CDIO 3

Kursus nr. 02313, 02314, 02315 - Gruppe nr: 45
Denne rapport indeholder 32 sider inkl. bilag og denne side.

Navn: Mads Martin Dickmeiss Hemer

S-nummer: s170185

Email: s170185@student.dtu.dk

Navn: Malte Brink Kristensen

S-nummer: s185039

Email: s170185@student.dtu.dk

Navn: Nicolai de Thurah Wulf

S-nummer: s185036

Email: s185036@student.dtu.dk

Navn: Neal Patrick Norman

S-nummer: s060527

Email: s060527@student.dtu.dk

Navn: Camilla Bruun Simonsen

S-nummer: s185038

Email: s185038@student.dtu.dk

Indholdsfortegnelse

Indledning

Den følgende rapport handler om et juniormatadorspil, vi har udviklet til IOOuterActive. Det er et simpelt spil, der i dets nuværende form kan betragtes som i en slags alpha-version. Spillet spilles af 2-4 spillere, der skiftes til at slå med en terning og får rykket en brik rundt på en spilleplade hovedsagligt bestående af grunde, der kan købes.

Via forskellige UML-artefakter finder vi frem til, hvordan spillet bliver kodet. Efterfølgende tester vi om et Bankroll-objekt bliver oprettet korrekt af et Player-objekt (i form af en driver), om Bankroll klassen's changeBalance metode kan ændre den aktuelle pengebeholdning, og om spillet crasher ved indtastning af forskellige navne.

Analyse

I de følgende afsnit specificerer vi kravene i en vision, en kravspecifikation, en navneordsanalyse og en domænemodel.

Vision og kravspecifikation

Det følgende er baseret på kundens egen formulering af deres vision. Vores vision er at udvikle et spil mellem to til fire personer, der kan spilles på maskinerne i DTU's databarer. Spillet går i korte træk ud på, at spillerne på skift slår med en terning og lander på et af 24 felter på en rund spilleplade.

Ud fra kundens vision vil aktører til dette program være alle der opholder sig på DTU. studerende, undervisere og evt gæster. Interessenterne til spillet vil være spilleren i form af gæster, studerende og undervisere samt kunden.

Funktionelle krav til systemet

K1	Spillet skal kunne spilles af to til fire personer.
K2	Spillerne skal på skift kunne slå med en terning, der er opfører sig som en teoretisk symmetrisk terning med en fejlprocent på højest 2% ved 1000 testslag.
K3	Spillerne skal foregå på en spilleplade med 24 felter, som spillerne kan lande på, og som har forskellige effekter på spillerne.

K4	Hver spiller starter med en pengebeholdning, der afhænger af antallet af spillere: 2 spillere: 20 hver. 3 spillere: 18 hver. 4 spillere: 16 hver.
K5	Når en spiller lander på et felt, skal spilleren købe feltet, hvis feltet er ledigt.
K6	Når en spiller lander på et felt og feltet er ejet, skal spilleren betale husleje. Huslejen er dobbelt hvis begge felter af samme farve ejes af samme person
K7	Når en spiller lander på et felt og spilleren selv ejer feltet, sker der ikke noget.
K8	Der er fem specielle felter, der har hver deres tilknyttede effekt (se tabel 1)
K9	Hver gang en spiller passerer startfeltet eller lander på startfeltet, får spilleren 2 penge
K10	Spillet skal slutte, når en spillers pengebeholdning bliver negativ. Det skal da være den spiller, der har flest penge, som vinder.
K11	Den yngste spiller starter. Spillerne skal derfor kunne indtaste deres alder, før spillet starter.
K12	Spillerne skal fortsætte deres tur fra det felt, den forrige tur sluttede på.
K13	Spillet skal kunne spilles på forskellige sprog (UI skal kunne vise tekst på forskellige sprog, og spillerne skal kunne skifte imellem disse).
K14	Spillet skal ved hver tur præsenteres en menu med følgende valgmuligheder: - Slå terninger - Giv op - Se stilling
K15	Spillet skal starte med visningen af en hovedmenu med følgende valgmuligheder: - Starte et spil - Læse reglerne - Skifte sprog - Afslutte programmet
K16	Når begge grunde i samme farvekategori ejes af samme spiller, skal der betales dobbelt husleje af den spiller, der lander på grundene.

Tabel 1: Funktionelle krav.

Tabel over krav

Krav	Must have	Nice to have	Should have
K1	x		
K2	x		
K3	x		
K4			x
K5	x		
K6		x	
K7		x	
K8		x	
K9	x		
K10	x		
K11		x	
K12			x
K13	x		
K14		x	
K15	x		
K16		x	

Tabel 2: Inddeling af funktionelle krav.

Felter

Felt	Effekt
Start	Spilleren modtager 2 penge fra banken, når feltet passeres eller landes på
Chance	Spilleren tager et chancekort og følger

	instruktionerne
Gå i fængsel	Spilleren rykkes til fængslet
Fængsel/på besøg	Hvis spilleren lander her, sker der ikke noget
Gratis parkering	Hvis spilleren lander her, sker der ikke noget

Tabel 3: Oversigt over de specielle felter og deres effekter.

Supplerende kravspecifikationer

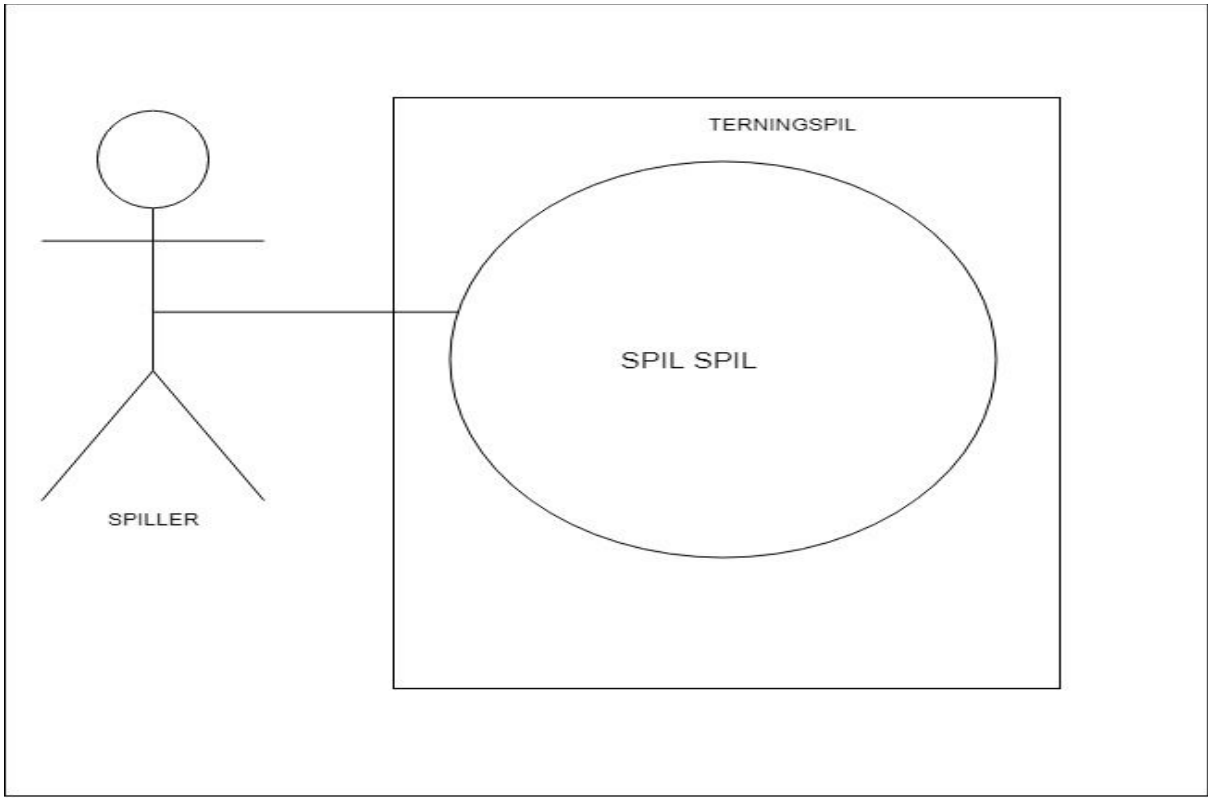
Ud over de funktionelle krav er der en række supplerende specifikationer til projektet. Dette inkluderer ikke-funktionelle krav, useability-krav og krav, der relaterer sig til selve udviklingsarbejdet.

- Kunden skal kunne se hver enkelt gruppemedlems bidrag.
- Kunden skal kunne se, hvordan udviklingen er foregået.
- Der skal leveres et git-repository så kunden kan se og følge med i commits.
- GRASP-principperne skal overholdes.
- Spillet skal kunne spilles på DTU´s databar, uden bemærkelsesværdige forsinkelser. Vi vil foreslå en responstid på 1-5 ms.
- Spillet skal ligesom terningespillet i CDIO2 kunne spilles på DTU´s databar uden bemærkelsesværdige forsinkelser.

Use case diagram

Vores ene use case (UC1) beskriver den ene funktion, som vores system indeholder – nemlig et spil, der spilles:

UC1



Figur 1: Use case 1.

Use case beskrivelse

Nedenfor har vi beskrevet vores use case. Først brief, så casual, så fully dressed. Ud fra use casen laver vi en navneordsanalyse, en domænemodel og et systemsekvensdiagram.

UC1: Spil spil

Brief: Spillerne starter spillet og skiftes til at slå med en terning. Vinder findes ved at en modstander går fallit.

Casual:

Hovedsuccesscenarie:
Spillerne starter spillet og skiftes til at slå med en terning. Afhængig af øjnenes værdi lander spillerne på forskellige felter, der påvirker deres pengebeholdning og spillets gang. Hvis en spiller går fallit, vinder spilleren med størst pengebeholdning.

Alternative scenarier:

- En spiller giver op. Spillet slutter straks.

- En spiller vil se stillingen. Stillingen vises og spillet fortsætter.

Fully dressed:

Scope: Monopoly junior.
Level: User goal.
Primær aktør: Spiller.
Interessenter:

- Spiller: Vil gerne spille (og vinde) et spil med en modspiller.
- IOOuterActive: Vil gerne have tilfredse spillere.

Postconditions: Et spil er afsluttet og en vinder er fundet.
Hovedsuccesscenarie:

1. Hvis spiller ønsker at ændre sprog inden spil:
    a. Inkludér (Skift sprog inden spil).
2. Spillerne præsenteres for en hovedmenu, hvor de kan vælge enten at starte et spil, læse reglerne, skifte sprog eller afslutte programmet.
3. Spillerne starter et spil.
4. Den yngste spiller starter.
5. Spilleren, der starter (spiller 1), præsenteres for en menu med de valg, han/hun har.
6. Spiller 1 vælger at slå med terningen.
7. Summen af terningen findes, og spiller 1 lander på det felt, som passer med den sum terningen viser lagt til det felt som spilleren står på.
8. Spiller lander på et felt, hvis det ikke ejes af andre spillere er spiller tvunget til at købe dette felt. Ejes feltet af en anden spiller skal spilleren betale “huslejen til feltets ejer”
    8.1. Spiller lander på et af de 5 special felter (læs 4, da vi har fjernet chancekort)
        a. Start, spiller modtager 2 penge (hvis start passeres eller hvis man lander på startfeltet, dog ikke hvis spiller passerer start som følge af, at man ryger i fængsel)
        b. Gå i fængsel, spiller ryger i fængsel.
        c. På besøg/fængsel, intet sker.
        d. Gratis parkering, intet sker.
            -
9. Turen går nu til spiller 2, som gennemgår det samme flow som spiller 1 netop har været igennem (dvs. punkt 3-8).
10. Spillet forsætter sin gang som beskrevet ovenfår, hvor de 2-4 spillere får en tur på skift.
11. Når en spiller går fallit, slutter spillet, og den spiller, der har flest penge, vinder.

12. Spillerne præsenteres igen for hovedmenuen, hvor de har samme valg som i punkt 1.
13. Når spillerne ikke ønsker at spille længere, lukker de programmet enten ved at vælge “luk programmet” i hovedmenuen, eller på anden vis.

Alternative scenarier:

1. Spiller beder om at læse spillets regler.
    a. Systemet viser reglerne.
2. Spiller beder om at tjekke scoren
    a. Systemet viser scoren
3. Spiller beder om at stoppe med at spille
    a. Systemet lukker ned.

Forekomstfrekvens:
Konstant.

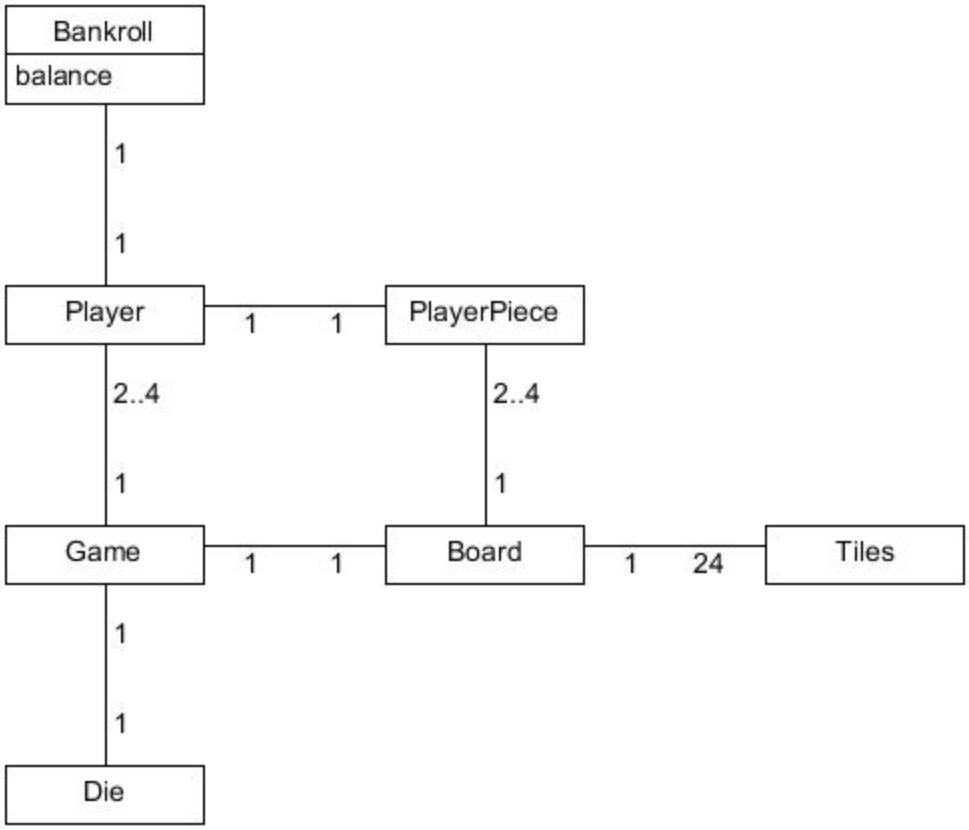
Navneordsanalyse

I navneordsanalysen ser vi på alle de navneord der giver anledning til en klasse eller attribut i vores domænemodel.

Navneord:
Spil, spiller, felt, spilleplade, sprog, terning, menu, regler, point/penge, pengebeholdning, husleje, ejer, spillebrik.

Domænemodel

Domænemodellen viser de grundlæggende elementer, der indgår i et Junior Matador-spil. Klasserne er skrevet på engelsk, for at relatere det til de klasser vi har kodet.



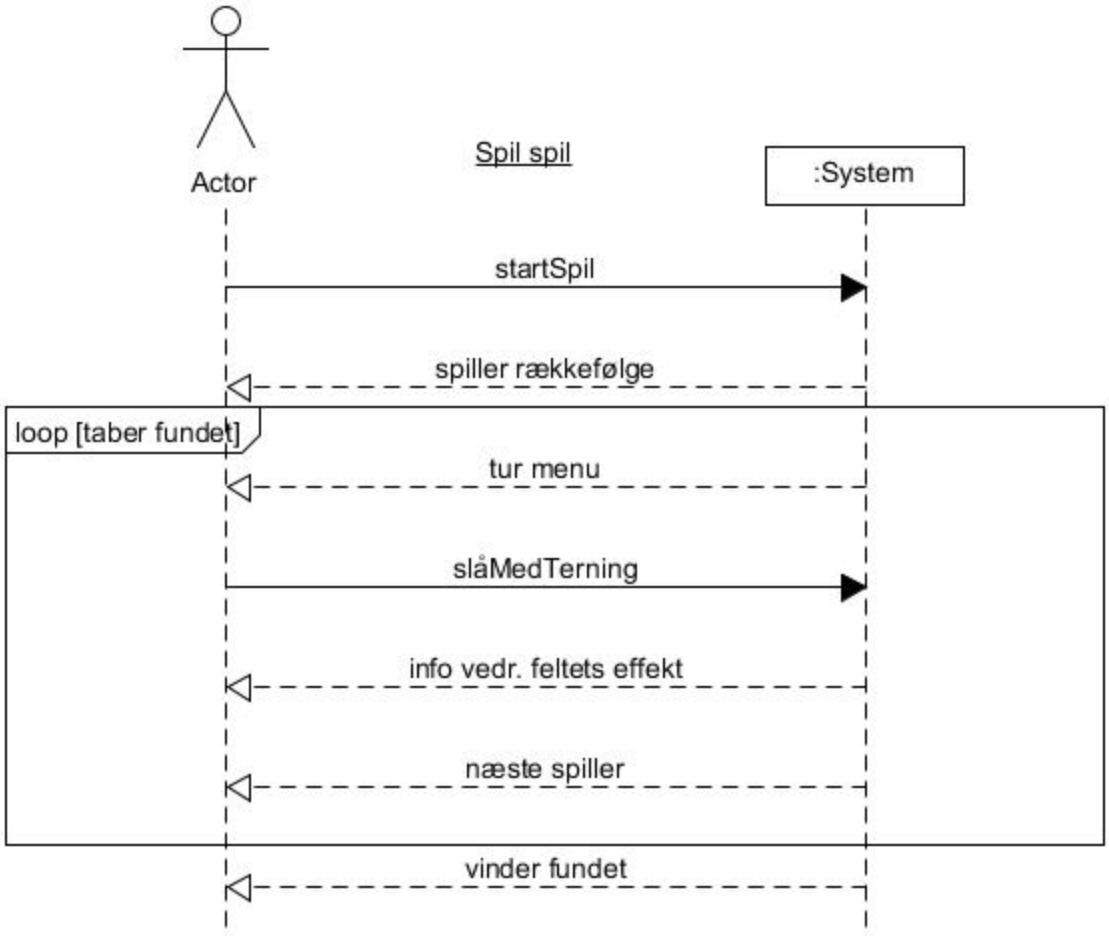
Figur 2: Domænemodel. Til et spil er der én terning, to til fire spillere og et spillebræt. Spillebrættet har 24 forskellige felter. Spilleren har en pengebeholdning og en brik.

Ordliste domænemodel

Bankroll: Pengebeholding
Player: Spiller
PlayerPiece: Spillebrik
Game: Spil
Board: Spilleplade
Tiles: Felt
Die: Terning
Balance: penge/point

Systemsekvensdiagram

Systemsekvensdiagrammet beskriver forløbet af et spil. Under overfladen er spillet faktisk så simpelt, at hver tur kun består af et terningslag og den efterfølgende effekt af dette, da spilleren ikke rigtig har nogle egentlige valg.

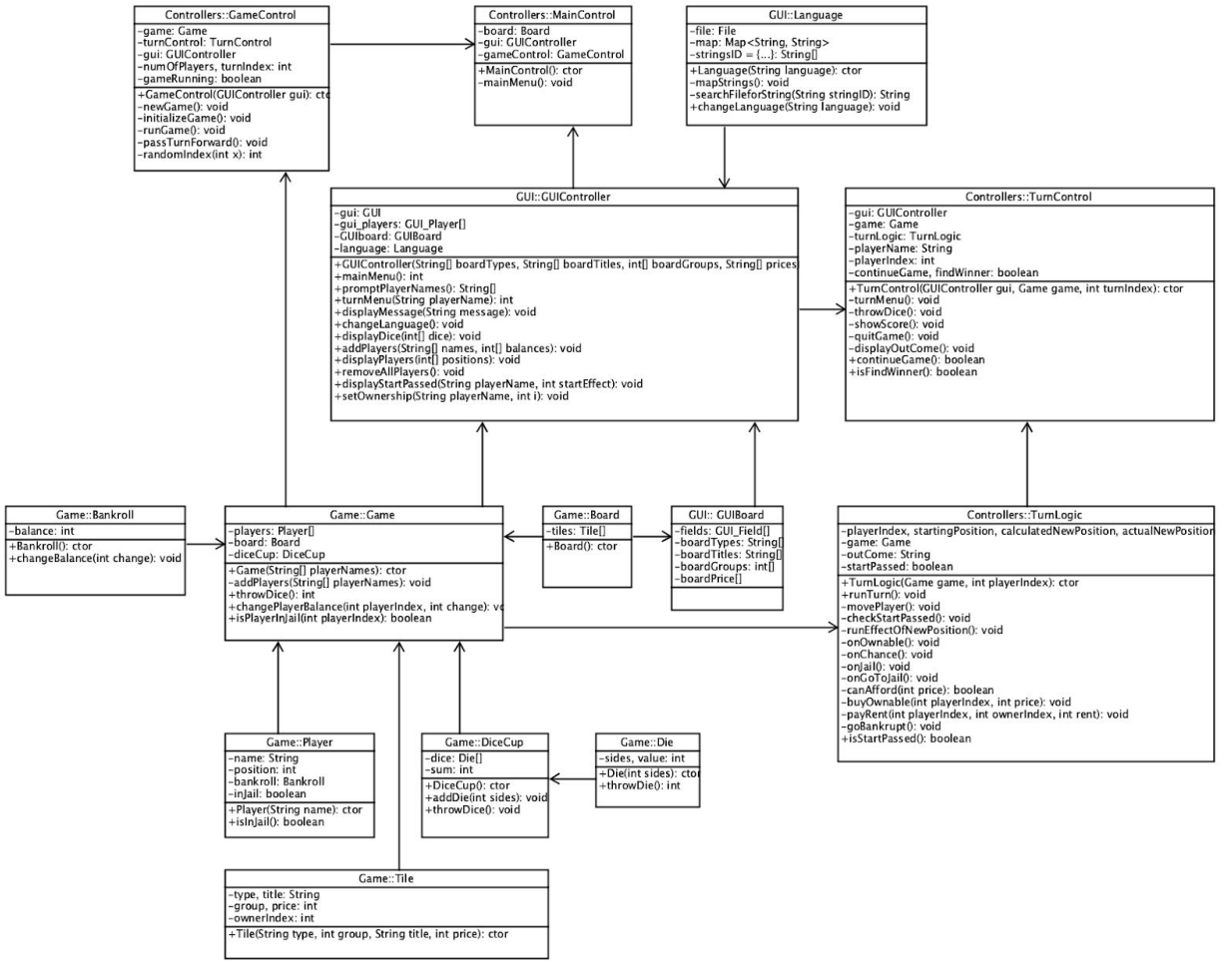


Figur 3: Systemsekvensdiagram. En spiller starter et spil, hvorefter rækkefølgen af spillerne fastsættes, og spiller 1 tager herefter første tur. Når en spiller taber, dvs. ikke har flere penge, stopper spillet, og vinderen er den spiller, der har flest penge.

Design

Nedenfor ses vores designmodeller - det vil sige et designklassediagram og et sekvensdiagram.

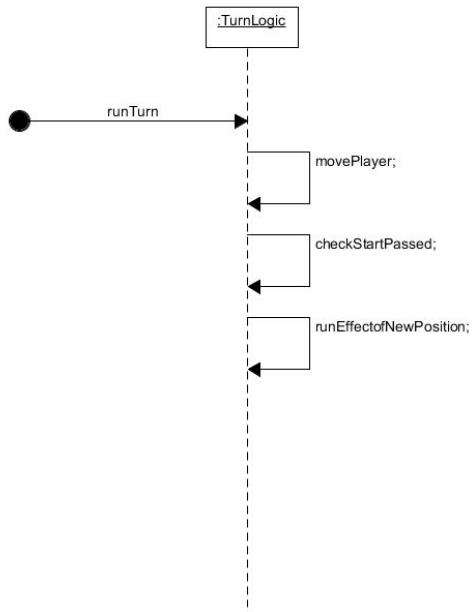
Designklassediagram



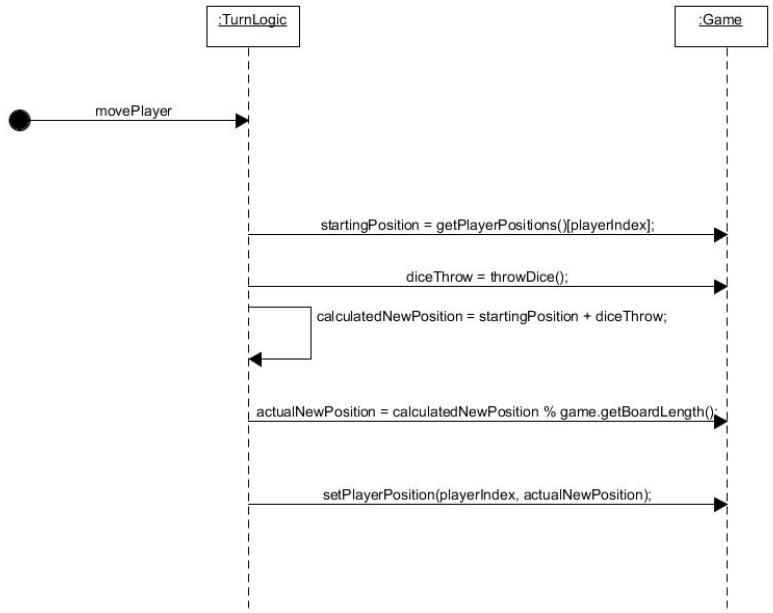
Figur 4: Designklassediagram. Diagrammet har været opdateret løbende i udviklingsprocessen, således at den ovenstående version lægger sig meget tæt op af den kode, vi har implementeret.

Sekvensdiagram

Vi har udarbejdet et sekvensdiagram, som viser metodekald i klassen TurnLogic for metoden runTurn. Derudover har vi uddybet metoden movePlayer i et separat sekvensdiagram.



Figur 5: Sekvensdiagram over metodekaldet runTurn i klassen :TurnLogic



Figur 6: sekvensdiagram over metodekald i klassen :TurnLogic hvor den bruger kendskab til klassen :Game til at udføre metoden movePlayer.

Implementering

Arv, abstrakte klasser og polymorfisme

Hvad er arv?

I objekt-orienteret programmering arbejdes der med klasser, der instantieres som objekter. Objekter er adskilte strukturer, der indeholder data og kommunikerer med andre objekter. Ofte designes klasser, således at de repræsenterer et koncept eller en ting - både virkelige eller mere computer-relaterede, fx en bil-klasse eller en hovedmenu-klasse. Vil man gerne implementere et objekt af en klasse, der er en underkategori til en anden klasse, fx en bestemt biltype eller en alternativ version af hovedmenuen, kan man anvende nedarvning. Ved nedarvning vil subklassen automatisk indeholde superklassens data (attributter) og kommunikationsformer (metoder). Subklasser arver med andre ord alt indhold fra deres superklasse.

I Java angiver man at en klasser arver fra en superklasse vha. “extends” keyword’et.

Eksempel:

```
public class ClownCar extends Car {...}
```

Car er superklasse for ClownCar.

Hvad betyder abstract?

En klasse, der repræsenterer et koncept på et mere overordnet, abstrakt plan kaldes for en abstrakt klasse. En abstrakt klasse vil man aldrig instantiere (bruge som objekt) i sit program, men blot anvende som en måde at beskrive de fællestræk, som den abstrakte klasses subklasser skal indeholde. Den ovennævnte bilklasse kunne være et eksempel på en abstrakt klasse - måske vil man aldrig bare have et rent bil-objekt i sit program, men udelukkende mere specifikke bil-subklasser. I et andet program kan det dog måske give god mening at have rene bil-objekter. Hvad der er abstrakt, er således kontekstafhængigt.

En abstrakt klasse i Java kan defineres ved hjælp af “abstract” keyword’et. En klasse, der er abstract, kan også have metoder, der er abstract. En abstract metode har intet indhold, hvilket betyder, at den ikke kan anvendes af en subklasse, før den er blevet defineret i subklassen.

Eksempel:

```
public abstract class Car {...}
```

Car er en abstrakt klasse.

Hvad hedder det, hvis alle fieldklasserne har en landOnField metode, der gør noget forskelligt?

I objekt-orienteret programmering refererer polymorfisme til det koncept, at det samme statement kan blive udført på forskellige måder af programmet. Overordnet kan der skelnes mellem to typer af polymorfisme, compile time polymorfisme og run time polymorfisme.

Ved run time polymorfisme kan programmet, som navnet antyder, først kende den korrekte udførelse af en given sætning, når programmet køres. Dette fordi programmets aktuelle status afgør, hvilken udførelse, der er den rigtige. I Java er et almindeligt eksempel på run time polymorfisme metodetilsidesættelse (eng.: method overriding).

Metodetilsidesættelse kan for eksempel opstå i forbindelse med nedarvningsforhold. I Java kan objekt-reference-variable både referere til objekter af selve klassen, variablen er deklareret som, men også objekter af denne klasses eventuelle subklasser. Samtidig kan en subklasse kan have tilsidesat en eller flere af superklassens metoder ved at definere en metode med samme signatur. Resultatet af dette er, at der kan opstå situationer, hvor en variabel på et givent tidspunkt kan indeholde objekter af forskellige klasser, der alle har metoder med samme signatur. I det tilfælde vil den metode, der bliver udført af programmet (ved run time), afhænge af hvilket objekt, den givne variabel rent faktisk indeholder på det tidspunkt, hvor metoden bliver kaldt.

Eksempel:

```
public class Car {
…
public void drive() {...}
}
```

```
public class ClownCar extends Car {
…
@override
public void drive() {...}
}
```

ClownCar's metode, drive, tilsidesætter superklassen Car's drive-metode

Ved compile time polymorfisme kan programmet godt kende den korrekte udførelse af en given sætning inden run time. Det er fordi de relevante faktorer er fastsat på forhånd i programmets implementering. Et almindeligt eksempel på compile time polymorfisme i Java er metodeoverbelastning (eng.: method overloading). Metodeoverbelastning refererer til det forhold, at en klasse kan indeholde flere forskellige metoder med samme navn, men forskellige

antal parametre, det vil sige forskellige signaturer. Hvilken metode, programmet så vil udføre, afhænger dermed af den signatur, der bliver anvendt.

Eksempel:

```
public class Car {
…
public void drive(a) {...}

public void drive(a, b) {...}
}
```

Klassen Car har to drive-metoder, der har forskellige signaturer. Drive-metoden er overbelastet, men udgaven har en unik signatur.

Svaret på det spørgsmål, der stilles i nærværende afsnits overskrift, er altså følgende: Det antages i spørgsmålet, at der er implementeret en række field-klasser, der nedarver fra en field-superklasse. Disse field-subklasser har alle en "landOnField" metode, der har tilsidesat superklassens landOnField metode. Med andre ord er der tale om polymorfisme - nærmere bestemt run time polymorfisme.

GRASP

I dette afsnit beskrives vores forsøg på - med varierende grader af succes - at anvende GRASP-designprincipperne.

Controller
Princippet i "controller"-mønsteret tager udgangspunkt i at tildele ansvaret for håndteringen af systemhandlinger og fordeling af ansvarsopgaver til en klasse, der ikke er en del af UI'et, men ligger lige i "laget under". Dette har vi forsøgt at opnå ved at oprette flere "controller"-klasser, bl.a. MainControl, som er den første klasse, der instantieres, når programmet køres.

MainControl sørger da for at kalde GUI'en, så der vises en hovedmenu, hvorefter MainControl behandler inputtet fra GUI'en. To andre controllers har også kendskab til GUI'en: GameControl og TurnControl. GameControl sørger for hele flowet fra at et spil startes med et antal spillere, til at turen går på skift, indtil der er fundet en vinder og spillet stoppes. TurnControl tager sig da af den kommunikation, der foregår mellem spillogikken og GUI'en ved hver tur. Essensen af selve spillogikken ligger i klassen TurnLogic, som ikke har kendskab til GUI'en, men blot til Game-klassen.

Creator
Princippet om "creator" har vi forsøgt at følge, ved at sørge for, at de klasser, der aggregerer, indeholder og gør brug af en anden klasse typisk også er den, der instantierer den. Fx kan

nævnes at MainControl instantierer GameControl, der så igen instantierer Game og TurnControl. TurnControl instantierer da TurnLogic. Game sørger da for at instantiere Player-objekter og en Dicecup. En mulig afvigelse fra princippet om creator kan være, at Board instantieres af MainControl, selvom det måske ville være mere logisk, at den blev instantieret af Game. Dette har vi gjort, fordi Board da gives som argument til instantieringen af GUIController, som da sørger for at oprette et GUIBoard, der svarer til det Board, der er defineret i Board-klassen. På den måde kan Board ændres, uden at GUIBoard behøver at blive ændret.

High cohesion
Princippet om high cohesion har vi tilstræbt at følge ved at holde ansvarsområderne så små og tydeligt definerede som muligt for hver klasse, hvilket også har medført et større antal af klasser. Endvidere har vi forsøgt at holde de enkelte metoder så korte og tydelige som muligt. Et eksempel kan være nogle af de controller-klasser, der tidligere er nævnt, der bryder lagene af kontrol op i ansvarsområder for hhv hovedmenu, flowet for et helt spil, og flowet i en enkelt tur. Som et andet eksempel kan fx nævnes, at vi vores klasse for en terning, Die, ikke selv skal holde styr på, om der er flere terninger i spillet og hvad deres fælles sum i så fald er. Dette ansvar lægges ud til DiceCup, som håndterer alle terninger, kaster dem og beregner deres sum.

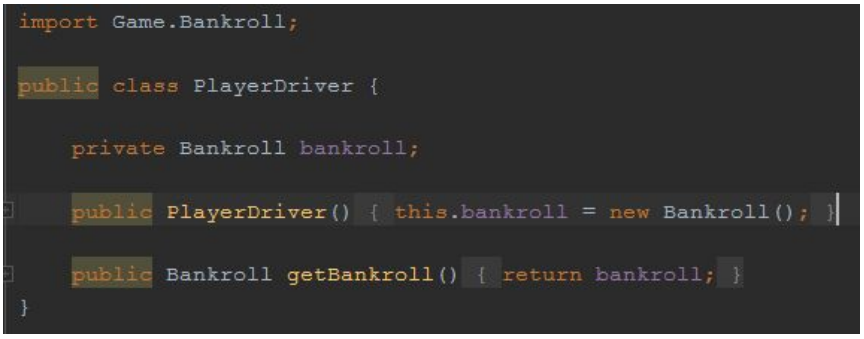
Information expert
Vi har forsøgt at efterleve princippet om “information expert” ved at sørge for, at hver klasse holder på lige nøjagtig de informationer, der skal til for, at den kan udføre sit ansvarsområde.

Low coupling
For at opnå low coupling - forstået som princippet om lav afhængighed og mindst muligt “kendskab” mellem klasser - har vi bl.a. oprettet en del wrapper-getter-metoder, især i Game-klassen, der sørger for at hente informationer fra “information experts” som f.eks de enkelte Tile-objekter i Game’s Board-attribut, eller fra de enkelte Player-objekters respektive Bankroll-attributter. På den måde behøver fx TurnLogic kun at kende til Game-klassen, og ikke Player, Board, Tile, Dicecup etc.

Et andet eksempel kan være, at vi i MainControl har oprettet fire getter-metoder, der sørger for at hente informationer fra Board-klassen og lagre dem i hver sin array med enten strings eller integers. Disse bruges da som argument i instantieringen af både GUIController og GUIBoard, hvilket medfører, at GUI’en ikke kender til Board-klassen.

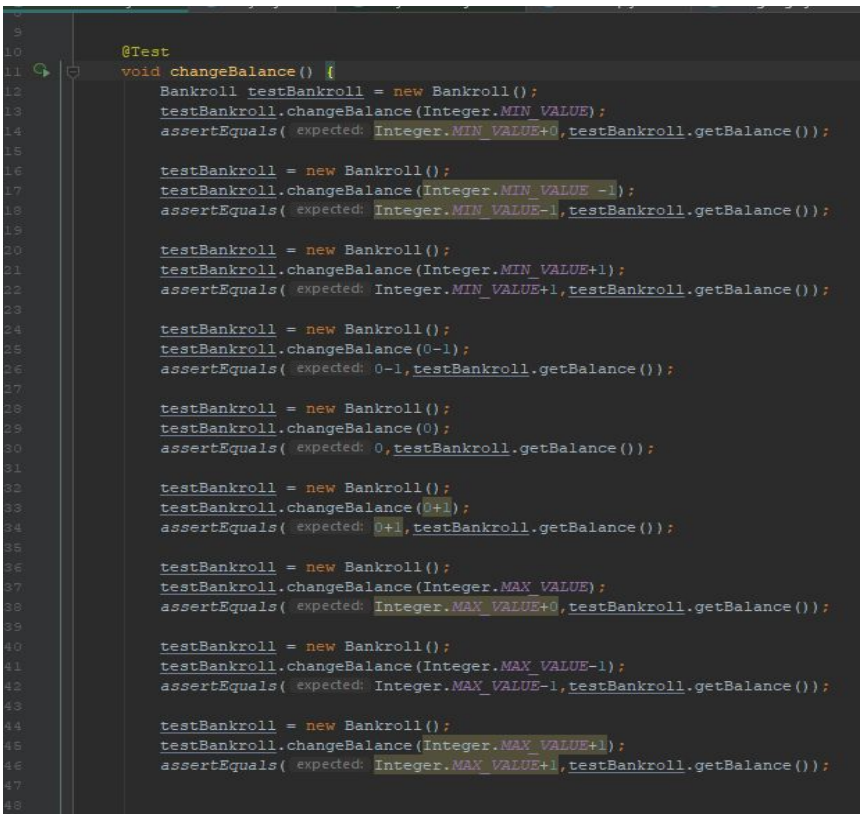
Test

Positive test

Test case ID	TC01 Test af Bankroll constructor
Summary	Tester om et Bankroll-objekt bliver oprettet korrekt af et Player-objekt (i form af en driver).
Requirements	Denne test er grundlag for at flere krav kan overholdes og testes. K4, K5, K6, K9, K10
Preconditions	-
Postconditions	Et PlayerDriver objekt er oprettet med et Bankroll-objekt som attribut. Bankroll har desuden 0 i startbalance (da den endelige start balance først udregnes, når spillet sættes op).
Test procedure	1. Opret PlayerDriver 2. Test om bankroll-attributten har det korrekte indhold 3. Test om bankroll balancen er 0 ved instantiering  
Test data	-
Expected result	Bankroll-attribut bliver instantieret med bankroll-objekt, der har 0 i

	balancen.
Actual result	Bankroll-attribut bliver instantieret med bankroll-objekt, der har 0 i balancen.
Status	Bestået
Tested by	Gruppen
Date	28-11-2018
Test environment	En JUnit-test i intelij LEVONO Y500

Test case ID	TC02 Bankroll changeBalance
Summary	Blackbox test af Bankroll klassen's changeBalance metode, der skal kunne ændre den aktuelle pengebeholdning
Requirements	Denne test er grundlag for at flere krav kan overholdes og testes. K4, K5, K6, K9, K10
Preconditions	Der skal være et bankroll-objekt med en pengebeholdning .
Postconditions	Pengebeholdningen er blevet ændret
Test procedure	1. Opret Bankroll-objekt 2. Anvend changeBalance(int change) metode ved ækvivalensklassernes grænseværdier ±1.

Test data	Ækvivalensklasser:
	- Alle positive heltal
	  - Grænseværdier:
	  - 0
	  - Højeste Integer
	- Alle negative heltal
	  - Grænseværdier:
	  - Laveste Integer
	- 0

Expected result	Ved oprettelsen af et bankroll-objekt er balancen 0 (se TC01). Ved de værdier, der ikke relaterer sig til højeste og laveste integer, forventer vi derfor bare at balancen ændres med den angivne mængde. Ved de værdier, der relaterer sig til højeste og laveste integer, er vi gået ud fra, at vi har implementeret det dovent, da det højst sandsynligt ikke er teoretisk muligt at komme op på de værdier i et spil alligevel.
Actual result	Hypotesen passer i alle tilfælde.
Status	Bestået
Tested by	Gruppen

Date	23-11-2018
Test environment	En JUnit-test i intelij LENOVO LEGION Y520

Test case ID	TC03
Summary	Integrationstest af TurnLogic og Game. Tester at en spiller køber et felt, hvis det er ledigt.
Requirements	K5
Preconditions	Et spil er startet med 2 spillere (starter hver med 20 penge). Burger Bar ejes ikke af nogen. Spiller 1 slår 1 med terningen og lander på “Burger bar” (pris: 1,-).
Postconditions	“Burger bar” ejes nu af spiller 1. Spiller 1 har nu 19 penge. Spiller 2 har stadig 20 penge.
Test procedure	JUnit test, der instantierer et Game med to spillere og en “mockDiceCup”, der giver et forudbestemt terningeslag. Tjekker først, om Burger Bar ejes af nogen. Instantierer og kører derefter en ny tur for spiller 1, og tjekker spillerens balance og hvem, der ejer Burger Bar.
Test data	String[] players = {“player1”, “player2”}; Int[] predeterminedDiceValues = {1};
Expected result	Burger Bar ownerIndex før turen: -1 (indikerer ingen ejer). Player 1 har balance: 19 Player 1 har balance: 20 Burger Bar ownerIndex efter turen: 0
Actual result	Burger Bar ownerIndex før turen: -1 Player 1 har balance: 19 Player 1 har balance: 20 Burger Bar ownerIndex efter turen: 0
Status	Bestået
Tested by	Gruppen
Date	29-11-18
Test environment	MacBook Pro mid-2012, OSX 10.10.

Test case ID	TC04
Summary	Integrationstest af TurnLogic og Game. Tester at en spiller, der lander på et felt, der ejes af en anden spiller, betaler leje til ejeren.
Requirements	K6
Preconditions	Et spil er i gang med 2 spillere. Spiller 2 ejer “Burger Bar”. Spiller 1 lander på Burger Bar (leje: 1,-)
Postconditions	Spiller 1 har 1,- mindre og spiller 2 har 1,- mere i sin pengebeholdning. Spiller 2 er stadig ejeren af Burger Bar.
Test procedure	JUnit test, der instantierer et Game med to spillere og en “mockDiceCup”, der giver et forudbestemt terningeslag. Burger Bar sættes til at være ejet af spiller 2. Instantierer og kører derefter en ny tur for spiller 1, og tjekker nu spillernes pengebeholdninger og hvem, der ejer Burger Bar.
Test data	String[] players = {“player1”, “player2”}; Int[] predeterminedDiceValues = {1};
Expected result	Player 1 har balance: 19 Player 2 har balance: 21 Burger Bar har ownerIndex: 1
Actual result	Player 1 har balance: 19 Player 2 har balance: 21 Burger Bar har ownerIndex: 1
Status	Bestået
Tested by	Gruppen
Date	29-11-18
Test environment	MacBook Pro mid-2012, OSX 10.10.

Test case ID	TC05
Summary	Integrationstest af TurnLogic og Game. Tester at en spiller, der lander på et felt, som spilleren selv ejer, har en uændret balance.
Requirements	K7
Preconditions	Et spil er i gang med 2 spillere. Spiller 1 ejer “Burger Bar”.

	Spiller 1 lander på Burger Bar (leje: 1,-)
Postconditions	Spiller 1 og 2 har samme pengebeholdning som før.
Test procedure	JUnit test, der instantierer et Game med to spillere og en “mockDiceCup”, der giver et forudbestemt terningeslag. Burger Bar sættes til at være ejet af spiller 1. Instantierer og kører derefter en ny tur for spiller 1, og tjekker nu spillernes pengebeholdninger.
Test data	String[] players = {“player1”, “player2”}; Int[] predeterminedDiceValues = {1};
Expected result	Player 1 har balance: 20 Player 2 har balance: 20
Actual result	Player 1 har balance: 20 Player 2 har balance: 20
Status	Bestået
Tested by	Gruppen
Date	29-11-18
Test environment	MacBook Pro mid-2012, OSX 10.10.

Test case ID	TC06
Summary	Integrationstest af TurnLogic og Game. Tester at en spiller, der lander på startfeltet, modtager 2,-.
Requirements	K8, K9
Preconditions	Et spil er i gang med 2 spillere. Spiller 1 har 20 penge, står et par felter før startfeltet, slår med terningen og lander på startfeltet.
Postconditions	Spiller 1 har nu 22 penge.
Test procedure	JUnit test, der instantierer et Game med to spillere og en “mockDiceCup”, der giver et forudbestemt terningeslag. Spillers 1 position ændres til to felter før startfeltet. Instantierer og kører derefter en ny tur for spiller 1, og tjekker nu spiller 1’s pengebeholdning.
Test data	String[] players = {“player1”, “player2”}; Int[] predeterminedDiceValues = {2};

	Spiller 1's position: 22 (feltet før start er 23, og start er 0).
Expected result	Player 1 har balance: 22
Actual result	Player 1 har balance: 22
Status	Bestået
Tested by	Gruppen
Date	29-11-18
Test environment	MacBook Pro mid-2012, OSX 10.10.

Test case ID	TC07
Summary	Integrationstest af TurnLogic og Game. Tester at en spiller, der passerer startfeltet, modtager 2,-.
Requirements	K8, K9
Preconditions	Et spil er i gang med 2 spillere. Spiller 1 har 20 penge, står et par felter før startfeltet, slår med terningen og passerer startfeltet.
Postconditions	Spiller 1 har nu 22 penge.
Test procedure	JUnit test, der instantierer et Game med to spillere og en "mockDiceCup", der giver et forudbestemt terningeslag. Spillers 1 position ændres til to felter før startfeltet. Instantierer og kører derefter en ny tur for spiller 1, og tjekker nu spiller 1's pengebeholdning.
Test data	String[] players = {"player1", "player2"}; Int[] predeterminedDiceValues = {3}; Spiller 1's position: 22 (feltet før start er 23, og start er 0).
Expected result	Player 1 har balance: 22
Actual result	Player 1 har balance: 22
Status	Bestået
Tested by	Gruppen
Date	29-11-18
Test environment	MacBook Pro mid-2012, OSX 10.10.

Test case ID	TC08
Summary	Integrationstest af TurnLogic og Game. Tester at en spiller kan rykke frem på brættet og fortsætte fra samme position i den næste tur.
Requirements	K12
Preconditions	Et spil er startet med 2 spillere. Spiller 1 slår først 3 og dernæst 5 med terningen.
Postconditions	Spiller 1 står nu på det ottende felt på brættet (startfeltet ikke talt med).
Test procedure	JUnit test, der instantierer et Game med to spillere og en “mockDiceCup”, der giver to forudbestemte terningeslag. Kører to ture for spiller 1 og tjekker spiller 1’s position.
Test data	String[] players = {“player1”, “player2”}; Int[] predeterminedDiceValues = {3, 5};
Expected result	Player 1’s position: 8
Actual result	Player 1’s position: 8
Status	Bestået
Tested by	Gruppen
Date	29-11-18
Test environment	MacBook Pro mid-2012, OSX 10.10.

Test case ID	TC09
Summary	Integrationstest af TurnLogic og Game. Tester at brættet har 24 felter, samt at en spiller kan opnå en række forskellige udfald, alt efter hvilke felter, spilleren lander på.
Requirements	K12
Preconditions	Et spil er startet med 2 spillere. Spiller 1 lander først på et felt, der ikke ejes af nogen, og køber det. Spiller 1 lander da på et felt, der ejes af nogen, og betaler leje. Spiller 1 lander da på et chancefelt.

	Spiller 1 lander da på “besøg i fængslet”. Spiller 1 lander da på “gratis parkering”. Spiller 1 lander da på “gå i fængsel”.
Postconditions	Spiller 1 har udløst 6 forskellige udfald, der kan registreres.
Test procedure	JUnit test, der instantierer et Game med to spillere og en “mockDiceCup”, der giver seks forudbestemte terningeslag. Kører seks ture for spiller 1 og tjekker hvilken string, der returneres som “udfald”.
Test data	String[] players = {“player1”, “player2”}; Int[] predeterminedDiceValues = {1, 1, 1, 3, 6, 6};
Expected result	Udfald: “boughtOwnable”, “paidRent”, “chance”, “jailOnVisit”, “refuge”, “gotojail”.
Actual result	Udfald: “boughtOwnable”, “paidRent”, “chance”, “jailOnVisit”, “refuge”, “gotojail”.
Status	Bestået
Tested by	Gruppen
Date	29-11-18
Test environment	MacBook Pro mid-2012, OSX 10.10.

Negative test

Test case ID	TC10
Summary	Forsøg på at crashe programmet ved indtastning af forskellige navne
Requirements	
Preconditions	Nyt spil startes og spillernavne skal indtastes.
Postconditions	Crashet program
Test procedure	Forskellige navne, der kunne tænkes at have sjove effekter, indtastes og et par ture gennemføres.

Test data	Null, void, /n, /n lol, , “”, “ “, 1111111111111111111111111111111111111111111111111111 1111111111111111111111111111111111111111111111111111 1111111111111111111111111111111111111111111111111111 1111111111111111111111111111111111111111111111111111 1111111111111111111111111111111111111111111111111111 1111111111111111111111111111111111111111111111111111 11111111111111111111111111111111111
Expected result	Spillet chrasher
Actual result	Spillet crasher ikke, men det meget lange navn soft-locker det, da knapperne til turmenuen forsvinder
Status	50 % bestået. På trods af manglende implementering af input-validering er det ikke nemt at få programmet til at gå ned, fordi GUI’en fungerer som doven input-validering.
Tested by	Gruppen
Date	28-11-2018
Test enviroment	IntelliJ

Traceability matrix

	K1	K2	K3	K4	K5	K6	K7	K8	K9	K10	K11	K12	K13	K14	K15
TC01															
TC02															
TC03					x										
TC04						x									
TC05							x								
TC06								x	x						
TC07								x	x						
TC08												x			
TC09												x			
TC10															

Brugertest

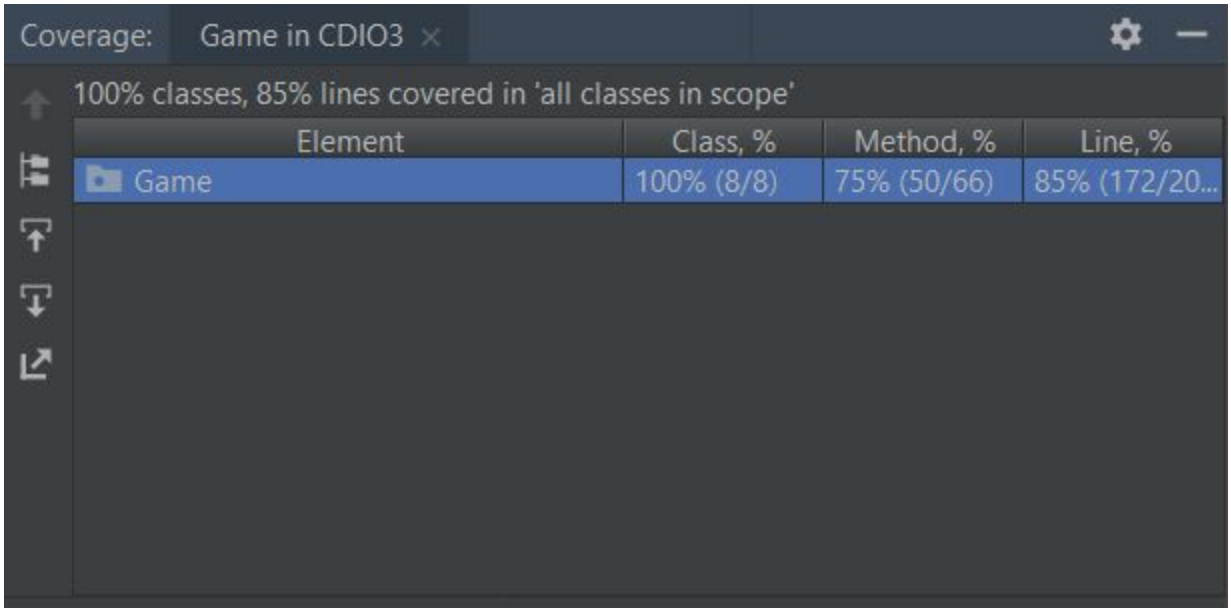
Vi har haft en studerende fra KU til at prøve vores monopolyspil og teste de forskellige funktioner og muligheder, der er i spillet (en beta test). Under testen blev der fundet nogle fejl i gui'en der primært omfatter tekstfejl og en fejl i koden. De forskellige fejl er beskrevet i følgende tabel.

Fejl nr.	GUI	Kode	Mulig løsning
1	Swimmingpool-teksten bliver printet som "Swimmin-\npool" når man lander på feltet		Her er problemmet linjeskift funktionen -\n der ikke fungerer korrekt. Den skifter linje, men den printer også -\n som tekst
2	Legetøjsbutikken-teksten bliver printet som "Legetøjs-\nbutikken" når man lander på feltet		Se fejl 2
3	Vandland-teksten bliver printet som "vand-\nland" når man lander på feltet		Se fejl 2
4	Strandpromenade-teksten bliver printet som "Strand-\npromenadel" når man lander på feltet		Se fejl 2
5		Chancekort registreres i GUI'en, men har ingen funktion.	Vi har valgt ikke at implementere chancekort, så der er en konflikt der skal løses.
6		Når begge grunde i samme farvekategori ejes af samme spiller, bliver der ikke betalt dobbelt husleje af den spiller, der lander på grundene (K16).	Der mangler en funktion.
7	Der er ingen information om, hvordan man tjekker ejerne af de forskellige grunde.		Tilføje tekst til regelsættet.
8	Der mangler information om, hvilken		GUI-baseret, evt.

	spillerbrik, der hører til hvilken spiller fra start af.		Navn på brikker eller bare vise info før tur et.

GUI'en er let at forstå og meget brugervenlig. På trods af de små fejl og mangler, der er. Fejlende påvirker ikke den overordnede brugeroplevelse. Der blev desuden lagt mærke til flotte farver og god navngivning på butikker/grunde.

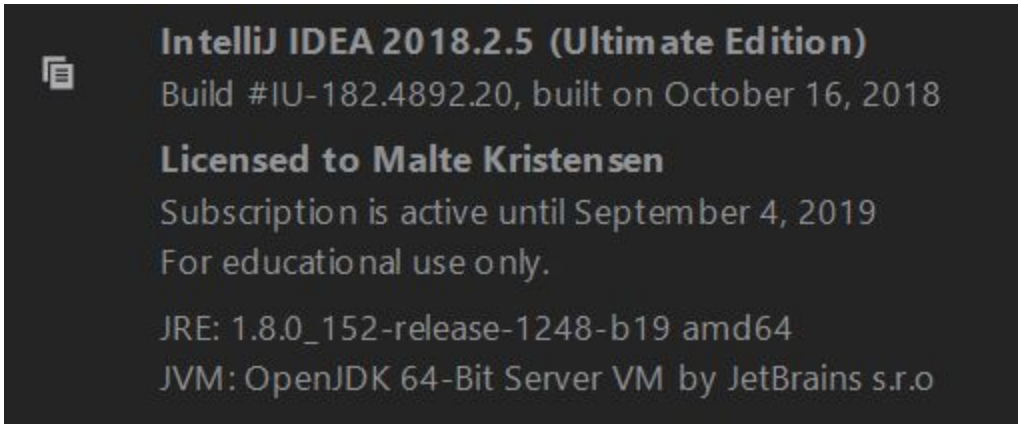
Codecoverage
Vi har kun testet gamepackage da de andre pakker er GUI og controllers. Tilgengæld har vi testet 100% af klasserne og 75% metoderne og 85% af kodelinjerne.



Figur 7: Code coverage rapport for samtlige test.

Konfiguration

Platform
Vores udviklingsplatform samt Java versionsnummer kan aflæses i nedenstående figur. Desuden inkluderer platformen forskellige styresystemer, herunder Windows 8.1, Windows 10 Home og MacOs Mojave.



Figur 8: IntelliJ.

Anvendelse af programmet
Programmet kan anvendes på to måder. Begge kræver Java Runtime Environment installeret.

Den første metode er med anvendelse af github og IntelliJ. Første skridt er at klone projektet til et lokalt repository. Dette gøres ved at navigere til det eksterne repository på github.com. Derefter skal URL'en, der henviser til den git-mappe, der anvendes til kloning, kopieres. Rent praktisk skal der klikkes på "clone or download"-knappen, hvorefter linket kan kopieres. Dette link kan nu anvendes, når man i IntelliJ trykker på file -> new -> project from existing sources. Projektet findes nu lokalt, og er åbent i IntelliJ. Herefter kan selve programmet køres, ved at køre Main-metoden.

Den anden metode er hverken afhængig af IntelliJ eller github, men er en måde, hvorpå programmet kan køres direkte fra en .jar-fil. Programmet leveres som en mappe, der indeholder en batch-fil (.bat). Programmet kan køres via denne fil, dog kræver det et windows-styresystem. Alternativt kan mappen lokaliseres og programmet køres via kommandolinjen og kommandoen "java -jar 45_CDIO3.jar"

Kildekoden kan desuden downloades som zip fil fra github, denne skal pakkes ud på computeren.

Konklusion

Vi har lavet et juniormatadorspil til IOOuterActive. Via forskellige UML-artefakter fandt vi frem til, hvordan spillet skulle kodes. Efterfølgende testede vi nogle af metoderne i spillet.

Det er et simpelt spil, der i dets nuværende form kan betragtes som i en slags alpha-version, selvom vi har lavet en enkelt beta brugertest med en potentiel slutbruger. Der er lavet en enkelt brugertest, der viste nogle ting, der kunne arbejdes videre på. Desuden har vi valgt at droppe nogle krav undervejs, da vi skulle prioritere vores tid i opgaven. Med andre ord er der nogle

krav, der på nuværende tidspunkt i udviklingsprocessen, ikke er opfyldt. Det er bl.a. funktionerne chancekort, at man kunne komme i fængsel, og at man skulle betale dobbelt husleje, når begge grunde i samme farvekategori ejes af samme spiller.

Referencer

Larman, C. (2004) *Applying UML and patterns: an introduction to object-oriented analysis and iterative development.* Upper Saddle River: Pearson Education.

Lewis & Loftus (2011) *Java Software Solutions Foundations of Program Design*, Pearson Education.

Bilag

Spilleregler på dansk og engelsk

Regler på dansk
En til fire spillere får tildelt en brik på spillepladen. Efter tur slår de med en terning og rykker så mange felter frem på brættet, som terningen viser. En del af felterne er grunde, der kan købes, hvis ingen spiller ejer dem i forvejen. De resterende felter sker der ikke noget ved at man lander på i denne version af spillet. Vinderen af spillet er den, der har flest penge, når en af de andre går fallit.

Regler på engelsk
The game is played by two to four players. Each player gets a piece to move on the board. In turns, the players cast the die and move forward as many tiles as the number the die shows. Some of the tiles are plots that can be bought if no player owns them already. The winner is the one with the most money, when one other player is declared bankrupt.