CDIO 3. Del

Hold A Gruppe 25

Fag:

02312, 02314, 02315

Afleveringsfrist 01-12-2017

Nicolas Vahman (s164404), Emil Vinkel (s175107) og Rasmus Hjorth (s175120) Simon Boye(s175118) & Simone Zethof (s175110)





Timeregnskab for CDIO 3. del

Dato	Deltager	Design	Impl.	Test	Dok.	Ialt
1/12/2017	Simon Boye	5	2	2	4	13
	Nicolas Vahman	4	4	2	3	13
	Emil Vinkel	4	4	0	5	13
	Simone Zethof	3	4	1	5	13
	Rasmus Hjorth	1	10	1	1	13

Abstract

This assignment examines the creation of a well known family game Monopoly Junior, developed as a computer game. The game was developed by the company IOuterActive, where we act as employees. By working together as a group combined with the knowledge obtained from three different courses we developed the game, Monopoly Junior. An important factor of this CDIO project was, that we prioritised which rules we wanted to implement in our game. Furthermore, we had to decide which requirement and rules were most important to us as developers and then implement them starting with the most important one. While working during the development process we took advantage of the different types of UML notation diagrams. By starting with drawing and constructing diagrams in the design process it helped us during the implementation process to manage the coding and to eliminate a large amount of risk failures. The coding is exclusively based on knowledge obtained from the course Introductory Programming. The test conducted is a JUnit test. The whole process of constructing the Monopoly Junior game leads to our conclusion where we successfully managed to implement the chosen rules and requirements.

Indholdsfortegnelse

Indledning

En væsentlig del af denne rapport, er samspillet mellem fagene Indledende programmering, Udviklingsmetoder til IT-systemer, samts Versionsstyring og testmetoder.
Rapporten omhandler udviklingen af brætspillet Monopoly Junior, og agere således som dokumentation for udviklingsprocessen. Der er givet forholdsvis frie tøjler for udviklingsprocessen og et af de centrale formål for den stillede opgave er således at få opstillet realistiske rammer og vurdere hvad der er mest essentielt. Den første del af rapporten beskæftiger sig med kravspecifikationer, prioriteringsrækkefølgen for implementering af valgte krav, samt en opdeling i funktionelle og ikke funktionelle krav. Efterfølgende arbejdes der videre med kravspecifikationerne ved hjælp af en central use cases og UML diagrammer benyttes til at designe og strukturer implementeringen.
Yderligere indeholder rapporten et afsnit der dokumentere den testning der er foregået med brug af JUnit, samt et diskussionsafsnit der bearbejder en række centrale spørgsmål til processen. Afslutningsvist opsumeres analysen, implementeringen i dokumentationen i konklussionen.

Analyse

Vision

I denne opgave ønskes en videreudvikling af CDIO 1 og 2 opgaven. Visionen er at udvikle et junior matador, med de virkelige felter og regler. Dog er det tilladt at prioritere hvilke regler der implementeres.

Kravliste

Funktionelle:

1. Der skal være 2 - 4 spillere.
2. Der skal være 1 terning.
3. Der skal være 24 felter.
4. Man skal slå med terning efter tur.
5. Spilleren har tur efter hinanden.
6. Vinderen er den med flest penge, når den ene går fallit.
7. Når man passere start, får spilleren 2 kr.
8. Hvis der er 2 spillere får man 20 kr. hvis 3 får man 18 kr. hvis 4 får man 16 kr.
9. Gratis parkering (helle-felt).
10. Der er 7 chancekort.
11. Lander du på fængsels feltet, skal man gå til dette og man mister sin tur.

Ikke funktionelle:

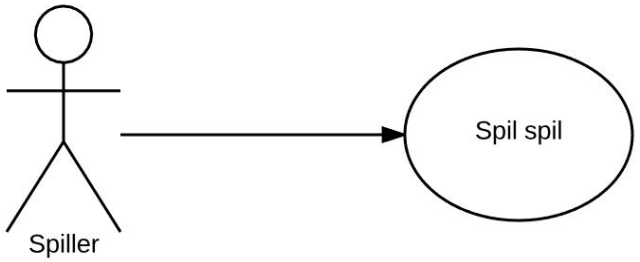
12. Den yngste starter.
13. Vælge farve på sin egen bil.
14. Kan vælge eget navn.

MoSCoW

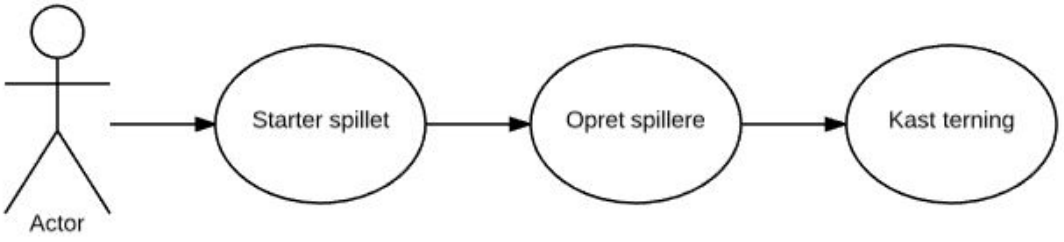
Kravene for matador spillet er inddelt i en MoSCoW prioritering, for at tydeliggøre hvilke krav der er vigtigst at implementere fra start af.

Must have Der skal være 2-4 spillere Der skal være 1 terning Der skal være 24 felter Spillerne har tur efter hinanden Spillernes biler rykker rundt på platformen Forlystelses felter med en pris Der udpeges en vinder for at afslutte spillet
Should have Købe og eje de forskellige forlystelser, og blive betalt når andre spillere lander på det respektive felt Forskellige chancekort Gratis parkerings felter Fængsels felter Vinderen er den med flest penge, når én af spillerne går fallit Når man passerer start får man 2 kr
Could have Kan vælge eget navn Ved 2 spillere starter man hver med 20 kr, ved 3 er det 18 kr. og ved 4 er det 16 kr.
Won't have Den yngste spiller starter Vælge farven på egen bil

Use Case diagram



Vores primære aktør i vores use case er en spiller. Denne spiller skal kunne spille spillet.



For at spillet kan spilles, skal aktøren først starte programmet, oprette antal spillere, og så kaste terningerne.
Diagrammet viser hvordan brugeren/aktøren interagere med systemet, som er med til at give et overblik i starten af arbejdsprocessen.

Use Case beskrivelse

Her er der eksempler på vores use case beskrevet . Der er taget henholdsvis en casual og en fully dressed use case. Begge use cases er lavet på baggrund af spil spillet (spilSpil). Forskellen på de to, er selvfølgelig datamængden inkluderet. Use casen der er casual, er naturligt ikke lige så detaljeret beskrevet som den der er fully dressed. Den hovedsagelige forskel ligger i Main flow, hvor der i detaljer er beskrevet hvad der sker når man lander på de forskellige felter. Andre elementer fra use casen (casual), er blevet bibeholdt, så som: Primary actors, Preconditions og Postconditions.

Use Case brief beskrivelse

Use case: spilSpil
ID: 1
Brief description:
Primary actors: 2 - 4 spillere
Preconditions: - Spillet skal være åben - Der skal oprettes antal spillere som er med - Hver spiller starter med en pengebeholdning på 20kr ved 2 spillere, 18 kr ved 3 og 16 kr ved 4.
Main flow: 1. *Use case starter når spiller1 kaster terningen* 2. *Spillerens bil rykkes til feltet der svarer til antal øjne* *2.1 hvis feltet er en betalingsfelt opdateres pengebeholdningen med det tilsvarende felt beløb* *2.2 hvis feltet er et chancekort, trækkes et tilfældigt kort. Spillerens pengebeholdning og placering opdateres alt efter kortet.* *2.3 hvis feltet er "gratis parkering" sker der intet, da spilleren her har helle Turen skifter til næste spiller, som får lov til at kaste med terningen.* *Spillerens bil rykket til feltet der svarer til antal øjne* *4.1 hvis feltet er en betalingsfelt opdateres pengebeholdningen med det tilsvarende felt beløb* *4.2 hvis feltet er et chancekort, trækkes et tilfældigt kort. Spillerens pengebeholdning og placering opdateres alt efter kortet.* *4.3 hvis feltet er "gratis parkering" sker der intet, da spilleren her har helle* *…. 1-4 gentages indtil én af spillerne ikke har flere penge tilbage. Den spiller med flest penge på kontoen vinder.*
Postconditions: - *Der bliver udpeget en vinder.*

Use Case Fully Dressed beskrivelse

Use case: spilSpil Fully dressed
ID: 2
Brief description:
Primary actors: 2 - 4 spillere
Preconditions: - Spillet skal være åben - Der skal oprettes antal spillere som er med - Hver spiller starter med en pengebeholdning på 20kr ved 2 spillere, 18 kr ved 3 og 16 kr ved 4.
Main flow: *1. Use case starter når spiller1 kaster terningen* *2. Spillerens bil rykkes til feltet der svarer til antal øjne* ***2.1*** *hvis feltet er et betalingsfelt opdateres pengebeholdningen med det tilsvarende felt beløb* *2.1.1. hvis feltet er "burgerbaren" trækkes 1 kr fra spillerens pengebeholdning* *2.1.2. hvis feltet er "Pizzahuset" trækkes 1 kr fra spillerens pengebeholdning* *2.1.3. hvis feltet er "Slikbutikken" trækkes 1 kr fra spillerens pengebeholdning* *2.1.4. hvis feltet er "Iskiosken" trækkes 1 kr fra spillerens pengebeholdning* *2.1.5. hvis feltet er "Museet" trækkes 2 kr fra spillerens pengebeholdning* *2.1.6. hvis feltet er "Biblioteket" trækkes 2 kr fra spillerens pengebeholdning* *2.1.7. hvis feltet er "Skaterparken" trækkes 2 kr fra spillerens pengebeholdning* *2.1.8. hvis feltet er "Swimmingpool" trækkes 2 kr fra spillerens pengebeholdning* *2.1.9. hvis feltet er "Spillehallen" trækkes 3 kr fra spillerens pengebeholdning* *2.1.10. hvis feltet er "Biografen" trækkes 3 kr fra spillerens pengebeholdning* *2.1.11. hvis feltet er "Legetøjsbutikken" trækkes 3 kr fra spillerens pengebeholdning* *2.1.12. hvis feltet er "Dyrehandlen" trækkes 3 kr fra spillerens pengebeholdning*

2.1.13. hvis feltet er "Bowlinghallen" trækkes 4 kr fra spillerens pengebeholdning
2.1.14. hvis feltet er "Zoo" trækkes 4 kr fra spillerens pengebeholdning
2.1.15. hvis feltet er "Vandlandet" trækkes 5 kr fra spillerens pengebeholdning
2.1.16. hvis feltet er "Strandpromenaden" trækkes 5 kr fra spillerens pengebeholdning

2.2 *hvis feltet er et chancekort, trækkes et tilfældigt kort. Spillerens pengebeholdning og placering opdateres alt efter kortet.*
2.2.1. hvis chancekortet er det 1. rykkes spillerens bil frem til start
2.2.2. hvis chancekortet er det 2. rykkes spillerens bil 5 felter frem
2.2.3. hvis chancekortet er det 3. rykkes spillerens bil 1 felt frem
2.2.4. hvis chancekortet er det 4. skal spilleren betale 2 kroner
2.2.5. hvis chancekortet er det 5. rykkes spillerens bil frem til "strandpromenaden"
2.2.6. hvis chancekortet er det 6. får spilleren 1 kr.
2.2.7. hvis chancekortet er det 7. får spilleren 2 kr.

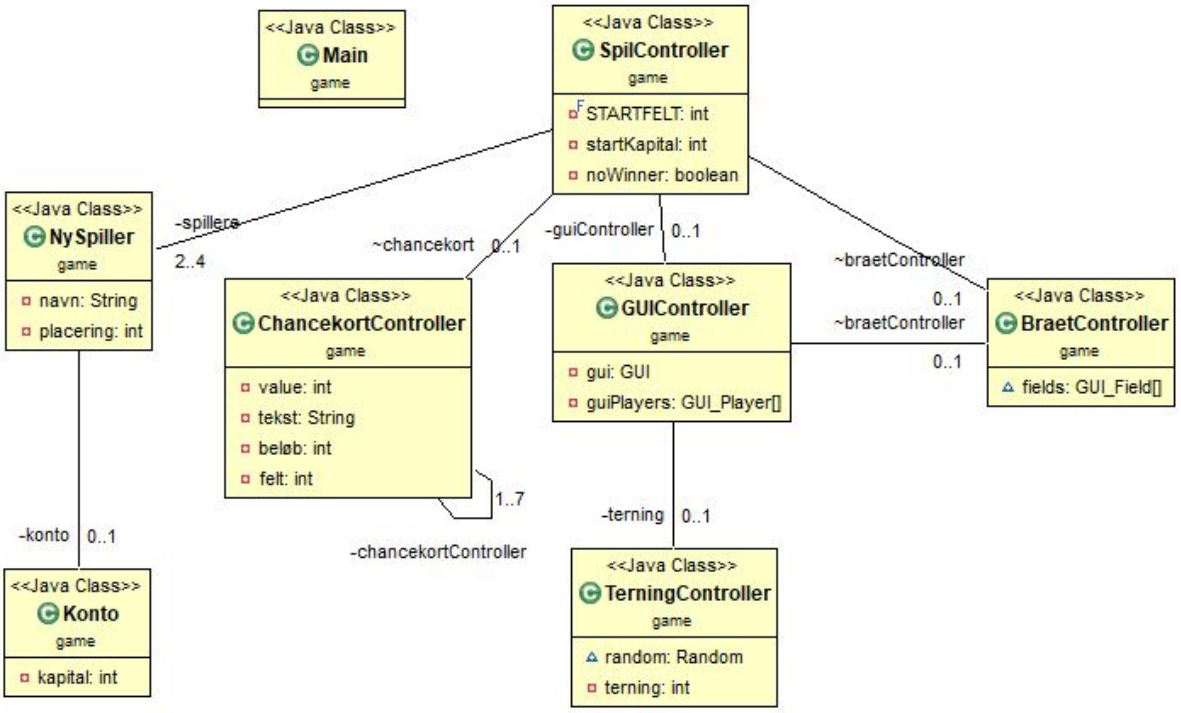
2.3 *hvis feltet er "Gratis parkering", sker der ingenting. Her har spilleren helle indtil næste runde.*

3. Turen skifter til næste spiller, som får lov til at kaste med terningen.

…. 1-4 gentages indtil én af spillerne ikke har flere penge tilbage. Den spiller med flest penge på kontoen vinder.

Postconditions:
- *Der bliver udpeget en vinder.*

Domænemodel



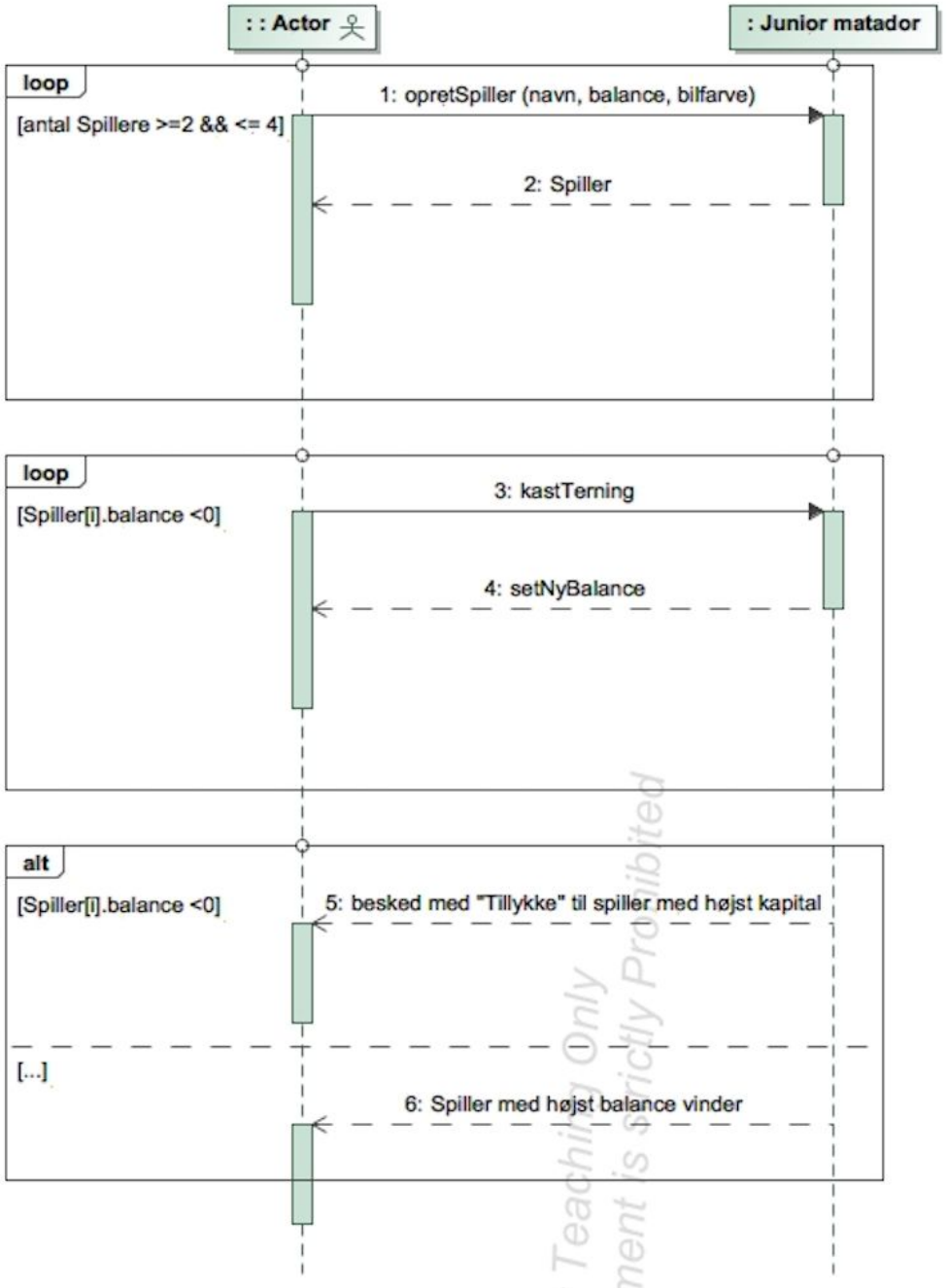
Dette diagram er en domænemodel. En domænemodel er med til at dele de forskellige klasser op i ting man har fra virkeligheden. Man benytter altså en domænemodel, til at give et bedre overblik over hvordan tingene fra virkeligheden kan blive implementeret og laves til et program.

I ovenstående model har vi vores klasser fra vores spil repræsenteret. I dette diagram har man kun attributter med.

Design

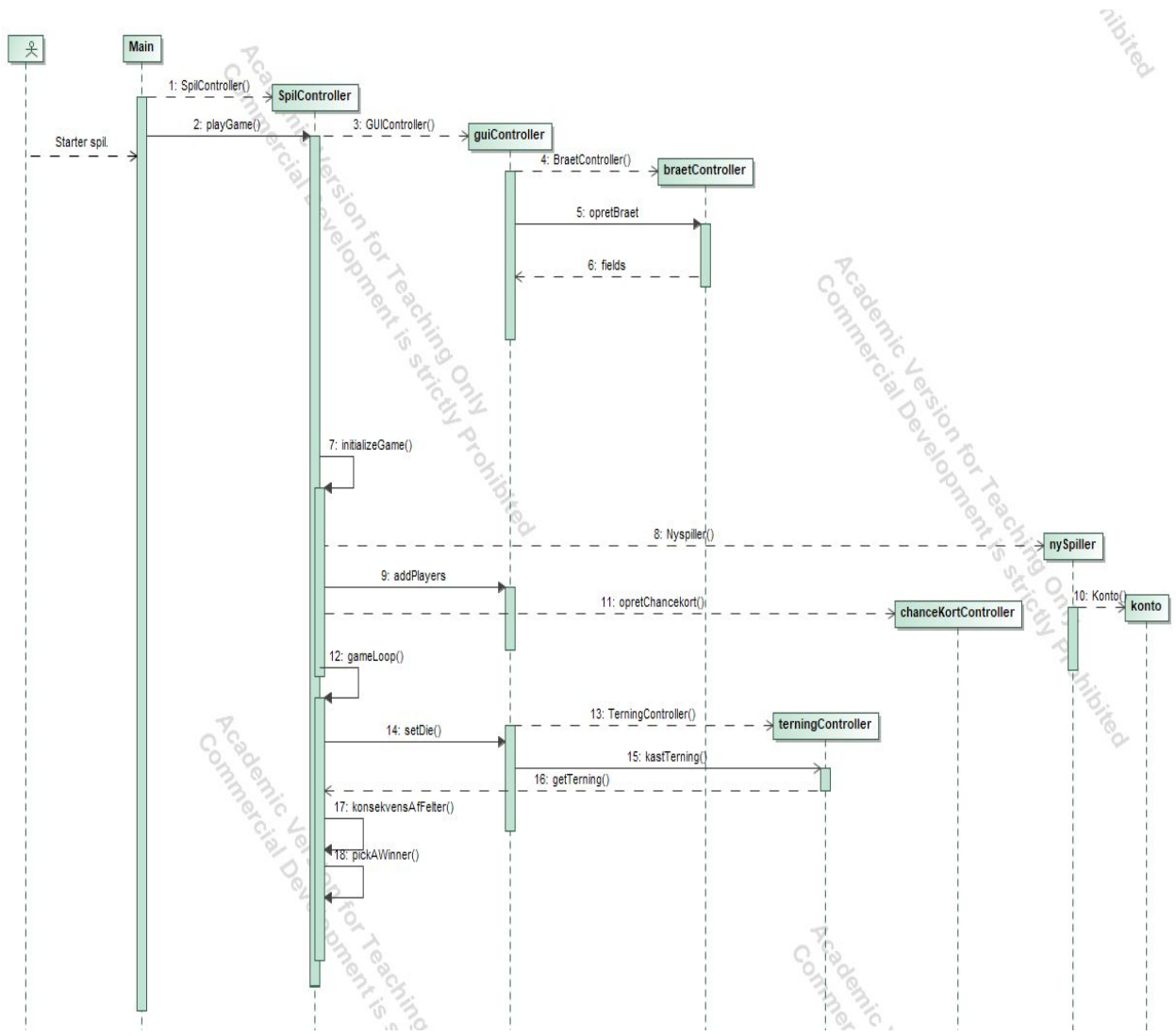
System sekvensdiagram

System sekvensdiagrammet viser det vigtigste scenarie fra vores use case. Her bliver al kommunikationen mellem aktør og system, behandlet som en blackbox. Det vil altså sige, at al kommunikation bliver gemt.

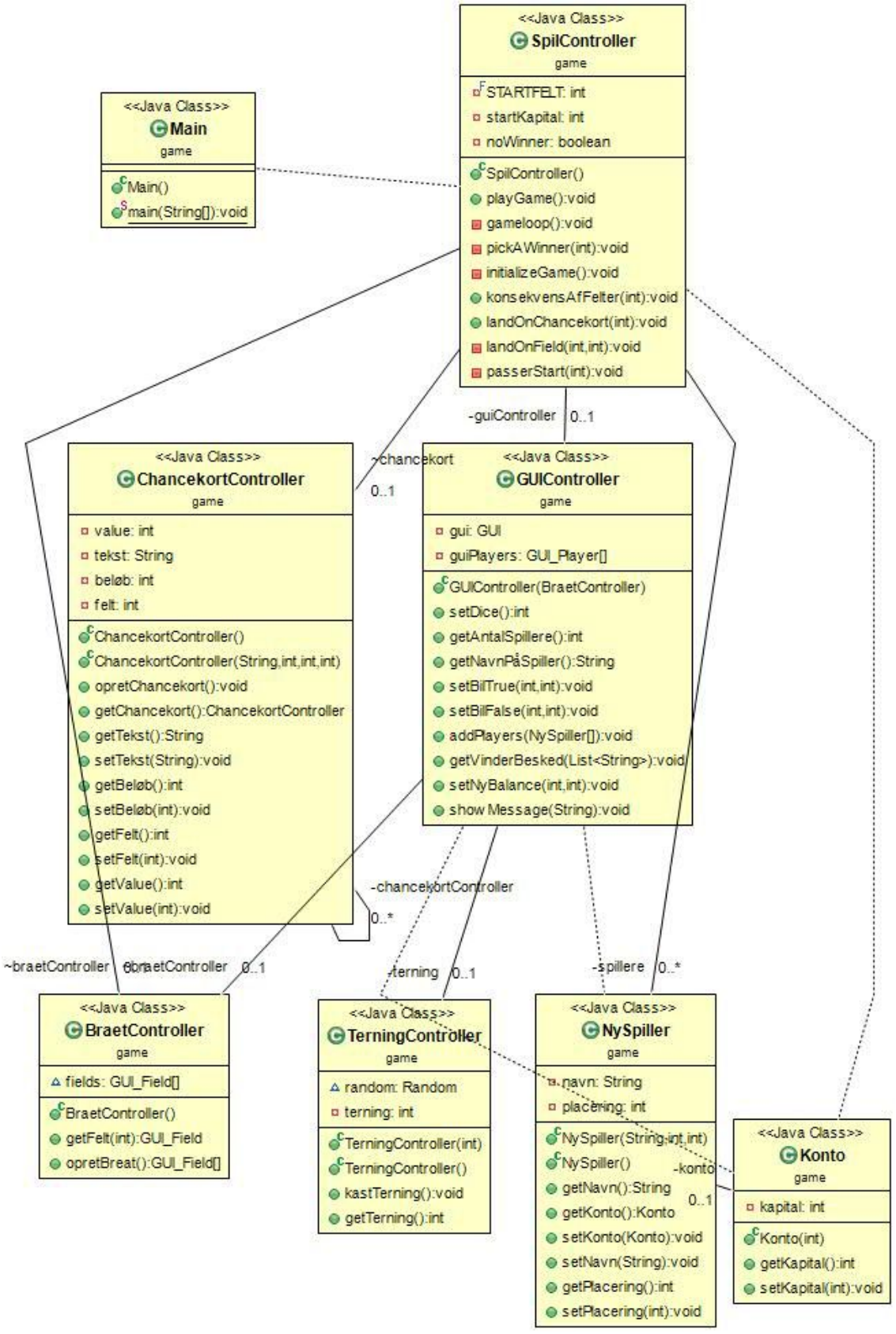


Sekvensdiagram

Vores interaktionsmodel, er det man kalder et sekvensdiagram. Det er med i forlængelse af vores domænemodel. Sekvensdiagrammet er en viderebygning på domænemodellen. Denne model viser også det tidsmæssige aspekt. Den tydeliggøre hvor lang tid de respektive metoder er i brug.



Design klassediagram



Design klassediagram er en udvidelse af domain modellen. Det der adskiller de to, er at design klassediagrammet også indeholder metoder.
Denne model er vigtig i forhold til forståelsen af hvordan programmet skal implementere de forskellige metoder, og hvilke klasser der skal oprettes.
Klasserne og metoderne kan kobles til de forskellige krav.I BraetController oprettes matadorspillets felter, som følger den originale spilleplade. I TerningeController oprettes den terning, som spillerne på skift slår med. I setBilTrue() metoden rykkes spillerens bil rundt på brættet og i setKonto().setKapital() opdateres spillerens balance alt efter det pågældende felt.I pickAWinner() metoden vælges en vinder af spillet som derefter afsluttes.

Dokumentation

Forklar hvad arv er
Arv mellem klasser er et kendt begreb indenfor programmering og opbyggelse af klasser. Arv forstås ved dannelse af en såkaldt superklasse. Denne superklasse indeholder sine egne attributter og metoder. Disse metoder og attributter, kan dog genbruges af andre klasser man kalder for subclasses. forholdet mellem superklassen og subclasses er det man kalder nedarvning, i det at de forskellige subclasses "arver" metoder og attributter for superklassen.
I kodesprog angives det i Java, med det reserverede ord; "Extends".
I UML, angives dette med en bestemt type pil.

Et godt eksempel på nedarvning kunne være hvis man har en superklasse der repræsentere dyr i en zoo: For at lave dette opretter man en klasse der hedder dyr. Dette er vores superklasse, som indeholder attributter som navn, alder, art, ID osv. Subclasses til denne ville være forskellige arter af dyr eks. Ræv, hund, ko, løve osv. Disse subclasses arver, alle egenskaberne/attributterne fra superklassen.

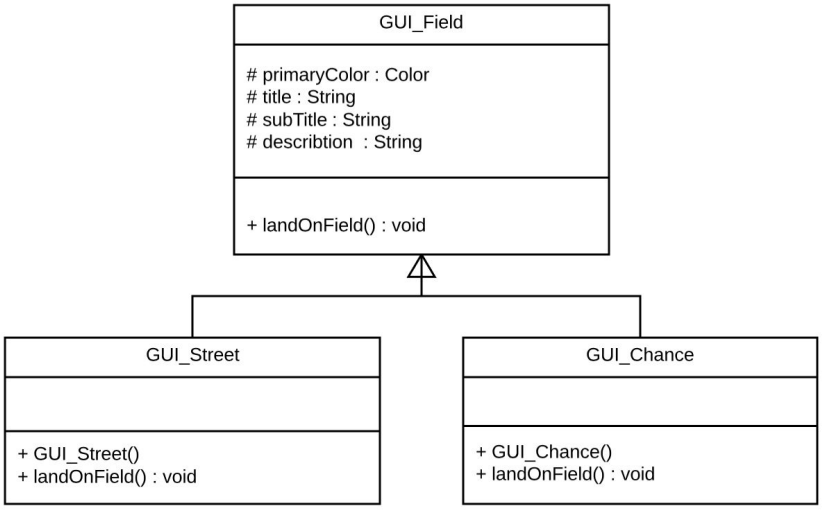
Udover, at man har de såkaldte super-klasser og sub-klasser, hvor sub-klasser nedarver fra superklassen, vil der altid være object klassen, som alle klasser i Java nedarves af. Det vil altså sige, at der er nogle helt generelle metoder som alle klasser kan kalde. fx. toString() og equals() metoden.

Forklar hvad abstract betyder
Abstract er en deklaration man benytter når man laver klasser. Abstract sikrer at man kan ikke institiantere et nyt objekt af klassen. En abstrakt klasse kan betragtes som en overordnet model for en underordnet gruppe af subklasser. Ser man fx på klassen Figurer, så kan den fx indeholde en tegn(); metode, og en areal(); metode. Det giver ikke nogen mening at bruge

disse metoder på klassen figurer, da det jo ikke er definerbart med et areal, eller at tegne en overordnet gruppe som Figurer. Det giver derfor god mening at erklærer klassen abstrakt, og man kan således ikke instantisere et objekt af klassen, og kalde metoder derpå. I stedet kan man oprette nye subklasser til Figurer klassen, som fx Firkant, Cirkel, Trekant m.m. og lade disse klasser nedarve metoderne. Her vil metoderne tegn(); og areal(); være brugbare og Figurer agerer således som en overordnet struktur model.

Fortæl hvad det hedder hvis alle fieldklasserne har en landOnField metode der gør noget forskelligt.

Dette kaldes for polymorfi. Det opstår når der oprettes klasser, der extender af en superklasse. I matadorspillet er superklassen GUI_Field, som har attributterne *primaryColor, title, subTitle, describtion.* GUI_Street og GUI_Chance nedarves af GUI_Field, og nedarver derfor også attributterne og metoderne. Disse metoder kan overskrives i de nedarvede klasser, så der sker noget forskelligt, alt efter om man lander på et Street eller et Chancekort felt.
Når de forskellige fieldklasser har en fælles metode, som fx landOnField, og at denne metode gør noget forskelligt, kaldes det for polymorfi. Vi har en abstrakt klasse fields, der er den grundlæggende model for de forskellige typer af fieldklasser. Når man opretter et array af Fields, og dette array indeholder forskellige typer af Fields, fx Street felter og Chance felter, så kan man køre arrayet igennem og kalde metoden landOnField på de respektive objekter.



GRASP

Når man udvikler et software program stræbes der efter at overholde GRASP principperne. Vi har i vores projekt arbejdet meget intensivt i ugen op til afleveringen. Dette skyldes at vi brugte meget tid på at omskrive vores kode, for at overholde GRASP principperne. Størstedelen af metoderne var før statiske og da dette ikke lever op til kravene om høj samhørighed og lav kobling, skrev vi koden om. Vi har herefter prøvet, at tilnærme os GRASP principperne (Controller, Creator, Low coupling, High Cohesion) ved at tildele de enkelte klasser et tydeligt ansvar, spilController styrer selve spil arkitekturen og kontrollerer metoder kaldt på de andre klasser, mens guiController styrer den grafiske user interface. Herefter benytter vi konstruktører (Creator) til at oprette nye objekter af de tydeligt defineret klasser og kalde metoderne herpå (modsat ved benyttelse statiske metoder hvor metoderne kaldes direkte på klassen). Ved at have veldefinerede metoder og herunder ansvar på de enkelte klasser fx TerningController, styrer terningen, får vi skabt en lav kobling (Low coupling), idet at ved at ændre metoderne i fx TerningController, kræver det ikke en ændring i de resterende klasser, da vi kalder TerningController metoderne, direkte på et objekt af TerningController. Yderligere har vi tilstræbt at have navne på klasserne der overskueliggøre de enkelte klassers ansvar, som fx SpilKontroller, GUIKontroller, attributter som integer startKapital, boolean noWinner og metoder der tydeligt forklarer hvad deres formål er; kastTerning(), opretChancekort(). Dette opfylder kravet om høj samhørighed (High Cohesion) og gør koden mere overskueligt og lettere at vedligeholde.

Konfigurationsstyring

Udviklingsplatform, som her også fungere som produktionsplatform:
Mac OS X Sierra 10.8 eller højere
Windows 7 eller højere
Java (Oracle) Version 8 Update 151.
Eclipse Version: Oxygen.1 Release (4.7.1)
Maven repository matadorgui.jar v. 3.1.2
Maven repository Junit 4.12

For at importere spillet til Eclipse udføres følgende trin[1]:

1) Eclipse åbnes.
2) Click file og import…
3) Søg efter Projects from Git, eller scroll down i menuen og find punktet på listen.
4) Herefter vælges "Clone URl"
5) Man vil nu blive bedt om Authentication details til Github.
6) Herefter vil der være mulighed for at vælge Branches, master branch indeholder final release, men det kan vælges at hente de resterende for at følge arbejdsprocessen.
7) I det følgende vælges hvor ens Git repository skal ligge lokalt.
8) Dernæst vælges det at importere projektet som et eksisterende eclipse projekt.
9) Herefter vælges lokation i workspace, og man vil nu have tilgang til projektet.
10) Vinduet Git Repositories bør nu være synligt i Eclipse, hvis ikke kan det findes ved at clicke på Window>Show View>other og søge efter Git Repositories.

[1] Denne vejledning til import til Eclipse er taget fra vores CDIO 2-rapport

Test

Test case 1

I første test Case tester vi om terningen slår random.
Resultatet af vores Test Case for forekomsten af terningens øjne, efter 100.000 rul med terningen ses her. Selve koden for testen, kan ses i *bilag 1*.

Sum	Forekomst af øjne
1	16755
2	16747
3	16654
4	16708
5	16665
6	16471

Det ses at ud af de 100.000 rul af terningen, fordeler hyppigheden sig som den skal. Der er nogenlunde lige stor sandsynlighed for forekomsten af de 6 udfald på terningen, derfor konkluderer vi, at vores kald af metoden rulTerning() bruger metoderne fra klassen Random som den skal.

Code coverage på vores JUnit test af terningen. Hele testen blev dækket.

Test case 2

I anden test case tester vi om placeringen af et NySpiller objekt kan sættes til en ny værdi og om den derefter kan hentes igen. Koden for testen kan ses i bilag 2.

Steps:

1. En 'int expected' sættes, i dette tilfælde, til en værdi på 20.
2. spillerens placering sættes ved brug af metoden setPlacering(int placering), kaldt på objektet, hvor placering er bliver sat til 20.
3. En 'int actual' sættes, i dettte tilfælde, til spillerens nuværende placering ved brug af metoden getPlacering(), kaldt på spiller objektet
4. expected og actual sammenlignes herefter ved brug af assertEquals(String, expected, actual) fra pakken org.junit.assert.*

Her ses det at testen forløb fejlfrit, og det kan hermed konstateres at test case 2, var en succes.

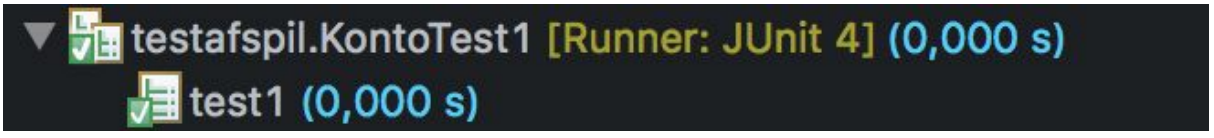
Code coverage på vores JUnit test af placering. Hele testen blev dækket.

Test case 3

Vi har valgt at teste vores konto klasse, da denne er central for at opdatere spillernes konto, når de lander på de forskellige felter.
Vi tester om placeringen af et Konto objekt kan sættes til en ny værdi og om den derefter kan hentes igen. Koden kan ses i bilag 2

Steps:

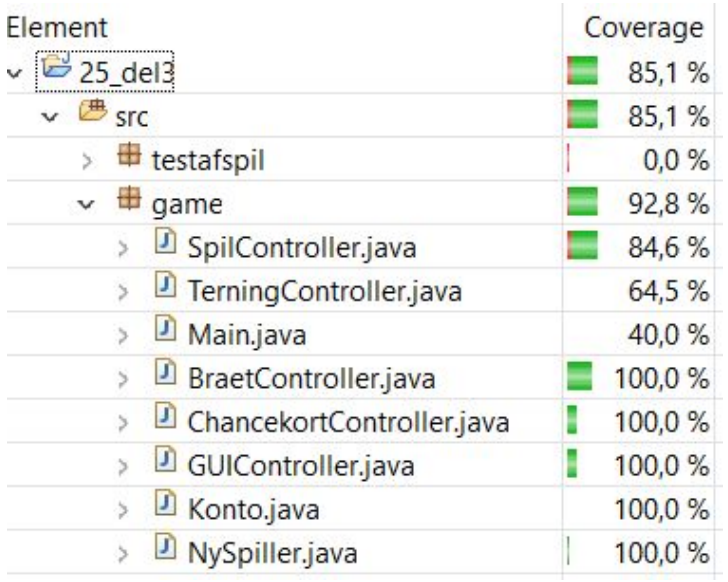
1. En 'int expected' sættes, i dette tilfælde, til en værdi på 100.
2. Konto objektets placering sættes, ved brug af metoden setKapital(int kapital), kaldt på objektet, hvor kapital bliver sat til 100.
3. En 'int actual' sættes, i dette tilfælde, til den pågældende kontos nuværende kapital ved brug af metoden getKapital(), kaldt på konto objektet
4. expected og actual sammenlignes herefter ved brug af assertEquals(String, expected, actual) fra pakken org.junit.assert.*.



Vores test blev en succes og metoderne fungerer derfor som de skal.

Code coverage på vores JUnit test af konto/balance. Hele testen blev dækket.

Code coverage:



Element	Coverage
25_del3	85,1 %
src	85,1 %
testafspil	0,0 %
game	92,8 %
SpilController.java	84,6 %
TerningController.java	64,5 %
Main.java	40,0 %
BraetController.java	100,0 %
ChancekortController.java	100,0 %
GUIController.java	100,0 %
Konto.java	100,0 %
NySpiller.java	100,0 %

Her ses en komplet Code Coverage af vores game-package, som indeholder samtlige klasser i spillet. Som det ses i game-package så er det 92,8 % af spillet som bliver dækket, hvilket må siges at være i orden. Typisk sigter udviklere efter, at få dækket omkring 80 % af koden[2], før softwaren eksempelvis kan udleveres til kunden.

Grunden til, at testmappen er 0,0 % er, at det er valgt kun at teste for game-package, men det ses, at det er med til at trække dækningen af src-mappen ned.

[2] https://msdn.microsoft.com/en-us/library/dd537628.aspx

Projektplanlægning

Vi har i løbet af dette projekt, alle deltaget aktivt i at lave opgaven samt lave koden. Vi har, ligesom sidst, arbejdet meget fra en computer af gangen, da vi alle har siddet samlet om koden. I forhold til rapporten og test osv. har vi fordelt arbejdet ligeligt mellem alle medlemmer af gruppen.

Vi har desuden arbejdet en del mere intensivt i ugen op til afleveringen. Dette skyldes at vi brugte meget tid på at refakturere vores kode. Størstedelen af metoderne var før statiske og da dette ikke lever op til kravene om høj samhørighed og lav kobling, skrev vi koden om.

Diskussion

Vi har lavet et matadorspil, som lever op til dele af kravene angivet.
Kravene fra starten af projektet er angivet i forhold til MosCows princip i henholdsvis: Must have, should have, could have og won't have, og er implementeret derefter. Dette betyder at det derfor ikke er alle kravene der er blevet implementeret i vores spil. Projektet lever dog op til kravene som skulle bruges i spillet, og de undladte er spillemæssigt uvæsentlige. Nogle af de krav som ikke er blevet benyttet, er blandt andet det, at spilleren er i stand til at købe en grund. Den del der bygger på, at man derfor kan købe grunde og tjene på den måde er derfor ikke mulig. Vores spil fungerer på den måde, at man for hvert felt mister den værdi der er tilknyttet det specifikke felt. Den spiller, der vinder spillet, er derfor den, der er i stand til at opretholde den højeste kapital på kontoen, når en vilkårlig spiller går i minus.

Et formål med denne opgave var desuden at give opgaveløserne et realistisk billede på rammer og hvad der var mest essentielt at implementere. Vi kan som sagt konkludere fagligt og personligt, at denne opgave har været med til at give et meget mere realistisk billede på programmeringsmæssige opgaver, som kunne have relation til det virkelige liv. Opgaven har været med til at vi nu er i stand til at vurdere, hvad der skal med i et programmet og hvad der kan sorteres fra, samt hvilke dele der er vigtigst at få implementeret først.

Ydermere er samspillet mellem de tre fag Indledende programmering, Versionsstyring og testmetoder og Udviklingsmetoder til it-systemer, i brug, og principperne fra hvert fag er benyttet. Analysefasen består af UML-diagrammer og er repræsenteret i form af faget udviklingsmetoder til it-systemer, implementeringen af koden er repræsenteret i form af faget indledende programmering, og vores test afsnit er repræsenteret i form af faget versionsstyring og testmetoder.

Konklusion

Vi kan samlet konkludere, at vi har lavet et færdigt spil, med de mest centrale krav, givet i opgaveformuleringen. Dog må vi også konkludere at vi ikke har været i stand til at implementere, at man kan købe og eje felter på nuværende tidspunkt, men muligheden for implementering er mulig og inden for rækkevidde.
Vi har brugt de tre fag der indgår i CDIO-projektet og principper fra alle fag er i brug.

Bilag

Bilag 1

```
@Test
public void random() {

    TerningController testTerning = new TerningController();

    int[] sum = { 0, 0, 0, 0, 0, 0, };

    int i = 0;
    while (i < 100000) {

        testTerning.kastTerning();


            switch (testTerning.getTerning()) {

            case 1:
                sum[0]++;
                break;

            case 2:
                sum[1]++;
                break;

            case 3:
                sum[2]++;
                break;

            case 4:
                sum[3]++;
                break;

            case 5:
                sum[4]++;
                break;
            case 6:
                sum[5]++;
                break;

            }

            i++;
        }
        System.out.println("Sum\tForekomst af øjne");

        for (int v = 0; v < sum.length; v++) {
            System.out.print(v + 1);
            System.out.print("\t");
            System.out.println(sum[v]);
        }
```

Bilag 2

```
package testafspil;

import static org.junit.Assert.*;

public class TestPlacering {

    @Before
    public void setUp() throws Exception {
    }

    @After
    public void tearDown() throws Exception {
    }

    NySpiller spiller = new NySpiller();
    @Test
    public void test() {
        int expected = 20;
        spiller.setPlacering(20);
        int actual = spiller.getPlacering();
        assertEquals("Stemmer ikke overens", expected, actual);

    }

}
```

Bilag 3

```
1  package testafspil;
2
3  import static org.junit.Assert.*;
10
11 public class KontoTest1 {
12
13     @Before
14     public void setUp() throws Exception {
15     }
16
17     @After
18     public void tearDown() throws Exception {
19     }
20
21     @Test
22     public void test1() {
23         Konto kontotest = new Konto(100);
24         kontotest.setKapital(100); // Sætter en værdi for kontoen
25         int expected = 100;
26         int actual = kontotest.getKapital(); // Henter værdien og tester for det indsatte
27         assertEquals(expected, actual);
28
29     }
30
31 }
32
```