CDIO 3

Gruppe 36

Afleveringsdato 1/12/2017

S175131: Nicolai Øyen Dam 

S175132 Jacob Jørgensen 

S175123 Mads Schultz 

S165526 Nils David Rasamoel 

S175130 Monica Schmidt 

DTU

Timeregnskab

CDIO3							
Time-regnskab	Ver. 2017-12-01						
Dato	Deltager	Analyse	Design	Impl.	Test	Dok.	Ialt
01-12-2017	Mads	3	3	4	3	5	18
01-12-2017	Nicolai	2	5	5	2	4	18
01-12-2017	Monica	3	3	4	3	5	18
01-12-2017	Nils	4	3	6	2	3	18
01-12-2017	Jacob	2	4	5	3	4	18

Indholdsfortegnelse

Figurliste

1. Indledning

Dette projekt handler om at programmere et Monopoly Junior spil ved brug af Java. Spillet vil bestå af en spilleplade med 24 felter, 8 chancekort, en terning og brikker. Der kan være mellem 2-4 spillere i spillet.

Når antal brugere har oprettet hver deres spiller med navn og en bil som brik, bliver spillerne tildelt en pengebeholdning på 20. Herefter slår første spiller med terningen og rykker det antal felter frem, som terningens værdi viser. Hermed afsluttes spillerens tur og næste spillers tur går i gang. Spillet forsætter ved, at spillerne rykker rundt i urets retning på spillepladen.

Spillet afsluttes når en spiller ikke har råd til at betale for et felt, eller ikke kan betale en anden spiller for at lande på spillerens felt. Den spiller med den største pengebeholdning, vil når spillet afsluttes vinde.

Til analyse af projektet anvendes FURPS+ til kravspecifikation. Til at beskrive hvilke funktioner programmet skal indeholde anvendes et use case-diagram og use case-beskrivelser. En domænemodel bruges til at vise sammenhængen mellem programmets analyseklasser og et systemsekvensdiagram vil illustrere interaktionen mellem eksterne aktører og programmet. Disse modeller er med til at en give forståelse af hvad programmet skal kunne, samt hvilke klasser der potentielt vil blive oprettet.

I designfasen vil det blive belyst hvordan spillets kode programmeres. I designfasen lægges der mere fokus på selve programmeringen end i analysen. Ud fra domænemodellen laves et designklasse diagram. Derudover udarbejdes to sekvensdiagrammer over to forskellige scenarier.

Under implementering vil GRASP og overordnede principper af nedarvning blive forklaret. Derefter vil forskellige testscenerier blive udført af programmet, for at sikre en høj kvalitet af koden. Slutteligt vil en konklusion samle op på projektet.

2. Analyse

I det følgende analyseafsnit vil krav, use case diagram, use case beskrivelse, domænemodel samt systemsekvensdiagram blive opstillet. Disse modeller er med at sikre en forståelse af det kommende spil, både for brugere og kunden.

Til dette projekt var kundens vigtigste vision, at spillet kan spilles. Der er derfor blevet foretaget en subjektiv vurdering af krav til programmet, så programmet kan køre.

2.1 Krav

Formålet med at specificere krav for et program er, at klargøre hvad der senere hen skal implementeres. Dette giver en specifikation af programmets egenskaber, hvad programmet skal tilbyde brugeren samt programmets afgrænsning.

Til at opstille en kravliste over Monopoly Junior, vil modellen FURPS+ blive anvendt. FURPS+ udspecificerer funktionelle- samt ikke-funktionelle krav. Kravlisten har til formål at belyse hvilke krav programmet skal opfylde.

2.1.1 Funktionelle krav

Disse funktionelle krav beskriver hvad Monopoly Junior spillet skal kunne.

Beskrivelse
1. Alle spillere skal hver gang det er deres tur kaste med en terning, terningen giver en tilfældig værdi mellem 1-6.
2. Spillepladen skal bestå af 24 felter.
3. Hver gang man lander på et felt, skal man følge feltets anvisning.
4. Lander man på et felt, der ikke ejes af en anden spiller, er man tvunget til at købe feltet. (Se feltliste i afsnit 2.1.3)
5. Lander man på et felt, som en anden spiller ejer, skal man betale ejeren den angivne pris. (Se feltliste i afsnit 2.1.3)
6. Lander man på et Chancekort trækker man et tilfældigt chancekort og følger kortets anvisning (Se Chancekort liste i afsnit 2.1.4)
7. Spilleren skal ved næste slag, starte hvor spilleren landede ved sidste slag, således man kommer til at gå i ring på spillepladen.
8. Programmet skal kunne spilles mellem 2-4 spillere.
9. Programmet skal hele tiden oplyse spillernes pointbalance.
10. Alle spillere starter med en balance på 20kr.
11. Spillet slutter når en spiller er gået bankerot. (0 kr. i balancen eller under) Vinderen er den med flest penge i pengebeholdning.
12. Hvis spilleren lander på feltet ”Gå i fængsel”, skal personen rykkes til feltet ”Fængsel” og personen bliver sprunget over i næste runde.
13. Hvis en spiller passerer start, modtager vedkommende 2 kr.
14. Hvis en spiller er ejer af feltet, er hans farve markeret rundt om feltet

2.1.2 Ikke funktionelle krav

Disse krav beskriver de tilhørende ikke-funktionelle krav.

Type	*Beskrivelse*
Useability	N/A
Reliability	N/A
Performance	12. Åbning af programmet skal have en lav responstid ved varm start (<0,333s).
Supportability	13. Koden skal være på dansk.
Implementation	14. Spille skal være på dansk.
Interface	15. Systemet skal fungere på computere med Windows 10, hvor den nyeste version af Java er installeret.
Operations	N/A
Packaging	16. Programmet vil ikke have en fysisk indpakning.

2.1.3 Feltliste

En oversigt af hvilke felter der er på pladen og hvilke forskellige handlinger felterne medfører.

Type	*Handling*
Felt 1: Start	Man modtager 2kr. hver gang man passerer start.
Felt 2: Burgerbar	1kr.
Felt 3: Pizzahuset	1kr.
Felt 4: Chancekort	Se mulige udfald under 2.1.4 Chancekort.
Felt 5: Slikbutikken	1kr.
Felt 6: Iskiosken	1kr.
Felt 7: Fængslet	Intet sker.
Felt 8: Museet	2kr.
Felt 9: Biblioteket	2kr.
Felt 10: Chancekort	Se mulige udfald under 2.1.4 Chancekort.
Felt 11: Skaterparken	2kr.
Felt 12: Swimmingpoolen	2kr.
Felt 13: Gratis parkering	Intet sker.
Felt 14: Spillehallen	3kr.
Felt 15: Biografen	3kr.
Felt 16: Chancekort	Se mulige udfald under 2.1.4 Chancekort.
Felt 17: Legetøjsbutikken	3kr.
Felt 18: Dyrehandelen	3kr.
Felt 19: Gå i fængslet	Flyt brikken over til "Fængslet".
Felt 20: Bowlinghallen	4kr.
Felt 21: Zoo	4kr.
Felt 22: Chancekort	Se mulige udfald under 2.1.4 Chancekort.
Felt 23: Vandlandet	4kr.
Felt 24: Strandpromenaden	4kr.

2.1.4 Chancekort

Her vises de 8 chancekort der er implementeret i spillet. Det vurderes at 8 chancekort er passende, da kortene genbruges. Der er oprettet forskellige typer af chancekort, så man både kan få penge, miste penge eller skal rykke hen til et bestemt felt.

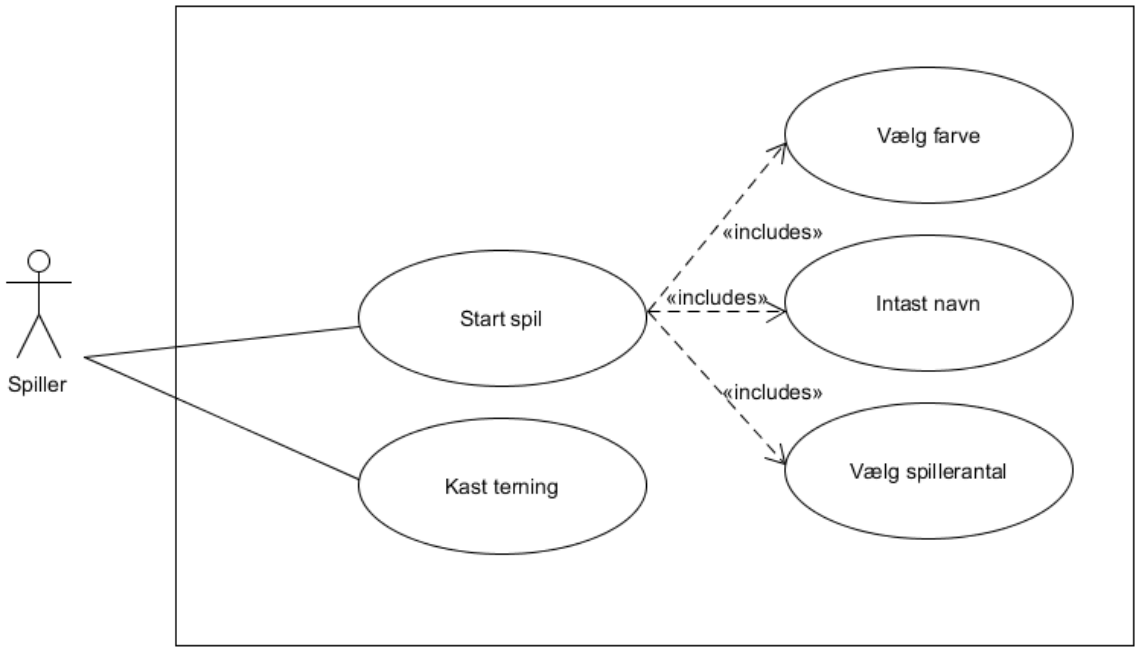
Type	*Handling*
Chancekort 1	Betal 3kr. til banken.
Chancekort 2	Flyt hen på ZOO feltet.
Chancekort 3	Betal 4kr. til banken.
Chancekort 4	Flyt hen på Spillehallen
Chancekort 5	Du ryger i fængsel.
Chancekort 6	Du får 1kr.
Chancekort 7	Du får 2kr.
Chancekort 8	Du får 4kr.

2.2 Use case

En use case er en beskrivelse af, hvordan en aktør benytter systemet til at få opfyldt et mål. For eksempel kan en aktør have et mål om, at kaste en terning.

2.2.1 Use case diagram

Ved at opstille use cases i et use case diagram kan man se relationerne mellem aktører og use cases. I vores use case diagram er der to base use cases, som er ”Start spil” og ”Kast terning”. Use casen ”Start spil”, har yderligere tre include use cases, som er “Vælg farve”, “Indtast navn” og “Vælg spillerantal”. Disse tre include cases indgår i systemets base use case ”Start spil”, da man først skal vælge antal spiller, navn og farve inden spillet kan starte.



Figur 1: Use case diagram over Monopoly Junior

2.2.2 Use case brief-beskrivelse

I følgende afsnit er en brief use case beskrivelse opstillet. En kort brief use case, anvendes til at beskrive alle systemets use cases kort, hvilket er med til at give et overblik, over de forskellige use cases betydning i praksis.

Use cases	*Brief beskrivelser*
Kast terning (base)	Den pågældende spiller kaster med terningen, så vedkommende kan få information omkring hvor meget spilleren skal rykke.
Start spil (base)	For at spilleren skal kunne starte spillet skal spilleren igennem et par use cases inden spillet kan starte. De er beskrevet i includes.
Vælg spillerantal (include)	Spilleren vælger antal spillere i spillet.
Indtast navn (include)	Spilleren indtaster sit navn.
Vælg farve (include)	Spilleren vælger farve.

2.2.3 Use case fully dressed-beskrivelse

I dette afsnit udarbejdes en fully dressed use case med udgangspunkt i systemets mest centrale use case ”Kast terning”. Med en fully dressed use case beskrivelse kommer man i dybden med use case. Grunden denne use case er valgt, skyldes at det er systemets vigtigste, da hele spillet afhænger af, at man kaster terningen. Denne handling afleder alle andre handlinger i systemet.

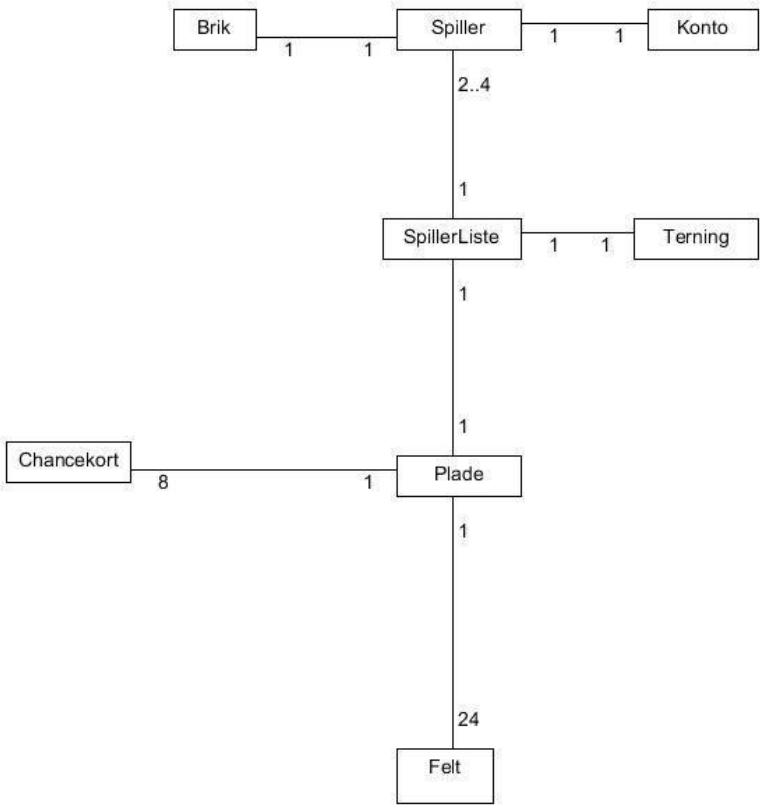
Use case sektion	*Kommentar*
Use case navn	Kast terning
Scope	Monopoly Junior
Level	Det skal lykkedes for den pågældende spiller at kaste med en enkelt terning, få terningens værdi, så spillerens brik kan rykke.
Primær aktør	Spiller
Interessenter	N/A
Forudsætninger	Det forudsættes at spillet er startet, før spilleren kan kaste med terningen.
Succeskriterier	Terningen kastes og der vises en tilfældig værdi mellem 1-6. Spillebrikken vil blive rykket frem og udskriver det, der skal udskrives for det pågældende felt. Kontoen opdateres og turen går videre.
Hoved succeskriterier	1. Spiller kaster med terning. 2. Terningens værdi vises. 3. Spilleren får rykket sin brik frem. 4. Hvis en spillers pengebeholdning kommer på eller under 0 har han tabt.
Udvidelser	1.A) Spilleren kan ikke kaste med terningen. 2.A) Terningens værdi vises ikke, eller vises ikke i det korrekte interval. 3.A) Spillerens konto opdateres ikke korrekt, ud fra det pågældende felts værdi. 4.A) Hvis en spillers pengebeholdning kommer under 0 og personen stadig kan spille videre.
Specielle krav	Specielle krav: • Når spilleren kaster med terningen, må der max være en responstid på 0,33 sekunder. • Terningens værdi skal være imellem 1-6. • En spillers konto opdateres afhængigt af det pågældende felt, som spilleren lander på. • Hvert felt har en form unik beskrivelse af hvad der skal ske med spilleren pengebeholdning enten positiv eller negativ. Et felt kan dog godt være neutralt, så der ikke sker noget.

Teknologi- og dataliste	2.A) For at generere tilfældige værdier mellem 1 til 6, anvendes Math.random fra Math klassen i java.lang pakken. 2.A) For at vise en terning og plade, anvendes der en GUI. 2.A) Til at give pladens felter farve er farve klassen, java.awt.color anvendt.
Optræden	"Kast terning" er en vigtig use case i forbindelse med spillet, da kast terninger udleder de andre handlinger i programmet. Denne use case fremkommer derfor mange gange, i løbet af et spil, da spillerne skal slå, hver gang det er deres tur. Det er svært at forudse hvor mange gange der skal kastes med terningen, før en spiller har vundet. Det forudsættes, at use casen "Kast terning" fremkommer flere gange når der er fire spillere, end hvis der er to spillere.

2.3 Domænemodel

For at sikre en fællesforståelse af spillets domæne med kunden, er en domænemodel udarbejdet, som viser analyseklasser, der er fundet via en navneordsanalyse.

Der er en spiller-klasse, som håndterer informationer om de forskellige spillere. Spiller-klassen er koblet til en brik- og konto-klasse, da en spiller har én brik og én konto. Yderligere er spiller-klassen koblet med en SpillerListe-klasse, hvor én SpillerListe kan indeholde 2-4 spillere. SpillerListe-klassen er koblet sammen med én Terning- og én Plade-klasse. Plade-klassen er koblet på en felt- og chancekort-klasse. En plade kan have 8 chancekort og 24 felter.

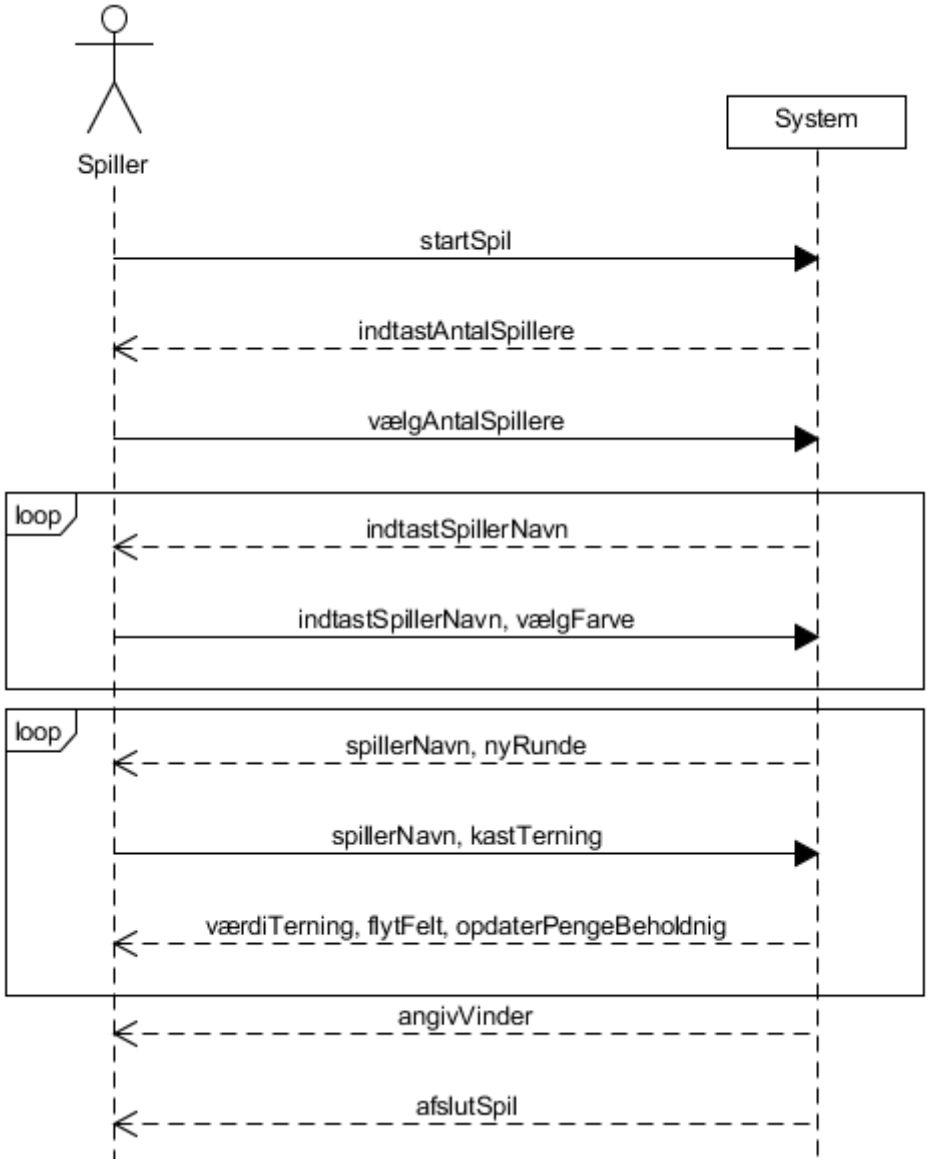


Figur 2: Domænemodel over Monopoly Junior

2.4 Systemsekvensdiagram

Et systemsekvensdiagram har til formål at vise interaktioner mellem aktøren og systemet, ved at illustrerer systemets inddata- og uddatahændelser. Et systemsekvensdiagram viser, hvad systemet gør og ikke hvordan, ud fra ét bestemt scenarie. Hvis der indgår flere systemer, kan et systemsekvensdiagram ligeledes vise interaktionen mellem disse.

Der er foretaget er systemsekvensdiagram for hele vores projekts system. I spillet Monopoly Junior, startes spillet som det første. Herefter spørger systemet brugerne hvor mange spillere der er. Hver spiller udfylder herefter sine oplysninger i et loop, efter første loop, starter et nyt loop, så alle spillerne kan indtaste deres individuelle oplysninger. Når sidste spiller har indtastet sine oplysninger, går systemet ud af loopet. Dernæst startes et nyt loop, som skifter mellem spillernes tur, som forsætter indtil den første spiller er gået fallit. Hermed vil loopet afsluttes, en vinder vil blive fundet og spillet afsluttes.



Figur 3: Systemsekvensdiagram over Monopoly Junior

3. Design

I dette afsnit arbejdes der med design, som en proces der har til formål at vise hvordan systemet skabes. Programmeringen har stor betydning i dette afsnit.

I designafsnittet vil der udarbejdes to forskellige sekvensdiagrammer, til to forskellige scenarier. Derefter udarbejdes et designklassediagram, som beskriver de klasser der implementeres i systemet og hvordan de interagerer med hinanden.

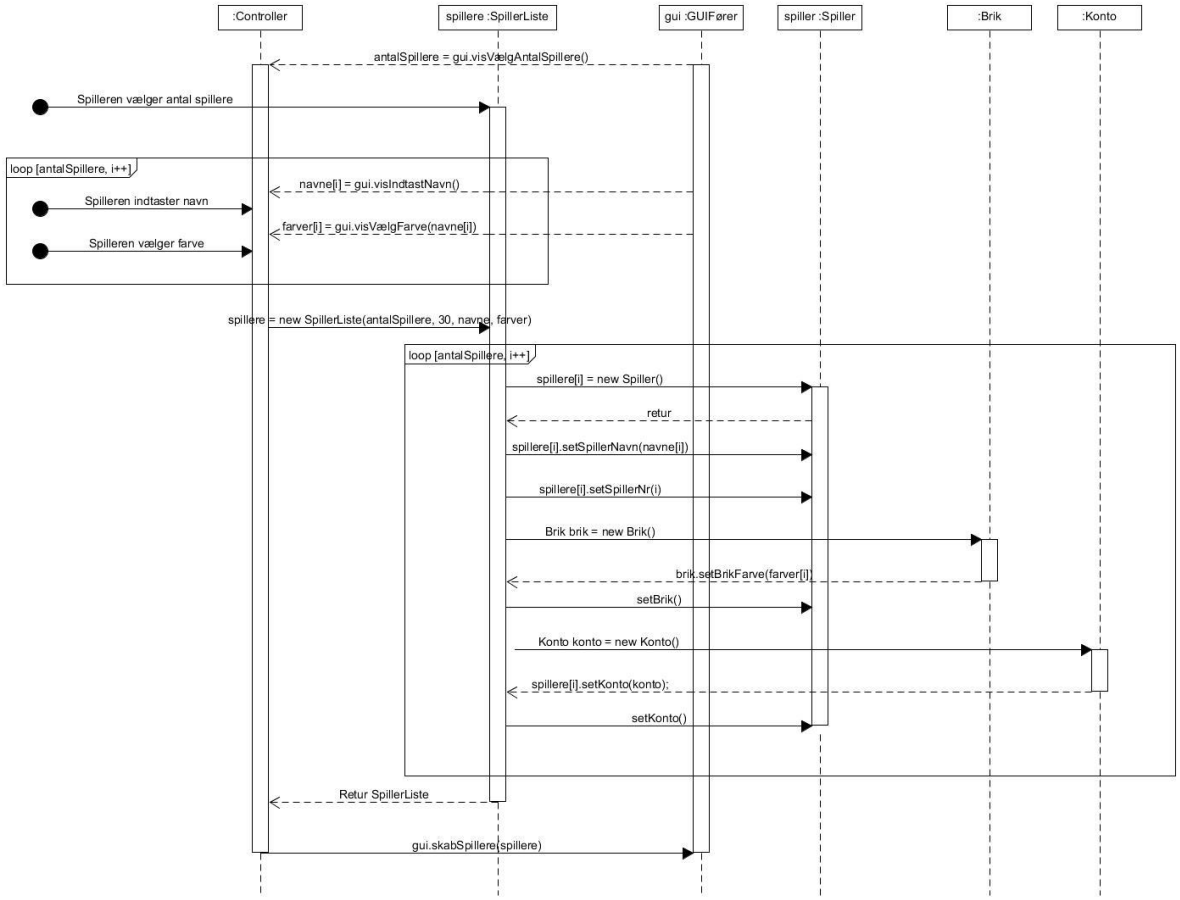
3.1 Sekvensdiagram

Et sekvensdiagram har til formål at vise hvordan objekter kommunikerer via metodekald. Det er en mere detaljeret beskrivelse, sammenlignet med systemsekvensdiagrammet, der kun giver et overordnet indblik i, hvad der forekommer mellem aktøren og systemet.

3.1.2 Scenarie 1: Fra brugeren vælger antal spillere til spillerne er skabt i GUI'en

Scenarie 1 viser hvordan objekterne kommunikerer når spillerne oprettes i systemet. Først vises en besked, hvor brugeren bliver bedt om at vælge antal spilere, herefter kommer en handling udefra, hvor brugeren vælger antal spillere. Dernæst startes et loop, hvor hver spiller indtaster navn og vælger en farve på deres spillebrik, GUIFører returnerer navn og farve visuelt, loopet afsluttes. Når spillernes farve og navn er oprettet, opretter Controller et spillerListe objekt med farve, navn og antal spillere. Herefter oprettes et nyt loop, hvori SpillerListe opretter et Spiller objekt, hvor navn og spillernummer tilføjes til spilleren. SpillerListe opretter et Brik objekt, og sætter brikkens farve. Herefter opretter SpillerListe et Konto objekt, som sætter kontoens pengebeholdning. Brikken og kontoen sættes til spilleren.

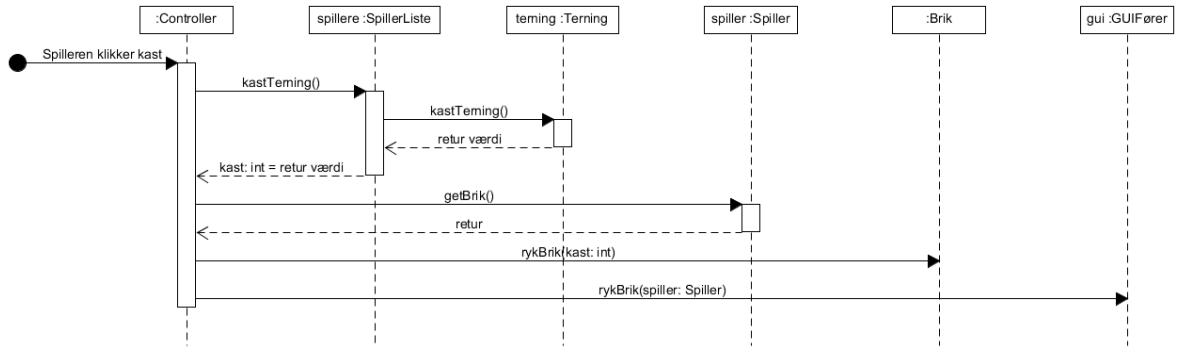
Når alle spillere har været igennem loopet afsluttes det, og Controller tager i mod SpillerListe objektet. Herefter skabes spillerne i GuiFører.



Figur 4: Sekvensdiagram for scenarie 1

3.1.3 Scenarie 2: Fra spilleren klikker tast til brikken er rykket.

Dette scenarie giver et indblik i hvordan klassen *Controller* interagerer med objektet *spiller* af klassen *Spiller,* for at få fat i den relevante spillerbrik. *Controller-klassen* kalder metoden *getBrik()* i objektet *spiller* af *Spiller-klassen* som efterfølgende returnere den pågældende spillers brik.



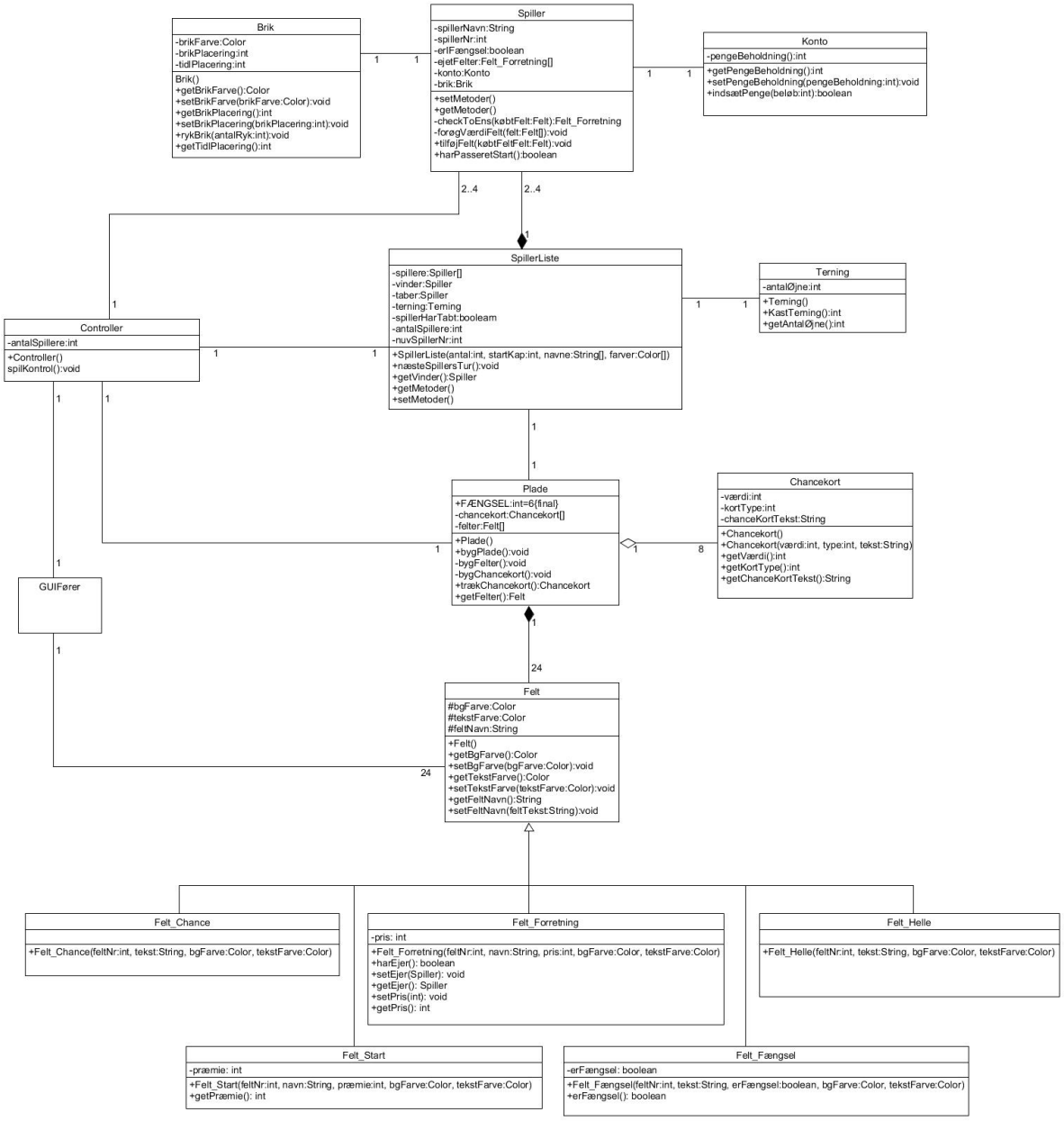
Figur 5: Sekvensdiagram for scenarie 2

3.2 Designklassediagram

Et designklassediagram er en opstilling af hvordan det endelige programs klasser agerer i relation til hinanden. Her inddrager man attributter, den øverster del af hver af klasserne og metoder, den nederste del af hver af klasserne.

I designklassediagrammet er *get*- og *set*-metoder udeladt, da det er underforstået at de indgår i klasserne. Eksempelvis fremgår det, at *SpillerListe.class* bruger en public konstruktør med en række tilhørende parametre *SpillerListe(antal:int, startKap:int, navne:String[], farver:Color[])*. Disse oplysninger fra designklassediagrammet kan være behjælpelig til at skrive koden, da det giver et godt overblik over hvad man skal kode, såfremt man opdatere den pr. iteration.

Man skal bemærke, at der er en relation mellem *Spiller* og *Felt_Forretning,* som er udeladt i designklassediagrammet. Dette skyldes, at der laves to metoder *getEjer* og *setEjer* i *Felt_Forretning,* som tager i mod et *Spiller-objekt.*



Figur 6: Designklassediagram over Monopoly Junior

4. Implementering

I dette afsnit kommenteres der på relevante emner i forhold til programmets kode. Arv vil blive kommenteret, da nedarvning bruges i feltklasserne, samt vil abstrakte metoder og klasser blive forklaret.

Yderligere vil det blive forklaret hvad der sker, når man eksempel har en metode, landOnField, i de nedarvede klasser og hver klasses landOnField metode, gør noget forskelligt i den respektive klasse. Til sidst vil programmet kode blive kommenteret, i forhold til GRASP-mønstre.

4.1 Arv

I Java kan en klasse arve variabler og metoder fra en anden klasse, hvilket gør det muligt at genbruge programkoder. En klasse kan kun have én superklasse, men alle klasser kan have et ubegrænset antal af subklasser.

Arv fungerer ved, at man tager en eksisterende klasse og udbygger på klassen med andre og/eller flere metoder og variabler. Eksempelvis, man kan have en superklasse, klasse Alpha, denne klasse har en sub-klasse, klasse Beta. Klasse Beta kan nedarve variabler og metoder fra klasse Alpha, nedarvning kører kun nedad og ikke opad. Hvilket vil sige at klasse Alpha ikke kan arve fra klassen Beta.

I vores system er der brugt nedarvning fra klassen *Felt* til klasserne *Felt_Chance, Felt_Forretning, Felt_Fængsel, Felt_Start og Felt_Helle*, hvilket betyder at *Felt* er super-klassen og de nedarvede klasser er subklasser.

Grunden til at nedarvning anvendes i programmets kode er, at spillepladen har felter med forskellig handling, men felterne har samtidig nogle karakteristika som er ens. Fælles for sub-klasserne er Instansvariablerne *feltNavn*, *tekstFarve* og *bgFarve,* der alle hentes ned fra super-klassen *Felt.* Nedarvning gør det muligt at specialiserer en klasse, samtidig med at gøre det muligt at genbruge samme funktionalitet flere steder.

4.2 Abstrakt

I arv kan det ofte være relevant at erklære en superklasse for abstrakt. Dette gøres hvis man ikke ønsker at klassen skal instantieres, man kan dermed erklære abstrakte metoder. Hvis en klasse ikke er abstrakt, kan den ikke indeholde abstrakte metoder, men en erklæret abstrakt klasse, kan godt indeholde metoder, der ikke er abstrakte.

En abstrakt klasse defineres ved at skrive *abstract* i klassedefinitionen, for eksempel: *public abstract KlasseNavn*. Når en klasse er abstrakt kan den indeholde abstrakte metoder, hvilket er metoder der ikke har af en ”metodekrop”, hvilket vil sige at metoden ikke har nogle { }. Eksempelvis kan en abstrakt metode skrives ved: *public abstract void testMetode();*

Grunden til at det kan være nyttigt at anvende abstrakte metoder er, at man tvinger subklasserne til at overskrive den abstrakte metode, der er i superklassen. Dette bruger man, når man er sikker på, at alle subklasser skal indeholde samme metode, men metoden skal gøre noget forskelligt i hver subklasse. En abstrakt metode, kan derfor ses som et løfte, som går ud på, at man ”lover” at man overskriver metoden i sub-klasserne, dette gør man ved at bruge *@Overriding*.

Et eksempel på en abstrakt superklasse kan være *Figur*, som har subklasserne *Cirkel* og *Firkant*. Der kan oprettes en abstrakt metode i *Figur* som hedder *Areal*, hermed tvinges man til at overskrive

metoden *Areal,* i Cirkel og Firkant klasserne. Hermed kan man definere forskellige areal-metoder for de to klasser, hvilket er relevant, da udregningerne er forskellige.

I systemets programkode oprettes der ikke nogle instanser af *Felt*, denne klasse er derfor erklæret abstrakt, for at være sikker på, at den ikke instantieres. Denne klasse har nogle metoder, som ikke er abstrakte, da de ikke overskrives af subklasserne. Disse metoder kaldes i klassen *GuiFører* der henter navn, tekstfarve og baggrundsfarve for felterne generelt. Disse metoder kunne godt nedarves til subklasserne, men da metoderne er ens for alle felterne ses dette for unødvendigt. Hvis en af metoderne havde haft forskellige funktioner for hvert felt, kunne det have været smart at overskrive metoden fra superklassen. Eventuel en overskrivning af abstrakt en metode, hvis det havde været relevant.

4.3 LandOnField

Hvis man har metoden *landOnField* i en superklasse og har samme metodenavn i de nedarvede klasser, så overskrives metoden fra superklassen i subklasserne.

Det betyder, at hvis man for eksempel har en superklasse der hedder ”*Person*”. Den har en toString metode, der udskiver personens alder og navn. Klassens *Person* har en nedarvet klasse der hedder ”Studerende”, i den forbindelse kan det være relevant at få udskrevet et studienummer og skolens navn. Dermed over skriver man superklassens toString metode, så den udskriver studienummer og skole navn, i stedet for personen alder og navn. Hvis det stadig ønskes at alder og navnet udskrives, kan man skrive super.toString.

4.4 GRASP

GRASP kan anvendes til at fordele ansvar mellem objekter. Overordnet kan man opdele klasser i tre kategorier. Creator, Information Expert og Controller.

Creator

En Creater-klasse er en klasse, som primært har til formål at oprette objekter. I vores system, har vi klassen *Plade*, som opretter henholdsvis 24 objekter af vores subklasser *Felt_Chance, Felt_Forretning, Felt_Fængsel, Felt_Start og Felt_Helle* og 8 chancekort fra klassen *Chancekort*. Hermed udledes det er klassen *Plade* en creater-klasse.

På samme grundlag er *GuiFører* og *SpillerListe* creators.

Controller

En Controller-klasse er en klasse, som repræsenterer det overordnede system, der sørger for, at alle use cases kører. Klassen *Controller* er en oplagt controller-klasse, eftersom den har til formål at holde spillet kørende, når det er sat i gang.

Man kan lave flere Controller-klasser, men da dette system ikke er omfangsrigt, har vi valgt kun at have en enkelt.

Information Expert

En Information Exptert-klasse er en klasse som holder på informationer.

SpillerListe klassen holder eksempel styr på spillerne, antallet af spillere, hvem taberen bliver og hvem der vinder. Den information kan andre klasser hente og bruges i deres metoder. For eksempel i *GUIfører* har vi en metode *skabSpillere(SpillerListe spillere),* som tager en SpillerListe-parameter. Den henter antallet af spillere med *getAntalSpillere()*. Dette bruges i et for-loop og henter hver enkel instans af *Spiller* med *getSpiller().*

Low Coupling

Low coupling er et begreb, som omhandler afhængighed mellem klasser. En lav afhængighed opnår man ved at tildele ansvar til de forskellige klasser, således der ikke er for mange klasser der påvirker hinanden. Dette vil gøre koden lettere at forstå og ændre i den, eftersom det er begrænset hvor mange steder man skal ændre i den, hvis man vil foretage en ændring.

I vores system har vi forsøgt at holde koblingen lav mellem klasserne. Dette kan man se på vores designklassediagram, hvor koblingerne mellem klasserne er anvist.

High Cohesion

High Cohesion er et begreb, som går ud på, at opdele klassernes funktioner på en smart måde. Man uddelegerer klassernes funktioner, således én klasse kun har én primær funktion. Dette medfører at man ikke ender op med store uoverskuelige klasser.

I vores system har vi har tænkt nøje over, hvordan vi har delt klassernes funktioner op, så klassernes funktioner ikke blandes sammen. Selve teksten til spillet er inddelt i specielle klasser, således at sproget kan ændres. Yderligere skal det nævnes, at pakkerne er inddelt efter deres funktioner i otte forskellige pakker; main, controller, gui, monopoly_junior, monopoly_junior_felter, test, tekst og spillogik. Grunden til vi har gjort dette er for, at holde projektet struktureret og bare overblikket over koden. Dette vil gøre det nemmere for en tredjepartsprogrammør at sætte sig ind i systemet, samt gøre det nemmere at genbruge eller ændre programmets kode.

5. Test

I dette afsnit vil forskellige test cases blive opstillet og testet, som skal sikre at spillet lever op til spillets krav. Dette vil sikres bl.a. ved at udføre Junit test. Derudover foretages en acceptance test til sidst, som sikrer at alle spillets krav er opfyldt.

5.1 Test case: Test af responstid ved varm start

Her testes der om spillet kan starte ligeså hurtigt som det forventes. Testen foretages ved varmstart, dette betyder at brugeren har spillet før. Det er valgt at teste programmet ved varmstart, da koldstart kun er første gang spillet spilles. Måden der testes på er, at vi har en long-variabel som holder på starttidspunktet og så holder en anden long-variabel på sluttidspunktet. De to variabler trækkes fra hinanden og kørselstiden bliver udskrevet.

Test case ID	TC1
Beskrivelse	I denne test case vil vi gerne teste spillet responstid ved varm start.
Krav	En responstid på <0,333 sekunder
Betingelse før	Tiden er 0
Betingelse efter	Tiden er <0,333 sekunder
Test data	-
Forventet	<0,333 sekunder
Faktisk	0,648 sekunder
Resultat	Fejl
Dato	1/12/17

Ud fra overstående resultat, har testen fejlet, fordi den har en responstid på over 0,333 sekunder. Dog er det valgt at acceptere at denne test ikke er succesfuld, da det er acceptabelt at responstiden ligger på lidt over et halvt sekund.

5.2 Test Case: To spillere af samme navn

I et rigtigt Monopoly Junior spil, kan det jo forekomme at man spiller samme med en anden, tredje eller fjerde, som har samme navn som en selv. Derfor testes Monopoly junior spillet for, hvorvidt det kan lade sig gøre at spille spillet med to spillere af sammen navn.

Test case ID	TC2
Beskrivelse	I denne test case, tester vi to spillere af samme navn.
Krav	Programmet kan håndtere to ens navne.
Betingelse før	Spiller 1 har samme navn som spiller 2.
Betingelse efter	Spiller 1 og 2 har indtastet det samme navn.
Test data	-
Forventet	Vi forventer at spillet vi kører videre og komme i gang med spillet.
Faktisk	Spillet køre videre med to spillere med ens navne
Resultat	Succes
Dato	01/12/17

Testen viser, at det kan det lade sig gøre, at hedde det samme i spillet. Altså spiller 1 heder Jens og spiller 2 kan også hedde Jens.

5.3 Junit-test

I følgende afsnit vil Junit testene blive opstillet, udført og kommenteret. Samtidig med Junit testene blev udført, blev der samtidig foretaget code coverage på den pågældende stykke kode. Code coverage er anvendt til at sikre at hele testkoden bliver udført.

5.3.1Test case: Test af setKontoBeholdning

Her testes vores setKontobeholdning metode, da det er væsentligt at kunne indsætte det bestemte beløb som spillerne skal starte med.

Test case ID	TC3
Beskrivelse	Vi tester her om vores metode setKontoBeholdning, giver muligheden for at sætte beholdningen til en bestemt værdi
Krav	Kontobeholdningen skal kunne sættes til den værdi der angives
Betingelse før	Kontoen er 0
Betingelse efter	Kontoen indeholder nu den indtastede værdi
Test data	100
Forventet	100
Faktisk	100
Resultat	Succes
Dato	23/11/17

5.3.2 Test case: Sandsynlighed af terning

Test case ID	TC4
Beskrivelse	I denne test case, er der blevet testet sandsynligheden af, at alle terningens sider bliver slået med en lige fordeling indenfor 10.000 kast.
Krav	At værdierne 1-6 fremkommer lige hyppigt
Betingelse før	Der er slået 0 af hver værdi
Betingelse efter	Hver værdi er slået cirka 1666 gange
Test data	-
Forventet	Vi forventer at hver værdi er slået mellem 1583 og 1749 gange. En afvigelse på 5% er acceptabel.
Faktisk	Alle siderne ramte i det givne interval. Se bilag 9.1
Resultat	Succes
Dato	23/11/17

Nedenfor er en tabel opstillet som viser udfaldene af terningekast. Ud fra tabellen kan det ses at hver værdi af terningens sider, ligger indenfor grænserne.

Terningside	**Test 1, 10.000 kast**	**Test 2, 10.000 kast**
Side1	1.645	1.703
Side2	1.648	1.588
Side3	1.678	1.686
Side4	1.728	1.692
Side5	1.675	1.694
Side6	1.626	1.637

I bilag 9.1 er der vedlagt et udklip af konsollen fra denne test, samt oversigt af code coverage. Bilag 9.1 viser at hele testkode er udført.

5.3.3 Test Case: Chancekort

I et fysisk spil Monopoly, fungerer chancekorttrækning ved at en spiller trækker et kort øverst fra chancekortbunken, derefter lægges spillerens kort nederst i bunken, der er derfor ikke stor chance for at det samme kort trækkes flere gange.

I dette program foregår trækningen af et chancekort tilfældigt, ligesom når man slår med en terning, derfor kan det samme kort forekomme flere gange indenfor et kort interval, i modsætningen til et fysisk Monopoly spil.

Test case ID	TC5
Beskrivelse	I denne test case, er trækChancekort metoden testet om den fungerer korrekt.
Krav	At der fremkommer forskellige chancekort.
Betingelse før	Der er ikke trukket nogle chancekort
Betingelse efter	Der er blevet trukket mindst fire forskellige chancekort
Test data	-

Forventet	Vi forventer at der fremkommer 10 chancekort, og at der minimum er fire forskellige slags.
Faktisk	Der blev trukket fem forskellige chancekort. Udklip af konsollen kan ses i bilag 9.2
Resultat	Succes
Dato	23/11/17

I bilag 9.2 er der et skærmklip af testkoden med code coverage, yderligere er der vedlagt et udklip af selve *Plade* klassen, hvori metoden ligger. Den fremgår i bilag 9.2, at hele metoden og chancekortene er testet.

5.3.4 Testcase: Kontobeholdning

Her testes om hvorvidt en spillers kontobeholdning kan være positiv, negativ og 0. Det er væsentligt for spillet, at der meldes korrekt tilbage hvis en spiller får 0 eller en negativ beholdning. Da dette betyder at spilleren har tabt. Det er vigtigt fordi hvis dette sker, er spillet slut, og der skal findes en vinder.

Test case ID	TC6
Beskrivelse	I denne test case, testes der om spillernes kontobeholdninger kan være negativ.
Krav	At det ikke må lade sig gøre at få en negativ kontobeholdning
Betingelse før	Kontoen er 31
Betingelse efter	At vi får en false besked, som betyder det ikke er muligt at hæve det ønskede beløb
Test data	-32
Forventet	Vi forventer at vores variabel ”succes” er 0
Faktisk	Succesvariablen var 0
Resultat	Succes
Dato	23/11/17

I bilag 9.3 er der vedlagt et udklip af testen, hvor den tester om en konto kan blive negativ.

Testen viser at det kan den ikke. Det kan ses på code coverage at hele koden kørt, undtagen den røde linje, hvor succes bliver talt 1 op. Dette er helt korrekt, da succes kun skal tælles 1 op, hvis det lykkedes at hæve de 32 kroner, hvilket man ikke må, da der kun står 31 på kontoen.

I bilag 9.3 er der vedlagt et udklip af selve kontoklassen med code coverage. Her ses de metoder der testes i den konkrete test. SetPengeBeholdning bruges til at bestemme hvad der skal stå på kontoen. Efterfølgende bruges indsætPenge metoden, til at teste om det er muligt at hæve et bestemt beløb.

5.4 Acceptance test

Her udføres en acceptance test, som sikrer at alle krav opfyldes.

Til hvert krav er der foretaget en test med case ID, så dermed kan refereres til. Yderligere indeholder det en beskrivelse af hvad der testes, samt er der skrevet en vejledning til hvordan testen kan genskabes. Der to kolonner der beskriver det forventede og det aktuelle output. I skemaet fremgår test dato, så man ved hvornår testen er udført. Den sidste kolonne er resultat, hvor der står succes hvis den er lykkedes og fail hvis den er fejlet.

ID	Krav	Beskrivelse	Vejledning	Forventet output	Aktuelle output	Test date	Resultat
TC7	K1	Hver gang en spiller kaster, skal terningen vise en værdi mellem 1-6	Start spil -> vælg antal spillere -> udfyld oplysninger -> spiller 1 kast -> spiller 2 kast.	Vi forventer at begge spillere får en værdi der ligger mellem 1-6.	Begge spillere slog værdier mellem 1-6.	27/11 2017	Succes
TC8	K2	Spillepladen skal bestå af 24 felter.	Start spil.	Vi forventer at spillepladen kommer op med 24 felter, når spillet startes.	Spillepladen kom op med 24 felter.	27/11 2017	Succes
TC9	K3	Der skal ske en bestemt ting, hvergang der landes på et felt.	Start spil -> vælg antal spillere -> udfyld oplysninger -> spiller 1 kast ->	Vi forventer at alt afhængig af hvad vi slår, så stemmer det overens med den konsekvens der passer til feltet.	Vi slog en 3'er og landede på feltet der koster 1 krone, og derfor mistede vi 1 krone på kontoen.	27/11 2017	Succes
TC 10	K4	Hvis et felt ikke ejes af nogen, så er man tvunget til at betale prisen og købe det.	Start spil -> vælg antal spillere -> udfyld oplysninger -> spiller 1 kast ->	Vi forventer at vi bliver trukket det beløb der står på feltet, da det jo er første kast, og dermed kan det ikke have en ejer.	Vi blev trukket beløbet, og modstanderen fik ikke pengene. Dermed havde det ingen ejer.	27/11 2017	Succes
TC 11	K5	Hvis feltet man lander på, har en ejer. Så skal ejeren modtage pengene	Start spil -> vælg antal spillere -> udfyld oplysninger -> spiller 1 kast -> spiller 2 kast……	Vi forventer at hvis en af spillene lander på et felt, efter at modstanderen har landet der. Så vil penge blive trukket på den ene konto, og lagt over på den anden.	Spiller1 ramte hen på det samme felt som spiller 2 to stod på (og ejede), så han betalte ham de 3 kroner det kostede.	27/11 2017	Succes
TC 12	K6	Hvis man lander på et chancekortfelt, skal man få vist et tilfældigt chancekort og man gør hvad der bliver anvist.	Start spil -> vælg antal spillere -> udfyld oplysninger -> spiller 1 kast -> spiller 2 kast……	Vi forventer at spillet vil udføre det der står på det trukne chancekort.	Vi fik tildelt et chancekort hvor vi skulle rykke til Zoo, og det gjorde den med det samme	27/11 2017	Succes
TC 13	K7	Ved hvert slag forsættes der fra det sted man stod ved sidste kast.	Start spil -> vælg antal spillere -> udfyld oplysninger -> spiller 1 kast -> spiller 2 kast -> spiller 1 kast.	Vi forventer at hvis spiller 1 i første slag slog en 4'er, så starter spiller 1 fra felt 4 i anden omgang.	Vi slog først en 1'er med spiller 2 og efterfølgende slog vi igen en 4'er. Dette medførte at spilleren så kom hen på felt 5.	27/11 2017	Succes

TC 14	K8	Spillet skal kunne bruges af 2-4 spillere	Start spil -> vælg antal spillere.	Vi forventer at når spillet startes, så vil spillet spørge om der skal være 2,3 eller 4 spillere	Spillede startede op, og gav os muligheden for at vælge 2-4 spillere	27/11 2017	Succes
TC 15	K9	Spillerens konto skal hele tiden oplyses	Start spil -> vælg antal spillere -> udfyld oplysninger.	Vi forventer at når spillet er startet, og oplysningerne udfyldt så vil navn og kontobeholdning blive vist	Da spillet blev startet og tingene udfyldt så kom der tekst frem som beskrev kontobeholdningen	27/11 2017	Succes
TC 16	K10	Samtlige spillere starter med en balance på 30	Start spil -> vælg antal spillere -> udfyld oplysninger	Vi forventer at den balance der vises ved hver spiller starter på 30	Samtlige af spillernes balance startede på 30	27/11 2017	Succes
TC 17	K11	Spillet slutter når en af spiller har en balance 0 eller under 0.	Start spil -> vælg antal spillere -> udfyld oplysninger -> spiller 1 kast -> spiller 2 kast……	Vi forventer at når en spiller har ramt 0 eller er kommet under, så fortæller spillet hvem der har vundet.	Da en spiller kommer under 0, fortalte spillet hvem der havde flest penge tilbage, og dermed hvem der var vinderen.	27/11 2017	Succes
TC 18	K12	Hvis en spiller lander på feltet, gå i fængsel. Skal personen i fængsel og springes over	Start spil -> vælg antal spillere -> udfyld oplysninger -> spiller 1 kast -> spiller 2 kast….	Vi forventer at spilleren rykkes til fængslet, og bliver sprunget over.	Spilleren blev rykket til fængslet, og får ikke lov at kaste i den kommende runde	1/12/17	Succes
TC 19	K13	Hvis en spiller passerer start, skal han modtage 2 kroner	Start spil -> vælg antal spillere -> udfyld oplysninger -> spiller 1 kast -> spiller 2 kast….	Vi forventer at spillerens kontobeholdning øges med 2 kroner.	Spillerens kontobeholdning blev øget med 2 kroner.	1/12/17	Succes
TC 20	K14	Hvis en spiller er ejer af feltet, er hans farvet markeret rundt om feltet	Start spil -> vælg antal spillere -> udfyld oplysninger -> spiller 1 kast -> spiller 2 kast….	Vi forventer at spillerens farve markerer hvilke felter han ejer	Spillerens farve kom op rundt om feltet, da han blev ejer af det	1/12/17	Succes

6. Brugervejledning

I dette afsnit fremgår brugervejledning til spillet samt hvordan programmets kode kan importeres fra www.github.com.

6.1 Brugervejledning

Start spillet. Efter at spillet er blevet startet, så skal brugeren vælge om der skal være 2, 3 eller 4 spillere. Efter dette er gjort, bliver spiller 1 bedt om at indtaste sit navn og vælge farve på sin brik. Dette forsættes til at alle spillere har fået indtastet navn og valgt farve på brikken. Spillet vil derefter oplyse hvem der skal kaste. For at kaste trykker man på knappen ”kast”. Efter der er trykket på kast, vil spillerens brik blive rykket, alt afhængig af hvad der slås med terningen. Spillet oplyser herefter hvis tur det er nu, dette forsættes til en af spillerne har tabt.

Når en spiller taber, vil spillet komme op og oplyse hvem taberen er, samtidig med den fortæller hvem der har flest penge, altså hvem vinderen er.

6.2 Git-vejledning

Her har vi vedlagt en vejledning i hvordan man kan importere et git projekt til eclipse ved hjælp af et link. Linket vi har vedlagt er til vores git repository. Der kan man se hvordan hele udviklingen af spillet er foregået.

Vi har vedlagt et link til vores git repository. Dette link kopieres - åbn eclipse - tryk på menulinjen ”file” - tryk på import - vælg ”Projects from Git” - vælg derefter ”Clone URl” - udfyld alle felterne med link, og login til git - tryk finish. Nu er projektet importeret til eclipse.

Link til git: https://github.com/ITE-Studiegroupe/CDIO_3.git

7. Konklusion

Det er lykkedes da der gennem projektet er blevet arbejdet iterativt igennem en tre ugers lang udviklingsproces og løbende revideret i rapporten, samt koden. Retrospekt kunne der godt have arbejdet i længere tid under elaboration fasen af udviklingsprocessen. Hvilket ville kunne have resulteret i et endnu højere niveau af kvalitet for Monopoly Junior spillet.

Ud fra de foretaget test cases vises det, at spillet lever op til de opstillede krav, det kan derfor udledes ud fra afsnittet test, at programmet er succesfuldt. Det kan derfor konkluderes, at det er lykkedes at programmere et velfungerende Monopoly Junior spil, samt give belæg for dokumentationen til udviklingsprocessen i form af denne rapport.

8. Litteraturliste

Larman, C. (2004) "*Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative development*" (3. udgave) Addison Wesley Professional

9. Bilag

9.1 Terning-test

Her er et udklip af vores terninge test med code coverage.

```
public void test() {
        Terning terning = new Terning();
        int side1 = 0;
        int side2 = 0;
        int side3 = 0;
        int side4 = 0;
        int side5 = 0;
        int side6 = 0;

        for (int i = 0; i < 10000; i++) {
            int fordeling = terning.kastTerning();

            if (1 == fordeling)
                side1++;
            else if (2 == fordeling)
                side2++;
            else if (3 == fordeling)
                side3++;
            else if (4 == fordeling)
                side4++;
            else if (5 == fordeling)
                side5++;
            else if (6 == fordeling)
                side6++;

        }
        int samlet = side1 + side2 + side3 + side4 + side5 + side6;

        System.out.println("Resultat: " + side1 + " " + side2 + " " + side3 + " " + side4 + " " + side5 + " " + side6);
        System.out.println("Samlet antal: " + samlet); //for at tjekke at der ikke kommer nogle værdier udenfor 1-6
```

Udklip af terninge klassen:

```
public class Terning {

    public int kastTerning() {
        int antalØjne = ((int) (Math.random() * 6) + 1);
        return antalØjne;
    }

}
```

```
<terminated> TerningTest [JUnit] C:\Program Files\Java\jre1.8.0_151\bin\javaw.exe (24. nov. 2017 10.21.51)
Resultat: 1645 1648 1678 1728 1675 1626
```

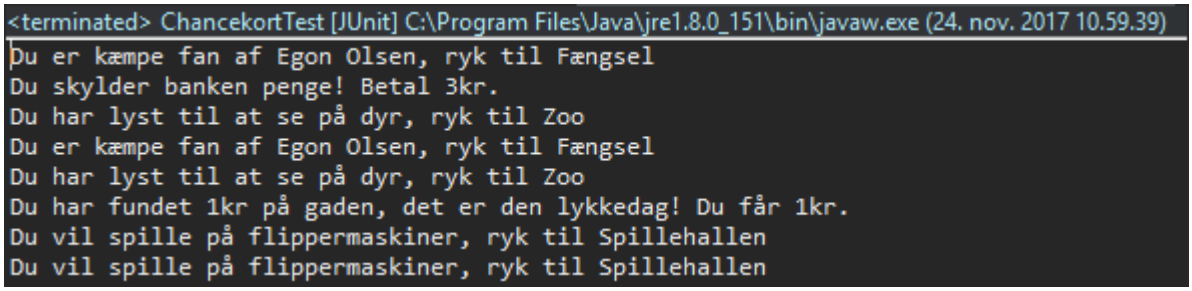
9.2 Chancekort-test

Her er dokumentation for vores Chancekort test. Der er kørt codecoverage på.

```
public class ChancekortTest {

    @Before
    public void setUp() throws Exception {
    }

    @After
    public void tearDown() throws Exception {
    }

    @Test
    public void test() {
        Plade plade = new Plade();
        plade.bygPlade();

        Chancekort chancekort;
        for (int i = 0; i < 8; i++) {

            plade.trækChancekort();

            chancekort = plade.trækChancekort();

            System.out.println(chancekort.getChanceKortTekst());

        };
    }

}
```

Udklip af pladeklassen:

```
 private void bygChancekort() {

     chancekort[0] = new Chancekort(3, 1, Tekst.TekstChancekort.TEKSTER[0]);

     chancekort[1] = new Chancekort(20, 3, Tekst.TekstChancekort.TEKSTER[1]);

     chancekort[2] = new Chancekort(4, 1, Tekst.TekstChancekort.TEKSTER[2]);

     chancekort[3] = new Chancekort(13, 3, Tekst.TekstChancekort.TEKSTER[3]);

     chancekort[4] = new Chancekort(6, 3, Tekst.TekstChancekort.TEKSTER[4]);

     chancekort[5] = new Chancekort(1, 2, Tekst.TekstChancekort.TEKSTER[5]);

     chancekort[6] = new Chancekort(2, 2, Tekst.TekstChancekort.TEKSTER[6]);

     chancekort[7] = new Chancekort(4, 2, Tekst.TekstChancekort.TEKSTER[7]);

     }
 /**
  * Træk et random chancekort
  *
  */
     public  Chancekort trækChancekort() {
     int trukketChancekort = ((int) (Math.random()*chancekort.length));
     return chancekort[trukketChancekort];

     }
```


<terminated> ChancekortTest [JUnit] C:\Program Files\Java\jre1.8.0_151\bin\javaw.exe (24. nov. 2017 10.59.39)
Du er kæmpe fan af Egon Olsen, ryk til Fængsel
Du skylder banken penge! Betal 3kr.
Du har lyst til at se på dyr, ryk til Zoo
Du er kæmpe fan af Egon Olsen, ryk til Fængsel
Du har lyst til at se på dyr, ryk til Zoo
Du har fundet 1kr på gaden, det er den lykkedag! Du får 1kr.
Du vil spille på flippermaskiner, ryk til Spillehallen
Du vil spille på flippermaskiner, ryk til Spillehallen

9.3 Konto-test

Her er et udklip af vores test der tester hvorvidt en kontobeholdning kan gå i minus.

```
@Test
public void test3() {
    Konto konto = new Konto();
    int succes = 0;
    konto.setPengeBeholdning(31);
    if(konto.indsætPenge(-32)) {
        succes++;
    }
    int actual = succes;
    int expected = 0;//det her må ikke lykkedes, da der ikke må trækkes over på kontoen

    assertEquals(expected,actual);
}
```

Udklip af Konto klassen med code coverage:

```
public void setPengeBeholdning(int pengeBeholdning) {
    this.pengeBeholdning = pengeBeholdning;
}

public boolean indsætPenge(int beløb) {
    int tjek = pengeBeholdning + beløb;
    boolean resultat;
    if (tjek <= 0) {
        resultat = false;
    } else {
        this.pengeBeholdning = tjek;
        resultat = true;
    }
    return resultat;
}
```