same colour.
255      *
256      * @param colour
257      *            The colour of the field.
258      * @return The amount of properties with the given colour.
259      */
260     public int getStreetsOwned(String colour) {
261       int numSameColour = 0;
262       if (fields == null) {
263         return numSameColour;
264       }
265       for (int i = 0; i < fields.length; i++) {
266         if ((fields[i]).getType().equals("Ejendom")) {
267           Street field_i = (Street) (fields[i]);
```

```
268             if ("Ejendom".equals(fields[i].getType()) && colour.equals(field_i.
                    getColour())) {
269               numSameColour++;
270             }
271           }
272         }
273         return numSameColour;
274       }
275
276       /**
277        * Method buyField: Lets the player buy a Ownable field.
278        *
279        * @param player
280        *            The player to buy the field.
281        * @return True if the buy succeeded.
282        */
283       public boolean buyField(Field field) {
284         Ownable ownable = (Ownable) (field);
285         if (getAccountBalance() > ownable.getPrice()) // Checks if the player
286                                 // has enough money to
287                                 // buy the field.
288         {
289           changeAccountBalance(-ownable.getPrice()); // Subtracts the price of
290                                 // the field from the
291                                 // player account
292                                 // balance.
293           ownable.setOwner(this); // Sets the player to be the owner of the
294                       // field.
295           this.setFields(field);
296
297           return true;
298         } else {
299           return false;
300         }
301       }
302
303       /**
304        * Method getFortune: Calculates and returns the total fortune of the
305        * player.
306        *
307        * @return The fortune of the player.
308        */
309       public int getFortune() {
310         int fortune = getAccountBalance();
311         if (fields != null) {
312           for (int i = 0; i < fields.length; i++) {
313
314             fortune = fortune + fields[i].getValue();
315             if (fields[i].getType().equals("Ejendom")) {
316               fortune = fortune + ((Street) fields[i]).getHousePrice() * ((
                      Street) fields[i]).getNumbOfHouses();
317             }
318           }
319         }
320         return fortune;
```

```
321     }
322
323     /**
324      * Method getStreetFromName: Returns the field with the given name.
325      *
326      * @param name
327      *             The name of the street that you want to find.
328      * @return The street with the name.
329      */
330     public Street getStreetFromName(String name) {
331       Street field = null;
332       for (int i = 0; i < fields.length; i++) {
333         if (name == fields[i].getName()) {
334           field = (Street) (fields[i]);
335         }
336       }
337       return field;
338     }
339
340     /**
341      * Method getHousePriceFromColour: Returns the house price of a specific
342      * coloured street.
343      *
344      * @param colour
345      *             The colour of the street that you want the house from.
346      * @return The house price of a specific coloured street.
347      */
348     public int getHousePriceFromColour(String colour) {
349       int housePrice = 0;
350       int housePriceBlue = 1000;
351       int housePriceOrange = 1000;
352       int housePriceGreen = 2000;
353       int housePriceGrey = 2000;
354       int housePriceRed = 3000;
355       int housePriceWhite = 3000;
356       int housePriceYellow = 4000;
357       int housePricePurple = 4000;
358       int[] housePrices = { housePriceBlue, housePriceOrange, housePriceGreen
              , housePriceGrey, housePriceRed,
359           housePriceWhite, housePriceYellow, housePricePurple };
360       String[] streetColours = { "Blå", "Orange", "Grøn", "Grå", "Rød", "Hvid
              ", "Gul", "Lilla" };
361
362       for (int i = 0; i < streetColours.length; i++) {
363         if (colour.equals(streetColours[i])) {
364           housePrice = housePrices[i];
365         }
366       }
367       return housePrice;
368     }
369
370     /**
371      * Method getTurnsInPrison: Returns how many turns the player has been in
372      * Prison.
373      *
```

```
374     * @return The amount of turns the player has been in the prison.
375     */
376    public int getTurnsInPrison() {
377      return turnsInPrison;
378    }
379
380    /**
381     * Method setTurnsInPrison: sets the value of the variable turnsInPrison.
382     *
383     * @param turnsInPrison
384     *             The value to be set.
385     */
386    public void setTurnsInPrison(int turnsInPrison) {
387      this.turnsInPrison = turnsInPrison;
388    }
389
390    /**
391     * Method changeTurnsInPrison: Changes the value of the variable
392     * turnInPrison.
393     *
394     * @param diffTurnsInPrison
395     */
396    public void changeTurnsInPrison(int diffTurnsInPrison) {
397      this.turnsInPrison += diffTurnsInPrison;
398    }
399
400 }
```

12.5.4 entity.chanceCard

12.5.4.1 ChanceCard

```java
package entity.chanceCard;

/**
 * This class describes a simple chance card.
 * @author Gruppe33
 *
 */
public abstract class ChanceCard {

  //Instance variables
  private String type;
  private String description;

  /**
   * Constructor: Constructs a ChanceCard
   * @param type The type of the ChanceCard
   * @param description The description of the ChanceCard
   */
  public ChanceCard(String type, String description)
  {
    this.type = type;
    this.description = description;

  }

  /**
   * Method getType(): Returns the type of the chanceCard.
   * @return The type of the chance card.
   */
  public String getType()
  {
    return type;
  }

  /**
   * Method getDescription: Returns the description of the chanceCard.
   * @return The description of the chance card.
   */
  public String getDescription()
  {
    return description;
  }
}
```

12.5.4.2 Grant

```java
package entity.chanceCard;



/**
 * This class describes the specific chanceCard Grant.
 * @author Gruppe33
 *
 */
public class Grant extends ChanceCard{

  //Instance variables
  private int amount;

  /**
   * Constructor: Constructs a Grant ChanceCard.
   * @param type The type of the ChanceCard.
   * @param description The description of the ChanceCard.
   * @param amount The amount to be granted.
   */
  public Grant(String type, String description,int amount)
  {
    super(type,description);
    this.amount = amount;
  }

  /**
   * Method getAmount: Returns the amount of the Grant.
   * @return The amount of the grant
   */
  public int getAmount()
  {
    return amount;
  }
}
```

12.5.4.3 Movement

```java
package entity.chanceCard;

/**
 * This class describes the card type movement 4 classes extends this class
    .
 * @author Gruppe33
 *
 */
public class Movement extends ChanceCard{

  /**
   * Constructor: Constructs a Movement ChanceCard.
   * @param type The type of the chanceCard.
   * @param description The description of the ChanceCard.
   */
  public Movement(String type, String description)
  {
    super(type,description);
  }
}
```

12.5.4.4 MoveThreeSteps

```java
package entity.chanceCard;

/**
 * This class describes the specific chanceCard MoveThreeSteps.
 * @author Gruppe33
 *
 */
public class MoveThreeSteps extends Movement {

  //Instance variables
  private int steps;

  /**
   * Constructor: Constructs a MoveThreeSteps Movement ChanceCard.
   * @param type The type of the Movement ChanceCard.
   * @param description The description of the Movement ChanceCard.
   * @param steps The amount of steps to move.
   */
  public MoveThreeSteps(String type, String description, int steps)
  {
    super(type, description);
    this.steps = steps;
  }

  /**
   * Method getSteps(): Returns how many steps to move.
   * @return The steps to move.
   */
  public int getSteps()
  {
    return steps;
  }
}
```

12.5.4.5 MoveToField

```java
package entity.chanceCard;


/**
 * This class describes the specific chanceCard MoveToField.
 * @author Gruppe33
 *
 */
public class MoveToField extends Movement {

  //Instance variables
  private int fieldPosition;

  /**
   * Constructor: Constructs a MoveToField Movement ChanceCard.
   * @param type The type of the Movement ChanceCard.
   * @param description The description of the Movement ChanceCard.
   * @param fieldPosition The position of the field to move to.
   */
  public MoveToField(String type, String description, int fieldPosition)
  {
    super(type, description);
    this.fieldPosition = fieldPosition;
  }

  /**
   * Method getMoveTo(): Returns the field position to move to..
   * @return The field position to move to.
   */
  public int getMoveTo()
  {
    return fieldPosition;
  }

}
```

12.5.4.6 MoveToNearestShipping

```java
package entity.chanceCard;

/**
 * This class describes the specific chanceCard MoveToNearestShipping.
 * @author Gruppe33
 *
 */
public class MoveToNearestShipping extends Movement {

  //Instance variables
  private int[] shippingPositions;
  private boolean doubleRent;

  /**
   * Constructor: Constructs a MoveToNearestShipping Movement ChanceCard.
   * @param type The type of the Movement ChanceCard.
   * @param description The description of the Movement ChanceCard.
   * @param shippingPositions The positions of all the shipping fields.
   * @param doubleRent True if you should pay double rent.
   */
  public MoveToNearestShipping(String type, String description, int[]
      shippingPositions, boolean doubleRent)
  {
    super(type, description);
    this.shippingPositions = shippingPositions;
    this.doubleRent = doubleRent;
  }

  /**
   * Method getShippingPositions: Returns the position of the shipping
      fields.
   * @return The position of the shipping fields.
   */
  public int[] getShippingPositions()
  {
    return shippingPositions;
  }

  /**
   * Method getDoubleRent: Returns if the player should pay double rent.
   * @return True if the player should pay double rent. False otherwise.
   */
  public boolean getDoubleRent()
  {
    return doubleRent;
  }
}
```

12.5.4.7 MoveToPrison

```java
package entity.chanceCard;

/**
 * This class describes the specific chanceCard MoveToPrison.
 * @author Gruppe33
 *
 */
public class MoveToPrison extends Movement {

  //Instance variables
  private boolean sentToPrison;

  /**
   * Constructor: Constructs a MoveToPrison ChanceCard.
   * @param type The type of the Movement ChanceCard.
   * @param description The description of Movement ChanceCard.
   */
  public MoveToPrison(String type, String description)
  {
    super(type, description);
    this.sentToPrison = true;
  }

  /**
   * Method getSentToPrison: Returns if the player should be sent to prison
      .
   * @return True if the player should be sent to prison and false
      otherwise.
   */
  public boolean getInPrison()
  {
    return sentToPrison;
  }
}
```

12.5.4.8 Party

```java
package entity.chanceCard;

/**
 * This class describes the specific chanceCard Party.
 * @author Gruppe33
 *
 */
public class Party extends ChanceCard{

  //Instance variables
  private int cost;

  /**
   * Constructor: Constructs a Party ChanceCard.
   * @param type The type of the ChanceCard
   * @param description The description of the ChanceCard
   * @param cost The cost of the party.
   */
  public Party(String type, String description,int cost)
  {
    super(type,description);
    this.cost = cost;
  }

  /**
   * Method getPrice: Returns the cost of the party.
   * @return The cost of the party.
   */
  public int getCost()
  {
    return cost;
  }
}
```

12.5.4.9 Payment

```
1  package entity.chanceCard;
2
3  /**
4   * This class describes the specific chanceCard Payment.
5   * @author Gruppe33
6   *
7   */
8    public class Payment extends ChanceCard{
9
10
11      //Instance variables
12      private int amount;
13
14      /**
15       * Constructor: Constructs a Payment ChanceCard.
16       * @param type The type of the ChanceCard.
17       * @param description The description of the ChanceCard.
18       * @param amount The amount of the payment.
19       */
20      public Payment(String type, String description, int amount)
21      {
22        super(type, description);
23        this.amount = amount;
24      }
25
26      /**
27       * Method getAmount: Returns the amount of the payment.
28       * @return The amount of the payment.
29       */
30      public int getAmount()
31      {
32        return amount;
33      }
34
35  }
```

12.5.4.10 Prison

```java
package entity.chanceCard;

/**
 * This class describes the specific chanceCard Prison.
 * @author Gruppe33
 *
 */
public class Prison extends ChanceCard{

  //Instance variables.
  private boolean prisonStatus;

  /**
   * Constructor: Constructs a Prison ChanceCard.
   * @param type The type of the ChanceCard.
   * @param description The description of the ChanceCard.
   */
  public Prison(String type, String description)
  {
    super(type,description);
    this.prisonStatus = false;
  }

  public boolean getPrisonStatus()
  {
    return prisonStatus;
  }
}
```

12.5.4.11 TaxCard

```java
package entity.chanceCard;

/**
 * This class describes the specific chanceCard Taxcard.
 * @author Gruppe33
 *
 */
public class TaxCard extends ChanceCard
{
  //Instance variables
  private int[] taxPrices;

  /**
   * Constructor: Constructs a TaxCard ChanceCard.
   * @param type The type of the ChanceCard.
   * @param description The description of the ChanceCard.
   * @param taxPrices The tax prices for houses and hotels.
   */
  public TaxCard(String type, String description, int[] taxPrices)
  {
    super(type,description);
    this.taxPrices = taxPrices;
  }

  public int[] getTaxPrices()
  {
    return taxPrices;
  }
}
```

12.5.5 entity.field

12.5.5.1 Brewery

```java
package entity.field;


/**
 * This class describes the ownable field type Brewery.
 * @author Gruppe33
 *
 */
public class Brewery extends Ownable{

  //Instance variables
  private int baseRent;
  private int diceSum;

  /**
   * Constructor: Constructs a brewery.
   * @param type The type of the field.
   * @param description The description of the field.
   * @param price The price of the field.
   */
  public Brewery(int fieldNumber,String name, String type, String
      description, int price)
  {
    super(fieldNumber, name, type, description, price);
    this.baseRent = 100;
  }

  /**
   * Method setDiceSum: Sets the value of diceSum.
   * @param diceSum The diceSum to be set.
   */
  public void setDiceSum(int diceSum)
  {
    this.diceSum = diceSum;
  }

  /**
   * Method getRent: Returns the rent to be paid by the player who lands a
      brewery field.
   * @return The rent to be paid.
   */
  @Override
  public int getRent()
  {
    int numbOfBreweries = super.getOwner().getBreweriesOwned();
    int rent = baseRent * diceSum * numbOfBreweries;
    return rent;
  }
}
```

12.5.5.2 ChanceField

```java
package entity.field;


/**
 * This class describes the field type Chancefield.
 * @author Gruppe33
 *
 */
public class ChanceField extends Field{

  /**
   * Constructor: Constructs a ChanceField.
   * @param fieldNumber The number of the field.
   * @param name The name of the field.
   * @param type The type of the field.
   * @param description The description of the field.
   */
  public ChanceField(int fieldNumber, String name, String type, String
      description)
  {
    super(fieldNumber, name, type, description);
  }

  /**
   * Method getRent: This method is not used but makes other classes
       simpler.
   */
  @Override
  public int getRent()
  {
    return -1;
  }
}
```

12.5.5.3 Field

```java
package entity.field;

/**
 * This class is the general description of a field.
 * @author Gruppe33
 *
 */
public abstract class Field {

  //Instance variables
  private int fieldNumber;
  private String name;
  private String type;
  private String description;

  /**
   * Abstract constructor: Constructs a field.
   * @param The type of the field.
   * @param The description of the field.
   */
  public Field(int fieldNumber, String name, String type, String
      description)
  {
    this.fieldNumber = fieldNumber;
    this.name = name;
    this.type = type;
    this.description = description;
  }

  /**
   * Method getType: Returns the type of the field.
   * @return The type of the field.
   */
  public String getType()
  {
    return type;
  }

  /**
   * Method getDescription: Returns the description of the field.
   * @return The description of the field.
   */
  public String getDescription()
  {
    return description;
  }

  /**
   * Method getName: Returns the name of the field.
   * @return The name of the field.
   */
  public String getName()
  {
```

```
53        return name;
54    }
55
56    /**
57     * Method getFieldNumber: Returns the number of the field.
58     * @return The number of the field.
59     */
60    public int getFieldNumber()
61    {
62        return fieldNumber;
63    }
64
65    /**
66     * The method getRent: Returns the rent to be paid by the player who
          lands on an Ownable field. <br>
67     * A method to be overridden by subclasses.
68     * @return The rent to be paid.
69     */
70    public abstract int getRent();
71
72 }
```

12.5.5.4 Neutral

```java
package entity.field;


/**
 * This class describes the field type Neutral. This includes the prison
    fields, start field and parking field.
 * @author Gruppe33
 *
 */
public class Neutral extends Field
{

  /**
   * Constructor: Constructs a Neutral field.
   * @param fieldNumber The number of the field.
   * @param name The name of the field.
   * @param type The type of the field.
   * @param description The description of the field.
   */
  public Neutral (int fieldNumber, String name, String type, String
      description){
    super(fieldNumber, name, type, description);
  }

  /**
   * Method getRent: This method is not used but makes other classes
      simpler.
   */
  @Override
  public int getRent()
  {
  return -1;
  }
}
```

12.5.5.5 Ownable

```java
package entity.field;

import entity.Player;
/**
 * This class is a general description of fields that can be owned by
    players.
 * @author Gruppe33
 *
 */
public abstract class Ownable extends Field {

  //Instance variables
  private int price;
  private Player owner;

  /**
   * Abstract constructor: Constructs an ownable field.
   * @param type The type of the field.
   * @param description The description of the field.
   * @param price The price of the field.
   */
  public Ownable(int fieldNumber, String name, String type, String
      description, int price)
  {
    super(fieldNumber, name, type, description);
    this.price = price;
    this.owner = null;
  }



  /**
   * Method getOwner: Returns the owner of the field.
   * @return The player who owns the field.
   */
  public Player getOwner()
  {
    return owner;
  }

  /**
   * Method getPrice: Returns the price of the field.
   * @return The price of the field.
   */
  public int getPrice()
  {
    return price;
  }

  /**
   * Method removeOwner: Removes the current owner of the field.
   */
  public void removeOwner()
```

```
52    {
53      owner = null;
54    }
55
56    /**
57     * Method setOwner: Sets the owner of the field to player.
58     * @param player The player to be set to own the field.
59     */
60    public void setOwner(Player player)
61    {
62      owner = player;
63    }
64
65    /**
66     * Method isFieldOwned: Checks if the field is owned by a player.
67     * @return True if the field is owned.
68     */
69    public boolean isFieldOwned()
70    {
71      if (owner == null) //Checks if the field has a owner.
72      {
73        return false;
74      }
75      else
76      {
77        return true;
78      }
79    }
80
81    /**
82     * Method isFieldOwnedByAnotherPlayer: Checks if the field is owned by
          another player.
83     * @param player The player who landed on the field.
84     * @return true if the field is owned by another player, otherwise it
          returns false.
85     */
86    public boolean isFieldOwnedByAnotherPlayer(Player player)
87    {
88      if(isFieldOwned())
89      {
90        if (owner.getName().equals(player.getName()))
91        {
92          return false;
93        }
94        else
95        {
96          return true;
97        }
98      }
99      else
100     {
101       return false;
102     }
103   }
104
```

```
105     /**
106      * Method getRent: Returns the rent to be paid by the player who lands on
             the Ownable field. <br>
107      * A method to be overridden by subclasses.
108      * @return The rent to be paid.
109      */
110     public abstract int getRent();
111 
112     /**
113      * Method getValue: Returns the value of field (The price of the field).
114      * @return The value of the field.
115      */
116     public int getValue()
117     {
118       return getPrice();
119     }
120 
121     /**
122      * Method getColour: This method is not used but makes other classes
             simpler.
123      * @return null
124      */
125     public String getColour()
126     {
127       return "Empty";
128     }
129 }
```

12.5.5.6 Shipping

```java
package entity.field;

/**
 * This class describes the ownable field type Shipping.
 * @author Gruppe33
 *
 */
public class Shipping extends Ownable {

  //Constants
  final private int[] RENT = {500, 1000, 2000, 4000};

  /**
   * Constructor: Constructs a fleet.
   * @param type The type of the field.
   * @param description The description of the field.
   * @param price The price of the field.
   */
  public Shipping(int fieldNumber, String name, String type, String
      description, int price)
  {
    super(fieldNumber, name, type, description, price);
  }

  /**
   * Method getRent: Returns the rent to be paid by the player who lands on
       a fleet field.
   * @return The rent to be paid.
   */
  @Override
  public int getRent()
  {
    int numbOfShippings = super.getOwner().getShippingsOwned(); //The
        amount of fleet fields the owner of the fleet field owns.
    int rent;
    switch(numbOfShippings)
    {
    case 1: rent = RENT[0]; //The rent to be paid if the owner owns one
        fleet field.
      break;
    case 2: rent = RENT[1]; //The rent to be paid if the owner owns two
        fleet fields.
      break;
    case 3: rent = RENT[2]; //The rent to be paid if the owner owns three
        fleet fields.
      break;
    case 4: rent = RENT[3]; //The rent to be paid if the owner owns all
        fleet fields.
      break;
    default: rent = 0;
    }
    return rent;
  }
```

```
47 || }
```

12.5.5.7 Street

```java
package entity.field;

/**
 * This class describes the ownable field type Street.
 * @author Gruppe33
 *
 */
public class Street extends Ownable {
  // Instance variables
  private String colour;
  private int baseRent;
  private int housePrice;
  private int[] houseRent;
  private int numbOfHouses;
  private int pledge;

  /**
   * constructor: Constructs a Street.
   *
   * @param name The name of the field.
   * @param type The type of the field.
   * @param description The description of the field.
   * @param price The price of the field.
   * @param colour The specific colour of the field on the board.
   * @param baseRent The base rent of the field.
   * @param houseRent The rent based on the number of houses on the field.
   * @param numbOfHouses The current number of houses built on the field.
   * @param pledge The value of a pledged field.
   */

  public Street(int fieldNumber,String name, String type, String
      description, int price, String colour, int baseRent, int housePrice,
       int[] houseRent, int pledge)
  {
    super(fieldNumber,name, type, description, price);
    this.colour = colour;
    this.baseRent = baseRent;
    this.housePrice = housePrice;
    this.houseRent = houseRent;
    this.numbOfHouses = 0;
    this.pledge = pledge;
  }

  /**
   * Method getColour: Returns the colour of the field.
   * @return The colour of the field.
   */
  @Override
  public String getColour() {
    return colour;
  }

  /**
```

```
53      * Method getBaseRent: Returns the rent to be paid.
54      * @return The base rent of the field.
55      */
56     public int getBaseRent() {
57       return baseRent;
58     }
59
60     /**
61      * Method getHouseRent: Returns the rent to be paid by a player landing
            on a
62      * built field.
63      * @return The rent based on the number of houses on the field.
64      */
65     public int getHouseRent(int numbOfHouses) {
66       return houseRent[numbOfHouses];
67     }
68
69     /**
70      * Method getRent: Calculates and returns the rent to be paid, depending
            on
71      * the number of houses.
72      * @return The rent to be paid on the field.
73      */
74     public int getRent() {
75       int rent = 0;
76
77       if (numbOfHouses == 0) {
78         rent = baseRent;
79
80         int streetsNeeded;
81         switch (colour) {
82         case "Blå":
83         case "Lilla":
84           streetsNeeded=2;
85           break;
86         default:
87           streetsNeeded=3;
88           break;
89         }
90         if(super.getOwner().getStreetsOwned(colour)==streetsNeeded){
91           rent=rent*2;
92         }
93
94       }
95       else
96       {
97         rent = houseRent[numbOfHouses-1];
98       }
99
100      return rent;
101    }
102
103    /**
104     * Method getPledge: Returns the value of the pledge given to the player,
            by
```

```
105       * the bank, when the player pledges the field.
106       */
107     public int getPledge() {
108       return pledge;
109     }
110
111     /**
112       * Method getValue: Returns the value of field (field price + prices for
113       * houses). <br>
114       *
115       * @return The value of the field.
116       */
117     @Override
118     public int getValue() {
119       return super.getValue() + numbOfHouses * housePrice;
120     }
121
122     /**
123       * Method getHousePrice: Returns the price of a house on the field.
124       * @return The price of a house on the field.
125       */
126     public int getHousePrice() {
127       return housePrice;
128     }
129
130     /**
131       * Method getNumbOfHouses: Returns the number of houses on the street.
132       * @return The number of the houses on the street.
133       */
134     public int getNumbOfHouses() {
135       return numbOfHouses;
136     }
137
138     /**
139       * Method changeOfNumbOfHouses: Change the numbOfhouses variable with the
              given amount.
140       * @param amount The amount to change the variable with.
141       */
142     public int changeNumbOfHouses(int amount)
143     {
144       numbOfHouses = numbOfHouses + amount;
145       return numbOfHouses;
146     }
147 }
```

12.5.5.8 Tax

```java
package entity.field;

/**
 * This class describes the field type Tax.
 * @author Gruppe33
 *
 */
public class Tax extends Field {

  //Instance variables
  private boolean hasTaxRate;
  private int amount;

  /**
   * Constructor: Constructs Tax field.
   * @param type The type of the field.
   * @param amount The fixed tax amount the player can choose to pay.
   * @param hasTaxRate If hasTaxRate is true the player can choose to pay
       10% of his fortune.
   */
  public Tax (int fieldNumber, String name, String type, String description
      , boolean hasTaxRate, int amount)
  {
    super(fieldNumber, name, type, description);
    this.amount = amount;
    this.hasTaxRate = hasTaxRate;
  }

  /**
   * Method getAmount: Returns the amount to be paid by the player who
       lands on the tax field.
   * @return Returns the tax amount to be paid.
   */
  public int getAmount()
  {
    return amount;
  }

  /**
   * Method getRate: Returns if the player can choose to pay 10% taxRate.
   * @return Returns True if the player can choose to pay 10% and false
       otherwise.
   */
  public boolean getHasTaxRate()
  {
    return hasTaxRate;
  }

  /**
   * Method getRent: This method is not used but makes other classes
       simpler.
   */
  @Override
```

```
49    public int getRent(){
50      return -1;
51    }
52  }
```

12.5.6 testModeController

12.5.6.1 TestModeController

```
1  package testModeController;
2
3
4  import entity.field.*;
5  import controller.FieldController;
6  import controller.MainController;
7  import desktop_resources.GUI;
8  import entity.GameBoard;
9  import entity.Player;
10
11 /**
12  * This class is a controller that interferes with normal game play.  This
        allows testers to test the program mroe easily.
13  * @author Gruppe33
14  *
15  */
16 public class TestModeController {
17   private boolean testModeStatus;
18   private String testingModeMessage = "*****DU ER I TESTING MODE*****\n";
19
20   /**
21    * Constructor: Constructs a TestModeController.
22    *
23    * @param testmodeOn
24    */
25   public TestModeController(boolean testmodeOn) {
26     if (testmodeOn) {
27       activateTestMode();
28     } else {
29       endTestMode();
30     }
31   }
32
33   /**
34    * Method activateTestMode: setsTestModeStatus to true. This will start
35    * testmode in the game.
36    */
37   private void activateTestMode() {
38     testModeStatus = true;
39     System.out.println("Test mode er aktiveret.");
40   }
41
42   /**
43    * Method endTestMode: setsTestModeStatus to false. This will end
          testmode
44    * in the game.
45    */
46   private void endTestMode() {
47     testModeStatus = false;
48   }
49
```

```
50    /**
51     * Method setPlayerOnField: Moves the player to a specific field given a
52     * field number by the player.
53     *
54     * @param player
55     *            The player whos turn it is.
56     * @return The amount of fields the player has to move forward to reach
          the
57     *         field.
58     */
59    private int setPlayerOnField(Player player) {
60      int newFieldPos = GUI.getUserInteger(
61          testingModeMessage + "Hvilket felt vil du rykke spilleren til?\n"
62              + "(Bemærk at spillere bevæger sig rundt på pladen med denne
                    funktion. Så han får start penge.)",
63          1, 40);
64      //Find the difference in position.
65      int diff = newFieldPos - player.getPosition();
66      //If the difference is less than 0, add the extra value he has to walk.
67      if (diff < 0) {
68        diff = (40 - player.getPosition()) + player.getPosition() + diff;
69      }
70
71      //Manipulate some GUI output
72      GUI.removeAllCars(player.getName());
73      int printpos = player.getPosition() + diff;
74      if (printpos > 40) {
75        printpos -= 40;
76      }
77      GUI.setCar(printpos, player.getName());
78
79      return diff;
80    }
81
82    /**
83     * Method givePlayerExtraTurn: Forces the player to receive an extra turn
          .
84     * Also counts as an extra turn when going to jail on your 3rd role.
85     *
86     * @param main
87     *            The maincontroller.
88     */
89    private void givePlayerExtraTurn(MainController main) {
90      GUI.getUserButtonPressed(testingModeMessage + "Bemærk at spilleren
          stadig sendes i fængsel efter 3 gange 2 ens",
91          "OK");
92      main.TESTsetExtraTurn(true);
93    }
94
95    /**
96     * Method setPlayerBalance: Sets the players balance to a new value.
97     *
98     * @param player
99     *            The player whos balance needs to be changed.
100    */
```

```
101   private void setPlayerBalance(Player player) {
102     int newBalance = GUI.getUserInteger(testingModeMessage + "Hvad skal den
          nye balance være?", 0, 10000000);
103     //Change the players balance.
104     player.changeAccountBalance(newBalance - player.getAccountBalance());
105     GUI.setBalance(player.getName(), player.getAccountBalance());
106   }
107
108   /**
109    * Method claimField: Claims a field for the player. Now he owns that
          field
110    * number.
111    *
112    * @param board
113    *            The boardGame in question.
114    * @param player
115    *            The players who claims the field.
116    */
117   private void claimField(GameBoard board, Player player) {
118     int fieldNum = GUI.getUserInteger("Hvilket felt vil du overtage?", 1,
          40);
119     //Claim the field.
120     changeFieldOwnership(board, player, fieldNum);
121
122   }
123
124   /**
125    * Method changeField: Changes the ownership of a field to a specific
126    * player.
127    *
128    * @param board
129    *            The boardGame in question.
130    * @param player
131    *            The players who claims the field.
132    * @param fieldNum
133    *            The fieldnumber of the field to be changed.
134    */
135   private void changeFieldOwnership(GameBoard board, Player player, int
          fieldNum) {
136     Field testField = board.getField(fieldNum);
137     //If the field can be owned.
138     if ((testField instanceof entity.field.Ownable)) {
139       Ownable currentField = (Ownable) testField;
140       //if the field is owned.
141       if (currentField.getOwner() != null) {
142         currentField.getOwner().removeField(currentField);
143         currentField.removeOwner();
144       }
145       //Buy the field, and give the player what i payed for the field.
146       player.changeAccountBalance(currentField.getPrice());
147       player.buyField(currentField);
148       GUI.setOwner(fieldNum, currentField.getOwner().getName());
149
150     } else {
151       GUI.getUserButtonPressed("Dette felt kan ikke købes", "Ok");
```

```
152         }
153     }
154 
155     /**
156      * Method claimColor: Changes the ownership of all fields of a specific
157      * color to the player.
158      *
159      * @param board
160      *             The boardGame in question.
161      * @param player
162      *             The players who claims the field.
163      */
164     private void claimColor(GameBoard board, Player player) {
165       String[] streetColours = { "Blå", "Orange", "Grøn", "Grå", "Rød", "Hvid
              ", "Gul", "Lilla" };
166       String color = GUI.getUserSelection("Hvilken farve vil du købe?",
              streetColours);
167 
168       //Check all the fields, if they have the stated color change ownership
              to the player.
169       for (int i = 0; i < 40; i++) {
170         Field field = board.getField(i + 1);
171         if (field instanceof entity.field.Ownable) {
172           if (((Ownable) field).getColour().equals(color)) {
173             changeFieldOwnership(board, player, field.getFieldNumber());
174 
175           }
176         }
177       }
178 
179     }
180 
181     /**
182      * Method options: A menu that contains the all the options the player
              has
183      * when testmode is active.
184      *
185      * @param player
186      *             The player whos turn it is.
187      * @param main
188      *             The main controller
189      * @param fieldController
190      *             The fieldController.
191      * @return The dicesum output if it has changed. Otherwise -1.
192      */
193     public int options(Player player, MainController main, FieldController
            fieldController) {
194       int output = -1;
195 
196       final String MOVE_PLAYER_TO_FIELD = "Ryk til et felt.";
197       final String EXTRA_TURN = "Giv en ekstra tur.";
198       final String SET_PLAYER_BALANCE = "Ændre balance.";
199       final String CLAIM_FIELD = "Giv en grund.";
200       final String CLAIM_COLOR = "Giv en farve.";
201       final String DEACTIVATE_TEST_MODE = "Deaktiver testmode.";
```

```
202     final String STOP_TEST_MODE = "Fortsæt spil.";
203
204     String input;
205     boolean menuActive = true;
206     if (player.getInPrison()) {
207       input = GUI.getUserSelection(testingModeMessage + "Hvad vil du gøre?"
            , "Slip ud af fængslet.",
208          "FortsætSpil");
209       if (input.equals("Slip ud af fængslet.")) {
210         main.TESTsetExtraTurn(true);
211         player.setInPrison(false);
212       }
213     } else
214       do {
215         input = GUI.getUserSelection(testingModeMessage + "Hvad vil du gøre
              ?", MOVE_PLAYER_TO_FIELD, EXTRA_TURN,
216             SET_PLAYER_BALANCE, CLAIM_FIELD, CLAIM_COLOR, STOP_TEST_MODE,
                  DEACTIVATE_TEST_MODE);
217         switch (input) {
218         case MOVE_PLAYER_TO_FIELD:
219           output = setPlayerOnField(player);
220           break;
221         case EXTRA_TURN:
222           givePlayerExtraTurn(main);
223           break;
224         case SET_PLAYER_BALANCE:
225           setPlayerBalance(player);
226           break;
227         case DEACTIVATE_TEST_MODE:
228           //Deactivate the test mode.
229           if (GUI.getUserLeftButtonPressed(
230               testingModeMessage + "Er du sikker? Du kan ikke gå tilbage
                    til testmode i denne session?",
231               "Ja", "Nej")) {
232             endTestMode();
233             menuActive = false;
234           }
235           break;
236         case STOP_TEST_MODE:
237           menuActive = false;
238           break;
239         case CLAIM_FIELD:
240           claimField(fieldController.TESTgetGameBoard(), player);
241           break;
242         case CLAIM_COLOR:
243           claimColor(fieldController.TESTgetGameBoard(), player);
244           break;
245
246
247         }
248       } while (menuActive);
249
250     return output;
251   }
252
```

```
253     /**
254      * Method isActive: Returns wheter the testmode is active or not.
255      *
256      * @return boolean that contain the status of the testmode.
257      */
258     public boolean isActive() {
259       return testModeStatus;
260     }
261
262 }
```