Projektopgave efterår 2013 – jan 2014
02312-14 Indledende programmering og 02313 Udviklingsmetoder til IT-Systemer.
Projektnavn: CDIO delopgave 2
Gruppe nr: 51.
Afleveringsfrist: *mandag den 11/11 2013 Kl. 5:00*
Denne rapport er afleveret via Campusnet (der skrives ikke under)
Denne rapport indeholder *21* sider inkl. denne side.

s123064, Nielsen, Martin _____

s130045, Kheder, Mustafa _____

s103185, Sløgedal, Magnus B. _____

s134000, Budtz, Christian _____

s133984, Freudendahl, Jens-Ulrik _____

s134004, Vørmadal, Rúni Egholm _____

Indholdsfortegnelse

Timeregnskab

51_CDIO_Del2.							
Time-regnskab	Ver. 2013-09-07						
Dato	Deltager	Design	Impl.	Test	Dok.	Andet	Ialt
							0
22/10/2013	Christian	2					2
22/10/2013	Christian		2	0.5			2.5
22/10/2013	Rúni	2					2
22/10/2013	Magnus	2	2				4
22/10/2013	Martin	2					2
23/10/2013	Magnus		1				1
23/10/2013	Mustafa		1				1
24/10/2013	Christian		1	0.5			1.5
25/10/2013	Rúni	1	3				4
25/10/2013	Christian				0.5		0.5
25/10/2013	Jens-Ulrik	0.5	0.5		0.5		1.5
29/10/2013	Magnus		4				4
31/10/2013	Rúni		1				1
1/11/2013	Runi		1				1
1/11/2013	Martin	1					1
1/11/2013	Christian	1					1
1/11/2013	Jens-Ulrik	1.5			0.5		2
1/11/2013	Magnus	1					1
5/11/2013	Rúni	1	0.5	1	0.5		3
5/11/2013	Martin		5				5
5/11/2013	Christian		0.5				0.5
5/11/2013	Magnus		1				1
6/11/2013	Christian		1	0.5			1.5
6/11/2013	Magnus	1	6	1			8
7/11/2013	Christian		1				1
7/11/2013	Magnus		0.5				0.5
8/11/2013	Christian		1		3		4
8/11/2013	Magnus		1		3		4
8/11/2013	Rúni		0.5	0.5	1		2
8/11/2013	Mustafa		1	0.5	0.5		2
9/11/2013	Christian		1		1	0.5	2.5
9/11/2013	Magnus				2		2
9/11/2013	Rúni	3		0.5	2		5.5
10/11/2013	Rúni	2			1		3
10/11/2013	Christian				0.5	0.5	1
10/11/2013	Magnus				3		3
10/11/2013	Jens-Ulrik	1.5			1		2.5
							0
							0
							0
							0
							0
							0
	Sum	20.5	36.5	5	20	1	83
	Magnus	4	15.5	1	8	0	28.5
	Christian	3	7.5	1.5	5	1	18
	Rúni	9	6	2	4.5	0	21.5
	Martin	3	5	0	0	0	8
	Jens-Ulrik	3.5	0.5	0	2	0	6
	Mustafa	0	2	0.5	0.5	0	3

Indledning

I denne rapport udvikle vi et nyt terningspil ud fra en analyse af kundens vision, vi beskriver et design heraf der opfylder GRASP principperne samt vores implementation af dette. Derudover tester vi programmet.
Vi beskriver og tester også en test klasse, der skal sandsynliggøre at Account ikke kan blive negativ.

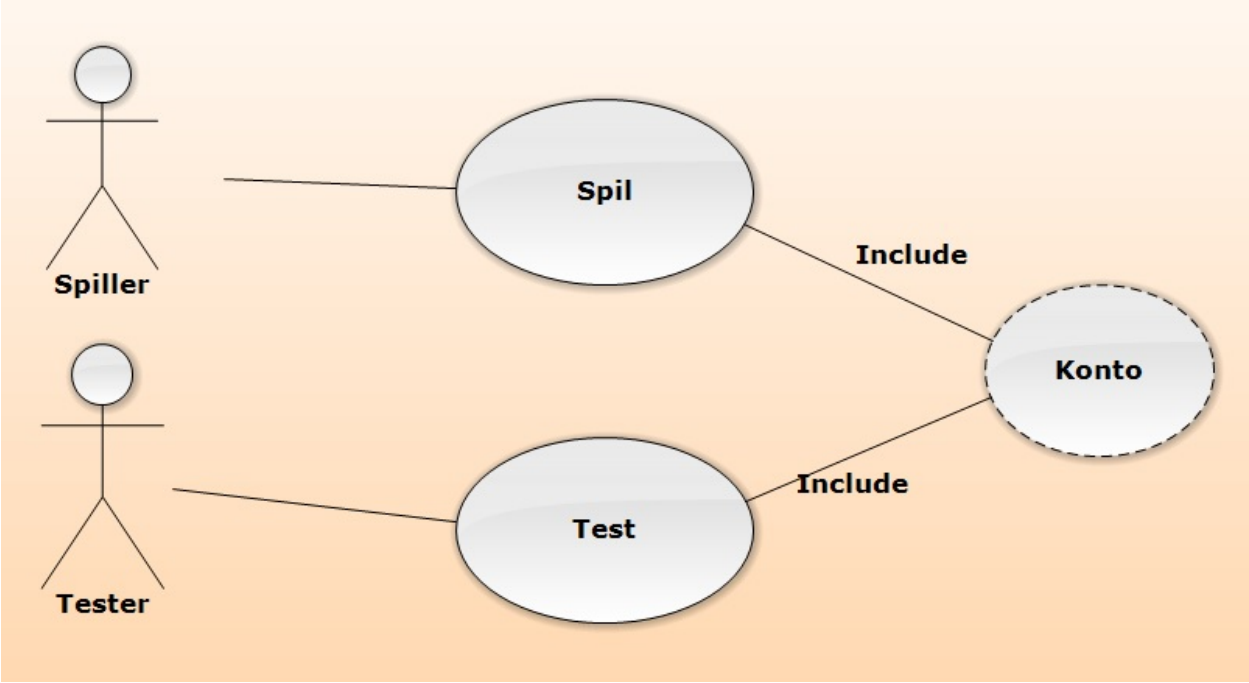
Analyse

Aktørbeskrivelser

Aktør 1: Spiller
Spillerne spiller spillet. Spillerne kan indtaste et navn. Spillerne kan vælge sprog. Spillerne kan spille spillet, indtil en vinder er fundet og evt. vælge at spille igen.

Aktør 2: Tester
Testeren afprøver kontoen og kan sandsynliggøre at den ikke kan gå i nul.

Use-case modeller



Use case diagram: *Viser vores to main use cases. Use case 'Spil spillet', beskriver selve spillets opbygning. Use case 'Test konto', beskriver test udgaven.*

Use-case beskrivelser

Use case ‘Spil spillet’

1. To spillere starter spillet.
    a. Spillerne vælger sprog.
2. Spillerne vælger spillernavne.
3. Spillerne slår efter tur med terningerne og rykker til et felt der svarer til summen af terningernes øjne.
4. Alt efter hvilket felt spilleren lander på, enten får eller mister spillerne penge fra deres pengebeholdning. Spilleren får eventuelt en ekstra tur.
5. Man vinder når en spiller har 3000 point eller når modstanderen går fallit. Hvis de begge har opnået 3000 i samme runde, så er det den spiller med flest point der vinder.
6. Spillet afsluttes når spillet har fundet en vinder eller en spiller går fallit.

Use case ‘Test konto’

1. Kontoen testes.
2. Det sandsynliggøres at balancen aldrig kan blive negativ.

Yderligere (non-funktionelle) krav

Ud over de beskrevne funktionelle krav i vores use-case, så har vi disse non-funktionelle krav:

- Skal kunne spilles uden bemærkelsesværdige forsinkelser
- Spillet skal kunne spilles på maskinerne i DTU’s databarer
- Spillerne og deres pengebeholdninger skal kunne bruges i andre spil
- Spillet skal let kunne oversættes til andre sprog
- Det skal være let at skifte til andre terninger
- Benyt de terninger vi lavede i CDIO-opgave 1
- Overhold GRASP principperne (Creator, Controller, Høj binding, Information Expert, Lav kobling)
- Alle tekster der bliver udskrevet skal holdes i boundary-klasserne.
- Koden til Spiller og pengebeholdning skal helst kunne genbruges.

Afklaring og antagelser

Ud fra oplægget noterer vi os følgende:

- Vi antager, at der i spillet bliver brugt normale 6-sidede terninger
- For at spillerne kan lande på et felt mellem numrene 2 og 12, så går vi ud fra at værdien af hver terning bliver summeret sammen.
- Ud fra brugerens eksempel på en tekst der kan udskrives når man lander på et felt, så antager vi at der skal være en sammenhæng imellem navnet på feltet og teksten der bliver udskrevet.

- I stedet for at spillet kun ”slutter” når en spiller har 3000, så går vi ud fra at systemet påpeger en vinder inden spillet slutter.
- Efter en diskussion med kunden, så antager vi at spillet skal være fair for begge spillere. Derfor får spiller to lov til at færdiggøre runden selvom spiller et allerede har opnået point på 3000 eller over. Vinderen af de to spillere bliver den person der opnår det højeste pointtal. Det bliver først uafgjort hvis begge spillere begge opnår de samme antal point i samme runde de begge opnår 3000 point eller over.
- Det fremgår ikke hvad en bemærkelsesværdig forsinkelse er.

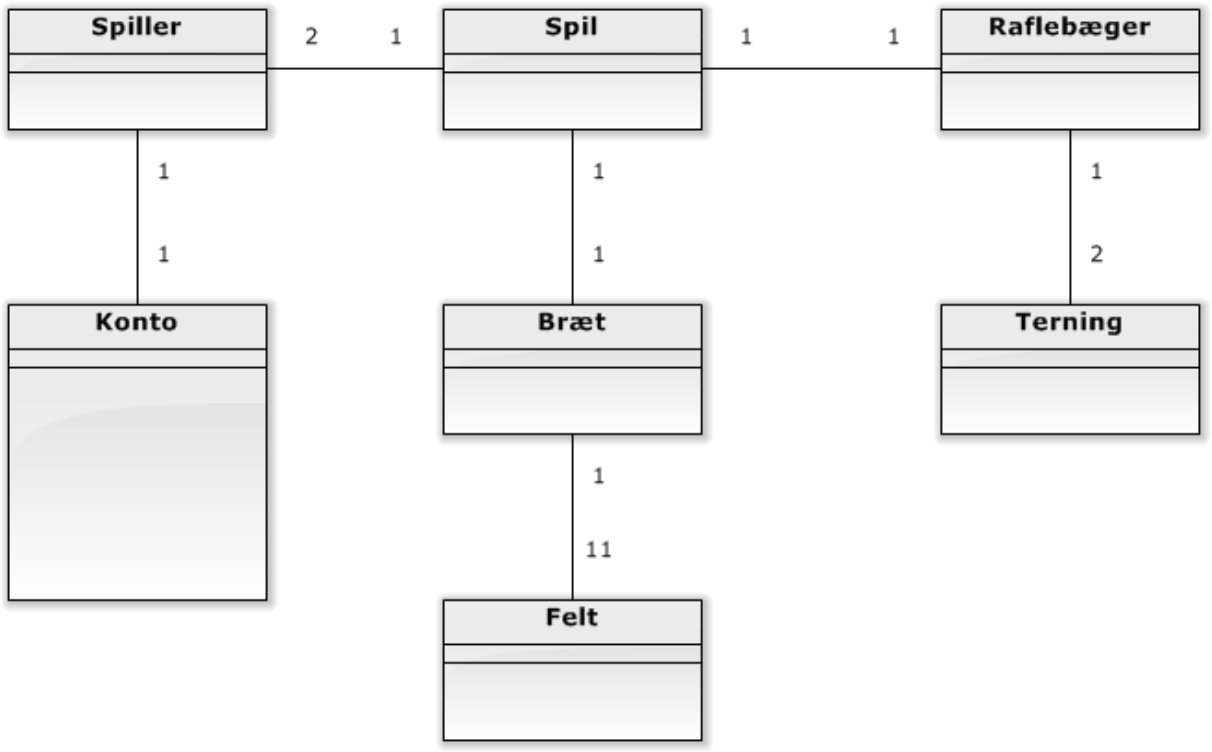
Navne- (og udsagnsords) analyse

Vi identificerer følgende begreber som signifikante fra vores use cases ‘Spil spillet’ og ‘Test konto’:

- Spil
- Spillere (med spillernavne)
- Terninger
- Pengebeholdning
- Sprog
- Vælger
- Slår
- Felt
- Penge
- 3000
- Point
- Vinder
- Test
- Konto
- Sandsynliggøre
- Negativ
- Fallit

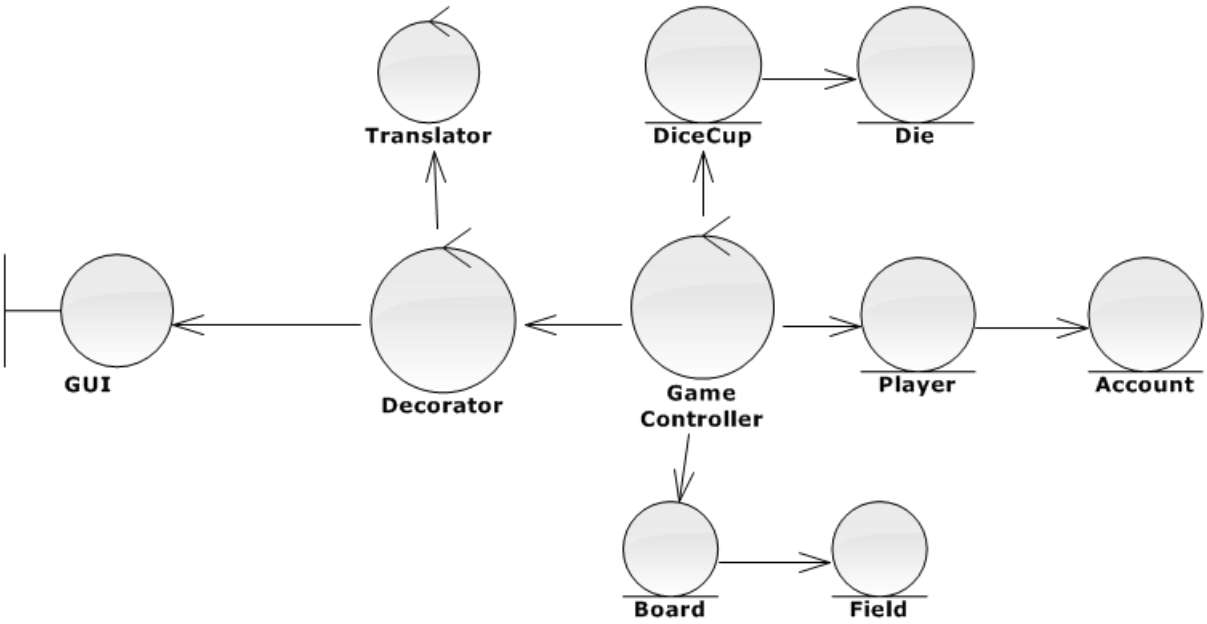
Domænemodel

Vi har i DOM’en valgt en abstraktion med et fysisk spil (‘spilleæsken’) der indeholder de andre komponenter - incl. spillebrikker (‘spiller’).



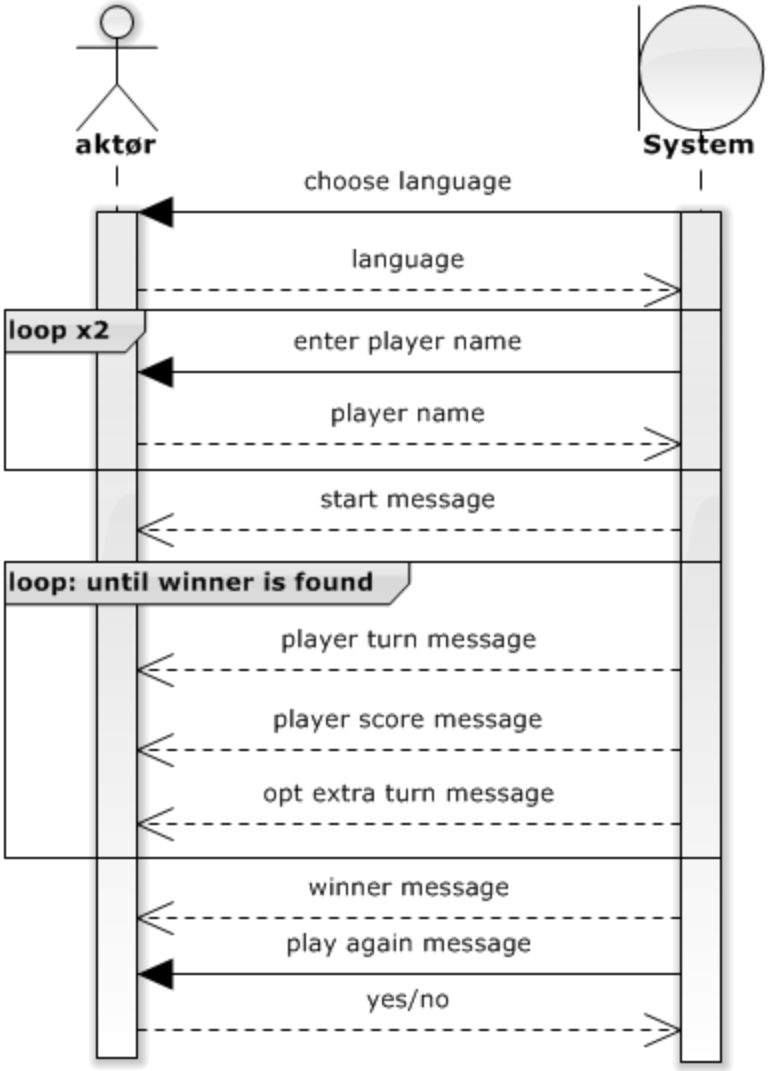
DOM. *Viser sammenhængen mellem delene i terningespillet. Spillet indeholder et raflebæger med to terninger, et bræt med felter og 2 spillere der har en konto hver.*

BCE-diagram



BCE diagram: *viser hvilke klasser er controllere og hvilke entities spillet har, samt boundary som er GUI'en. Der er tre controllere med i spillet. GameControlleren indeholder al spillelogik og har forbindelse til de entities spillet har. Translatoren er controller fordi den kun behandler data, og ikke indeholder nogen data selv. Dataen kommer fra en tekstfil ved siden af. Decoratoren indeholder nogen metoder som gør at de oplysninger som kommer fra translator og gamecontroller er forståelige for GUI'en.*

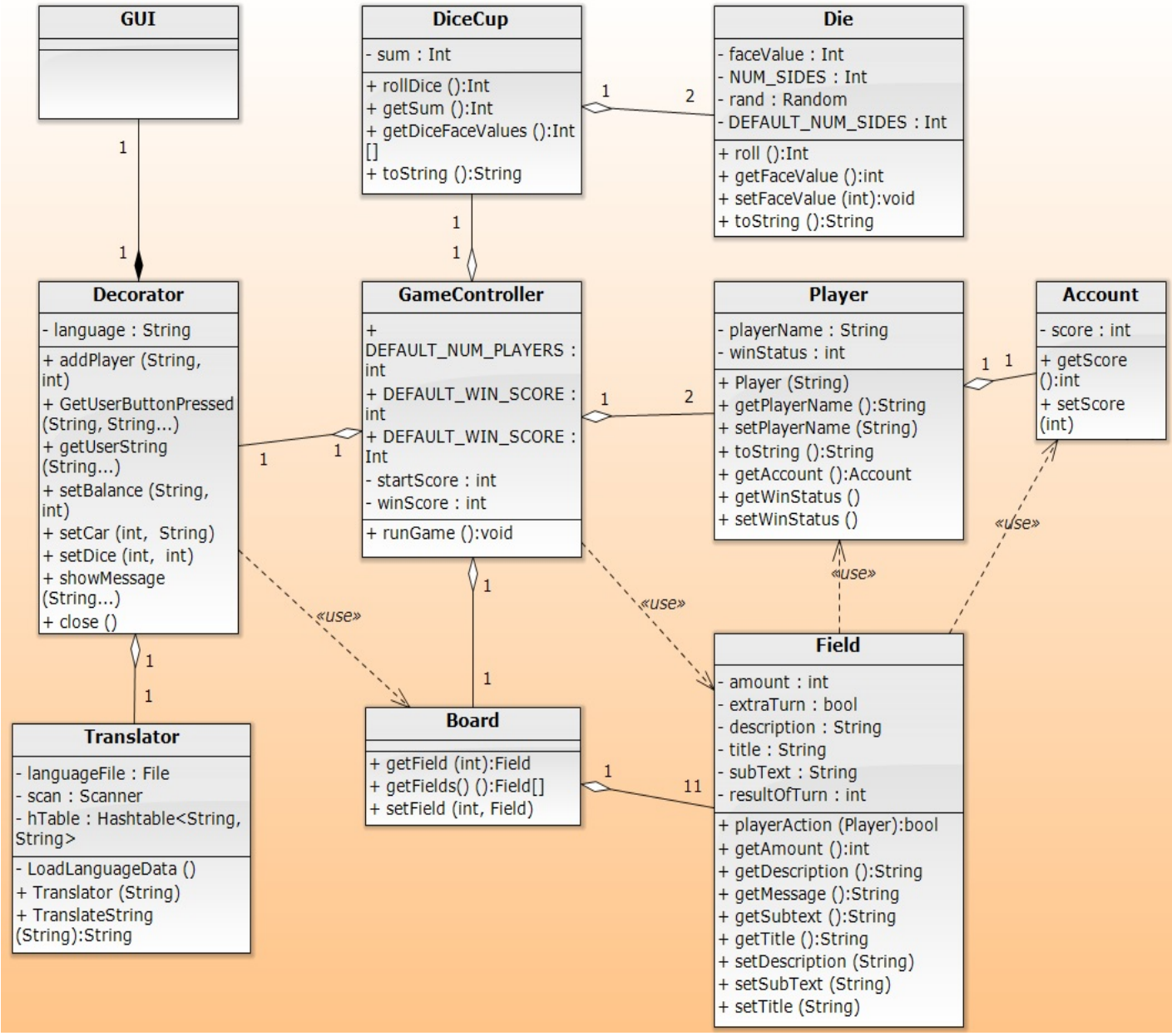
System sekvens diagrammer (Use-cases “spil” og “test”)



System sekvens diagram: *Viser hvad systemet skal gøre og hvad aktøren(spilleren) skal gøre. Der skal lægges mærke til at for hver besked systemet giver til aktøren skal aktøren trykke OK for at gå videre.*

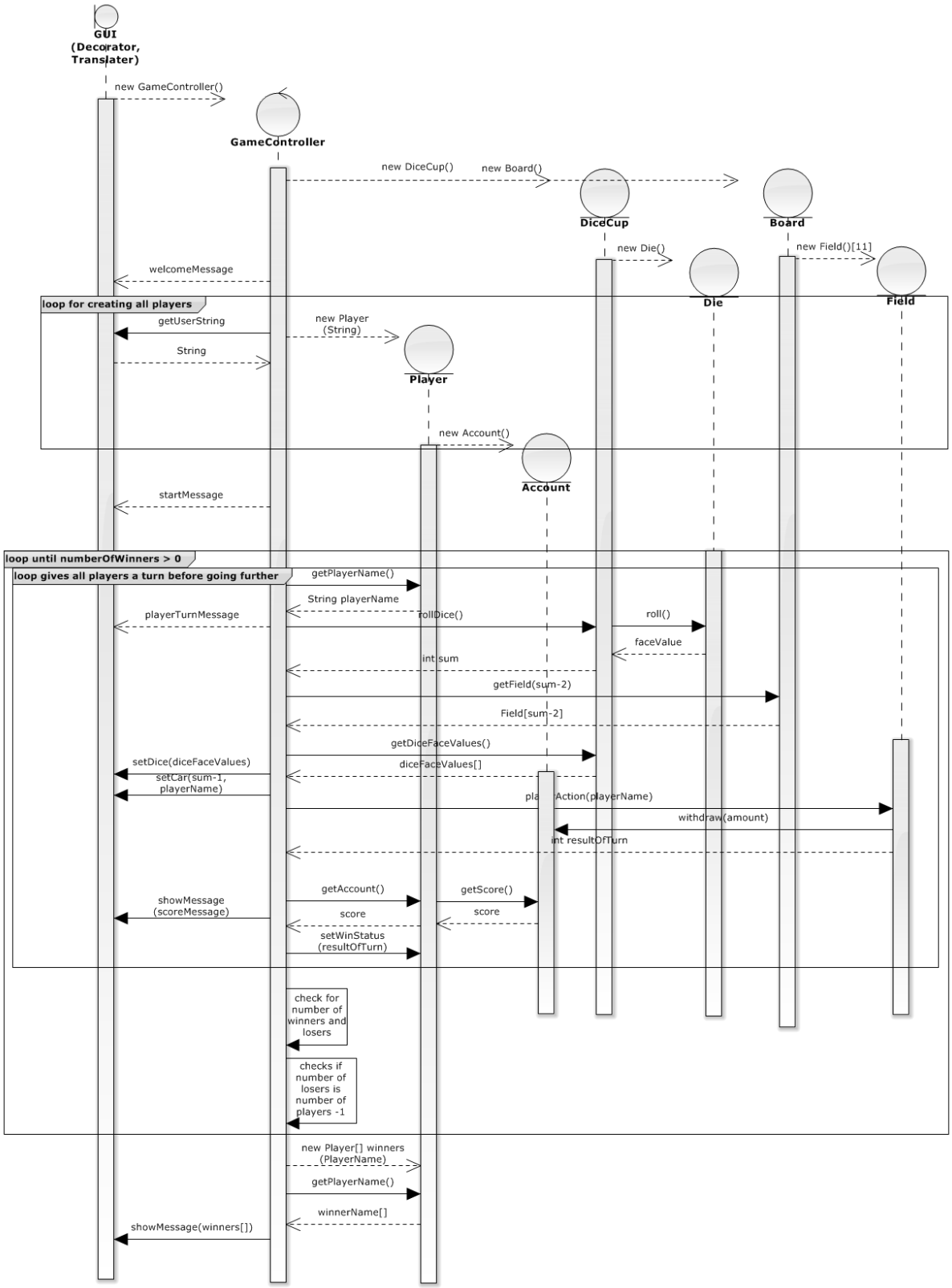
Design

Design klasse diagram



***Klassediagram**. Viser relationer mellem klasserne i programmet. Den statiske hjælper klasse SortPlayers er udeladt.*

Design sekvensdiagram


GUI
(Decorator, Translater)
GameController
DiceCup
Board
Die
Field
Player
Account
new GameController()
new DiceCup()
new Board()
new Die()
new Field()[11]
welcomeMessage
loop for creating all players
getUserString
String
new Player (String)
new Account()
startMessage
loop until numberOfWinners > 0
loop gives all players a turn before going further
getPlayerName()
String playerName
playerTurnMessage
rollDice()
roll()
faceValue
int sum
getField(sum-2)
Field[sum-2]
getDiceFaceValues()
diceFaceValues[]
setDice(diceFaceValues)
setCar(sum-1, playerName)
withdraw(amount)
int resultOfTurn
getAccount()
getScore()
score
showMessage (scoreMessage)
setWinStatus (resultOfTurn)
check for number of winners and losers
checks if number of losers is number of players -1
new Player[] winners (PlayerName)
winnerName[]
showMessage(winners[])

Design Sekvens diagram: *sekvens diagrammet er noget forsimplet, da det ikke indeholder decorator og translator, det er kun for at få bedre forståelse og da førnævnte ikke har noget med selv spillet at gøre. Ligesom system sekvensdiagrammet har vi også udeladt at GameController, efter hver besked til GUI, kræver respons. DiceCup laver to Die objekter og Board laver 11 Field objekter. withdraw(amount) som bliver sendt fra GameController til Field, kan også være deposit(amount), det kommer an på om amount er negativ eller positiv. check for number of winners and losers, checker hvilken værdi winStatus har (-1 eller 1). Hvis vindere er fundet slutter loopen, og hvis flere er vindere bliver de sorteret efter score og den med den højeste score, vinder.*

Implementering af GRASP principper

I oplægget er det en forudsætning at vi forsøger at overholde GRASP principperne Creator, Controller, Høj Binding, Information Expert og Lav Kobling.
For Creator patternet har vi gjort det ved at sørge for at de objekter der 'har' ('aggregates') andre objekter, også opretter dem. GameControlleren opretter Dicecup - da det er den der har referencen til den. Ligeledes opretter den Board og Player og Decoratoren. Med samme argumentation oprettes Die, Account og Field af hhv. DiceCup, Player og Board. Decoratoren er det eneste objekt i kontakt med GUI'en og får dermed ansvaret for at oprette den. På samme måde er den det eneste objekt i kontakt med translatoren, og opretter derfor også den.
Information Expert er implementeret ved at identificere hvilke klasser der bør kende til hvilke klasser. Controlleren bør kende til DiceCup, Board, Decorator og Player, da den 'aggregerer dem'. Den bør til gengæld ikke kende Field, Die, Account, GUI eller Translator. Det er hhv. Board, DiceCup, Decorator og Players ansvar, da de allerede har informationen om dem.
Lav Kobling er opnået ved at færrest mulige klasser tilgår hinanden direkte. Controlleren tilgår således kun Field, ved at bruge Boards getField(int); og tilgår kun Account ved at bruge Players getAccount(); Field bruger Player til at tilgå og modificere hans Account - men det er en kortvarig kobling, der har lavt impact, ved ændringer i koden.
Høj Binding er opnået ved at klasserne har afgrænsede arbejdsområder, der så vidt muligt vedrører egne egenskaber. Som ovenfor nævnt benytter Field dog Players Account til at opdatere den, men det er fortsat Accounts metode, der anvendes til at manipulere Account, hvorfor Høj Binding opretholdes.

Implementering

Sideløbende med vores analyse og design for vores to use cases 'Spil spillet' og 'Test konto' har vi arbejdet med at kode vores implementering. Som opgaven er opstillet har vi segmenteret vores kode i to packages, spil og test. Hver package indeholder en Klasse med en main-metode, hvor vi derfor netop har to controllers for begge vores use cases.

Beskrivelse af Klasserne i vores system

(default package)

Main
Eneste klasse i default package. Opretter en gameController og overgiver ansvaret til den.

package spil

Account
Har en int, der modsvarer pengebeholdningen. Har en constructor, der opretter en account med et beløb og en default constructor, der opretter en account med standardbeløbet 3000.
Account er specificeret i oplægget til at den ikke må få en negativ score. Derfor er der implementeret tjek for negative beløb i metoderne. Disse er handlet med exceptions som en øvelse i anvendelse af disse.
Har en standard getScore og en setScore, der indeholder et tjek for om metoden forsøges kaldt med en negativ int. Metoden deposit tjekker for om den modtager en negativ int og for om score vil gå over maksimumværdien for int. Metoden withdraw kontrollerer for en negativ int og for om score vil gå under 0.
toString er autogenereret af eclipse.

Board
Board indeholder et array med de 11 felter som er i spillet, og en getter, getFields() som returner hele arrayet. Den skal bruges til opsætning af spillepladen i GUI'en. Til sidst er en getter, getField(int field), som returnere et bestemt felt, med dennes egenskaber. Constructoren laver et array på 11 pladser, og kører metoden setupFields() som instantiere de 11 felter.

Decorator
Decorator er bygget som en composition af GUI'en, og har en tilhørende translator klasse, den er ansvarlig for at behandle al tekst og grafik der skal vises på GUI'en.
I constructoren bruges et throwaway JOptionPane til at vælge sprog, herefter oprettes en translator klasse med det givne sprog. Til sidst oprettes en ny GUI med tekst oversat til det valgte sprog.
Decorator har en masse uinteressante metoder som for eksempel setDice(int) der intet gør ud over at kalde den samme metode på GUI'en, det er lidt anderledes for de metoder der har med tekstinput til GUI'en at gøre. Eksempelvis showMessage(String…), disse metoder er lavet med String arrays for at vi kan sammensætte strenge af variable såsom brugernavn, og stadig oversætte resten af sætningen.

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

GameControlleren beder om at få kastet terningerne kører DiceCup metoden rollDice, så summen af de to terninger bliver beregnet. Når summen af de netop kastede terninger skal bruges, bruger GameController getSum, som returnere summen. Fordi at GUI'en også skal vise øjnene på de to terninger, indeholder DiceCup også getDiceFaceValues som returnere et array af de to terningers øjne, som GameControlleren så kan tilkalde. For at have muligheden for at lave de avancerede regler måtte vi også lave en getter til TwoOfAKind (getTwoOfAKind), denne indeholder ikke selve reglerne til spillet, men giver GameControlleren muligheden til at tilkalde og at få at vide om terningerne har slået to ens, og hvad der er slået to ens af. getTwoOfAKind returnerer derfor enten 0, hvis terningerne ikke er lige ellers 1 for to 1'ere og 2 for to 2'ere osv. String toString kan blive brugt hvis man vil vide terningernes værdier. Den kan være nyttig hvis man ikke kører spillet gennem en GUI, så bliver en String returneret med forklaring på hvilke værdier terningerne har.

Die

Klassen Die har to attributter (faceValue: Int og NUM_SIDES: Int). Den har et objekt der refererer til klassen Random, som bruges til at generere en pseudotilfæling integer med variablen NUM_SIDES og en integer med variablen faceValue, der holder terningens værdi. Klassen Die er udvalgt fra det forrige terningespil med den mest interessante metode roll (), der henter en pseudo tilfældig integer mellem 0 og 5 fra objektet Random, hvor faceValue sætter denne random til +1 og derefter returnerer faceValue.

Field

Field constructoren har 5 parametre som bestemmer objekternes attributter: String title, String subText, String description, int amount og int extraTurn. Der er tre forskellige strenge, det er fordi at GUI'en har tre forskellige metoder at skrive tekst, tre forskellige steder. Amount er den parameter som bestemmer om der skal trækkes fra eller lægges til spillerens score. ExtraTurn er en int som fortæller om feltet giver ekstra tur eller ej. Hvis extraTurn bliver sat til 0, er der ikke ekstra tur og hvis det er 1 er der ekstra tur. extraTurn er en int fordi det skal bruges i sandhedstabellen for resultOfTurn (PlayerAction metode i Field), hvor man kan få fire forskellige udfald, fra -1 til og med 2. Klassen har en getter for hver attribut samt to metoder, playerAction og updatePlayerAccount. playerAction opdaterer førs resultOfTurn = 0 + extraTurn, for den aktuelle spiller. Hvis spilleren lander på det felt som giver ekstra tur bliver resultOfTurn = 0 + 1. Derefter kører playerAction Field's egen metode updatePlayerAccount, som går ind på den aktuelle spiller og enten trækker fra hans score eller lægger til, med metoderne withdraw og deposit, som account har. playerAction og updatePlayerAccount kunne være samme metode, men er delt op til fremtidigt brug.

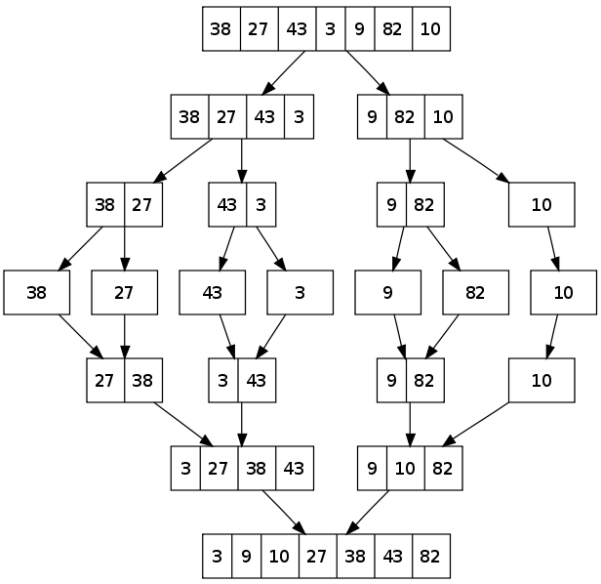
	All OK	All OK + extraTurn	Fail	Fail + extraTurn
Init = 0	0	0	0	0
ExtraTurn = 1	0	1	0	1
OK = 1	1	1	0	0
Fail = -1	0	0	-1	-1
Return	1	2	-1	0

Sandhedstabel for resultOfTurn der returneres fra Field.

GameController
Gamecontroller indeholder spillets logik, constructoren opretter DiceCup, Board og Decorator, og opretter et array af Players. Den har en public metode runGame(), samt to private hjælpemetoder playerTurn(Player) og playerSetup().
runGame kaldes af Main og bruger playerSetup til at få spillernes navne, herefter starter spillet. En tur køres af playerTurn, og hver spiller, der ikke er gået falit, får en tur før Gamcontroller checker efter vindere og tabere.
Tabere behandles ved at der inkrementeres en counter, hvis den counter når antallet af spillere - 1, bliver den tilbageværende spiller sat som vinder.
Når en vinder er fundet sorteres spillerne efter deres point, og og den spiller med flest point bliver vist som vinder på GUI'en. Hvis der er flere vindere bliver de alle sammen vist som uafgjort.

Player
Klassen Player har til formål at opbevare data om de enkelte spillere. Den indeholder variablerne playerName og winStatus, som henholdsvis har attributterne String og Int. Klassen har derudover de normale get/set metoder for både playerName og winStatus. Derudover er der en aggregation (associering) i mellem Klasserne Account og Player ved en getAccount () : Account metode.

SortPlayers
Hjælper klasse til at sortere spillerne efter deres point, så vi kan finde vinderen og eventuelle uafgjorte. Har to statiske metoder, en mergeSort(Player[]) og dens hjælper merge(Player[], Player[]). Dette er en rekursiv implementation af den velkendte mergeSort algoritme, den virker ved at dele arrayet op i dets mindste bestanddele, og sætte dem sammen igen i den rigtige rækkefølge.



***MergeSort**. Viser hvordan mergesort først splitter data op, og derefter sætter det sammen igen i stigende rækkefølge.*
Kilde: http://en.wikipedia.org/wiki/File:Merge_sort_algorithm_diagram.svg (9/11/2013)

Translator
Translator er en klasse designet til holde data omkring text. Dens constructor tager en filplacering som en streng, og opretter et file-object. Herefter læser Translator data fra filen ind i et hashtable, hver linie deles op i en key og en value, så de er klar til brug i programmet.
Translator har kun en enkelt public metode, translateString(String): String. Denne tager en string fra programmet, forsøger at finde en key der matcher i hashtablet, og returnere den tilhørende value.
Hvis der ikke findes en key der matcher i hashtablet returneres den string som metoden blev kaldt med, dette er gjort sådan for at ting som tal og brugernavne kan behandles af samme metode.

Package test

MainAccountTest
Test klasse der tester vores implementation af account, ved at kalde withdraw, deposit og setscore med nogle udvalgte grænsetilfælde.

SortPlayersTest
Dette er en simpel testmetode der viser at mergesort fungere, også på mere end 2 spillere.

Test af software systemet

Test af "Spil"

Vi har testet vores implementation af spillet, og fundet et par fejl relateret til vores brug af GUI'en. Vi har ikke fundet fejl i logikken bag spillet, det kører som forventet og giver, så vidt vi kan se, de ønskede resultater.
De fejl vi har fundet er:

- Der er et et offset mellem tallet på terningerne og nummeret på feltet man lander på. Dette har at gøre med måden felterne genereres på, og kunne muligvis afhjælpes ved at indsætte et "dummy" felt som felt nr. 1.
- Hvis 2 spillere har samme navn bliver der kun vist én spiller på GUI'en. Logikken bag spillet arbejder stadig med 2 spillere og spillet fungere stadig.
- Hvis man vælger "spil igen" efter at have afsluttet et spil, bliver spillerne ikke slettet fra GUI'en. I stedet bliver 2 nye spillere tilføjet. Disse to bliver nu brugt i logikken og spillet fungere, men de to forrige spillere forbliver på pladen selvom de er inaktive.

Test af "Account"

For at afprøve account har vi anvendt en test klasse MainAccountTest, der forsøger at afprøve grænsetilfældene mellem ækvivalensklasserne. Der testes for om kontoen kan sættes til negativ, positiv og nul. Der testes for om der kan indsættes negativt og positivt beløb, samt beløbet nul. Desuden testes for om der kan indsættes et beløb, så kontoens saldo overstiger største integer værdi (usandsynligt scenarie, men bør håndteres). Desuden testes der for om der kan hæves positivt, negativt beløb og nul. Der testes også for om der kan trækkes et beløb, så kontoen går i negativ og et beløb så kontoen går i nul.

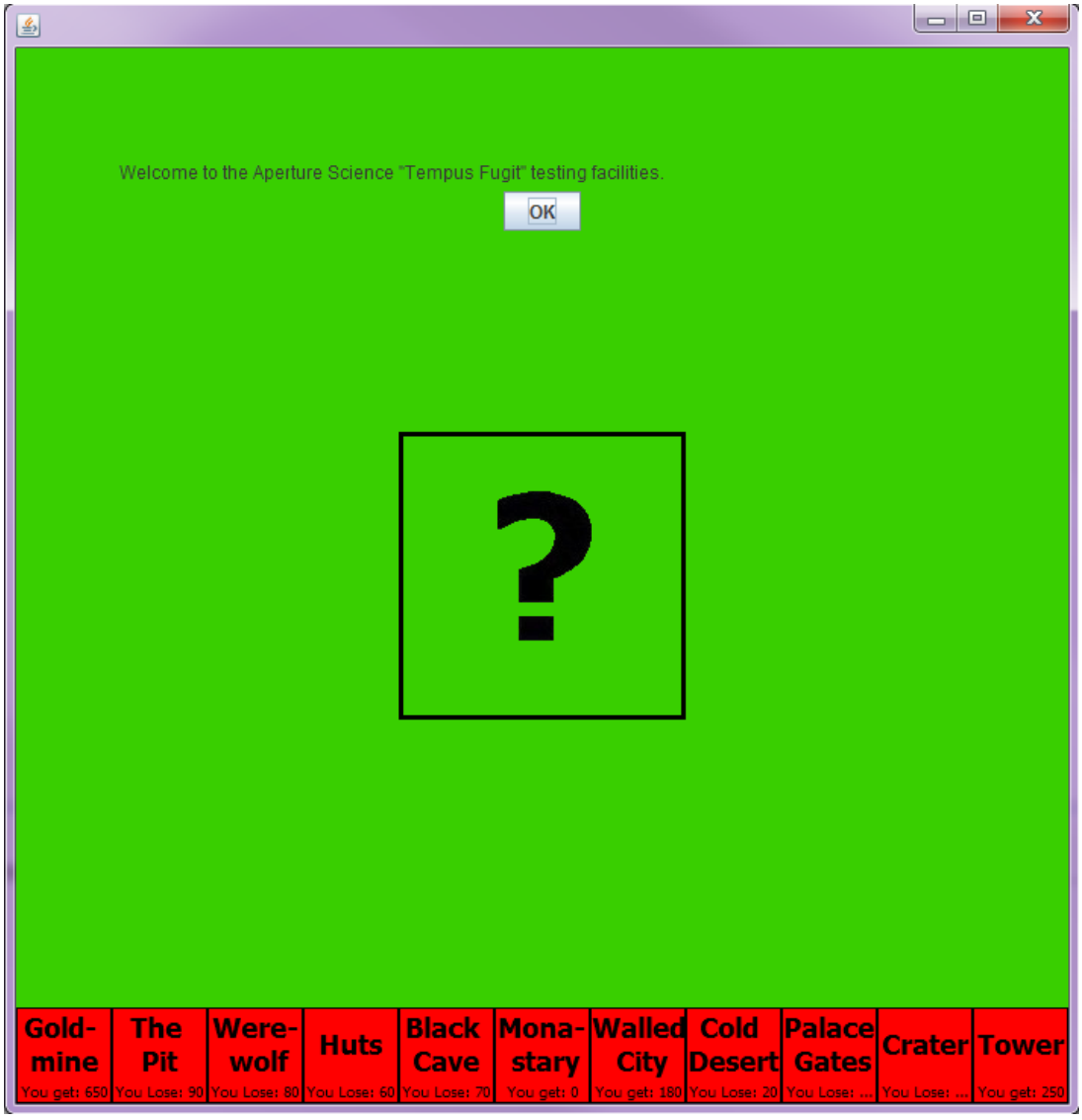
Konklusion

Det er lykkedes os at udvikle en fungerende løsning på problemstillingen. Koden er visse steder blevet unødvendigt kompliceret. Vi har valgt et designmønster med en decorator og en translatorklasse for at udfordre de af os, der allerede har kodeerfaring.
I klassen Account er der anvendt exceptions til at håndtere de tilfælde hvor account er i risiko for at gå i negativ. Anvendelsen af exceptions er experimentel og ikke gennemført i resten af programmet.
Sent i processen er det gået op for os at vi ikke havde taget højde for håndtering af fallit situtationen og vores kode var uheldigt struktureret til det scenarie. Derfor endte vi med en unødvendigt kompliceret algoritmik i vores controllerklasse.

Bilag

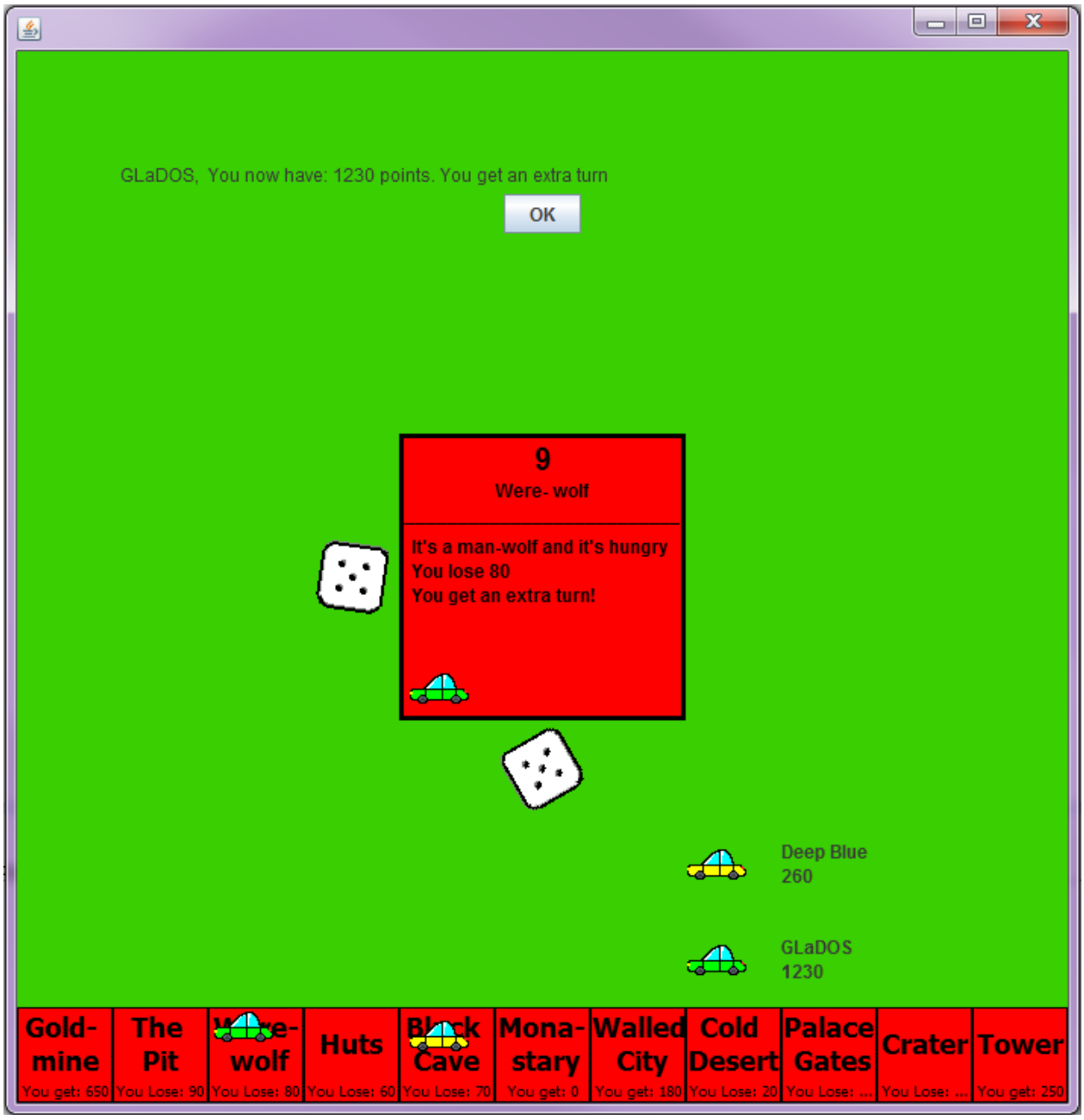
Bilag 1

Screenshot af spil: Opstart



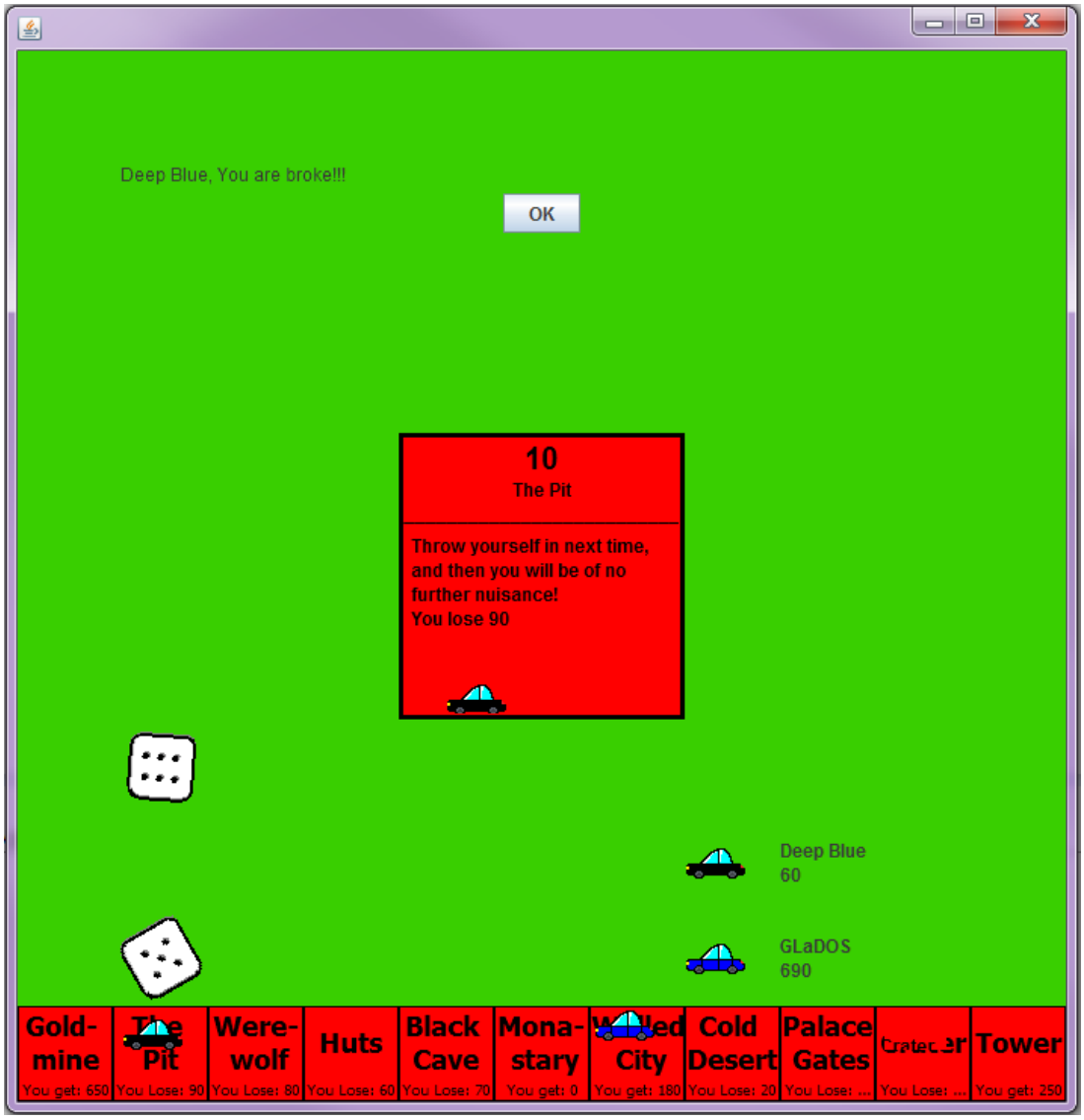
Bilag 2

Screenshot af spil: Midt i spillet



Bilag 3

Screenshot af spil: En spiller går fallit



Bilag 4

Screenshot af spil: Viser alle tre nævnte fejl, slår 4 lander på 3, kun en extra spiller da de har samme navn, 2 forældede spillere der er inaktive.