02312 CDIO_Final

Gruppe 34

Alan Jafi s122417
Asef Khatak s155918
Burhan Shafiq s175446
David Nikolaj Milutin s160601
Peter Tønder Blendstrup s175210
Sebastian Stokkebro Sørensen s170423

15. Januar 2018

Contents

Timeregnskab

CDIO Final timeregnskab						
	Asef	**Alan**	**Burhan**	**David**	**Peter**	**Sebastian**
Indledning	0	0	0	0	0	0,5
Systemkrav	0	1	3	0	1	3
Designmodeller	3	7	2,5	0	3	4
Implementering	2	18	22	4	25	25
Test	0	1	1	0	3	3
Rapportskrivning	1	1	3	0	3	8
I ALT	6	28	31,5	4	35	43,5

1 Indledning

De følgende sider, er en gennemgående rapport over CDIO_Final, som omhandler udviklingen af spillet Matador. Formålet med projektet er at skabe et funktionelt Matador-spil med de klassiske regler, som er knyttet til spillet. Der vil i løbet af rapporten blive uddybet hvilke fremgangsmetoder der er brugt til at nå frem til de mål, som er sat for projektet. Disse uddybelser indeholder eksempler og forklaringer af blandt andet designmodeller, i form af sekvensdiagrammer og designklassediagrammer, samt kodestykker af testmetoder og implementering.

I begyndelsen af rapporten fremlægges systemkravene i form af prioritering, use cases og en domænemodel. Dette giver et bedre overblik og en mere klar forståelse for hvilken retning projektet skal bevæge sig i, samt hvordan systemet skal struktureres og opbygges.

2 Systemkrav

For at bedre kunne have overblik over de forskellige krav, og for at opfylde kundens ønske om at have et kørende spil med lidt implementeret end et spil med meget implementeret, der ikke kan køres, har vi dannet en kravliste, der deler kravene op alt efter, hvor vigtigt det er at have dem implementeret for det endelige produkt.

2.1 Must have (højeste prioritet)

- Spillet skal kunne have 2-6 spillere i et spil.
- Spillet skal have et raflebæger og to terninger.
- Der skal være et bræt med 40 felter.
- Spillet skal kunne køres på en DTU maskine.
- Spillet skal kunne reagere passende, afhængigt af, hvilken type felt, der landes på.
- Felterne skal have passende ikoner/ farver.
- Spillet skal have et fungerende gå i fængsel felt
- Spillerne modtager 200kr når de passerer start.

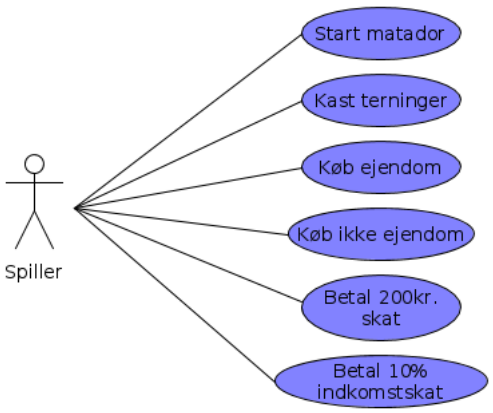
2.2 Good to have (Næst-højeste prioritet)

- Lejen på et gade felt skal stiges afhængigt af, hvor mange bygninger der er på feltet.
- lejen på et rederi felt skal også stige afhængigt af, hvor mange andre rederier ejeren af rederiet ejer.
- Chancekort trækkes fra toppen af bunken, og derefter sættes nederst i bunken.
- Alle felter, der kan købes, skal have deres pris stående på selve feltet.
- Hvis brugeren lander på et chancefelt, skal de få en meddelelse om, hvilket chancekort det er, som de har trukket
- Brugeren skal kun have mulighed for at købe en bygning på et felt, hvis de ejer alle 3 områder på pladen af samme farve.

2.3 Nice to have (Laveste prioritet)

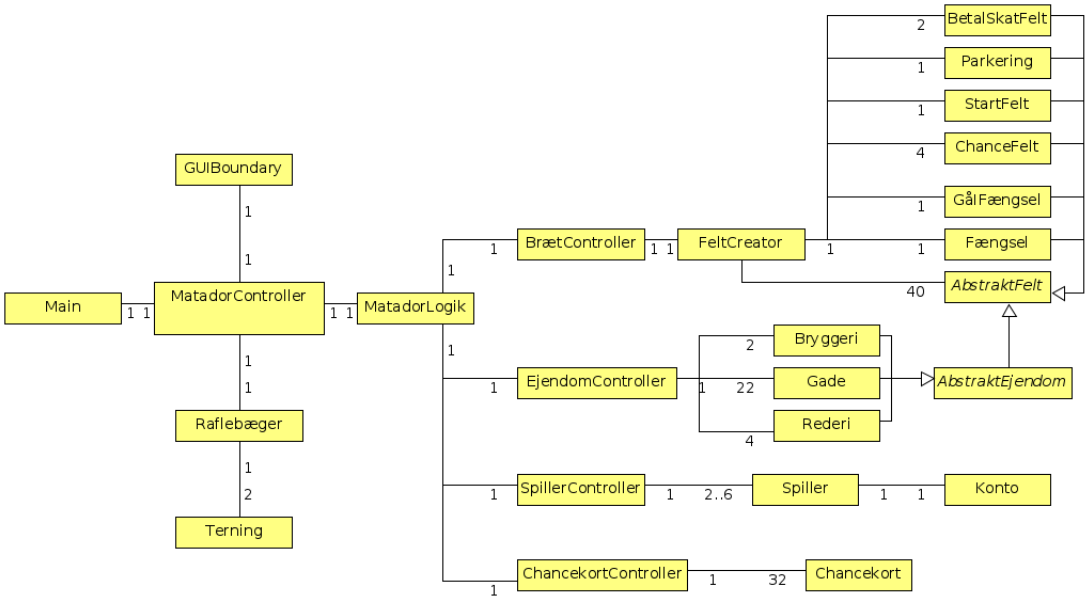
- Pantsætning af huse
- Alle 32 chancekort skal indgå i spillet.
- Mulighed for spillerne at kunne forhandle/ bytte deres ejendomme og penge med hinanden.
- Spilleren skal have mulighed for at betale 10 procent af den samlet værdi på deres aktiver, i stedet for blot at betale 10 procent af deres kontanter.

2.4 Use case diagram



Ud fra diagrammet, kan man se de centrale scenarier spilleren indgår i. Helt enkelt vi spilleren have mulighed for at starte spillet. Derefter vil spilleren have mulighed for hhv., at kaste terningen, købe eller afslå købet af en ejendom og betaling af skat.

2.5 Domænemodel

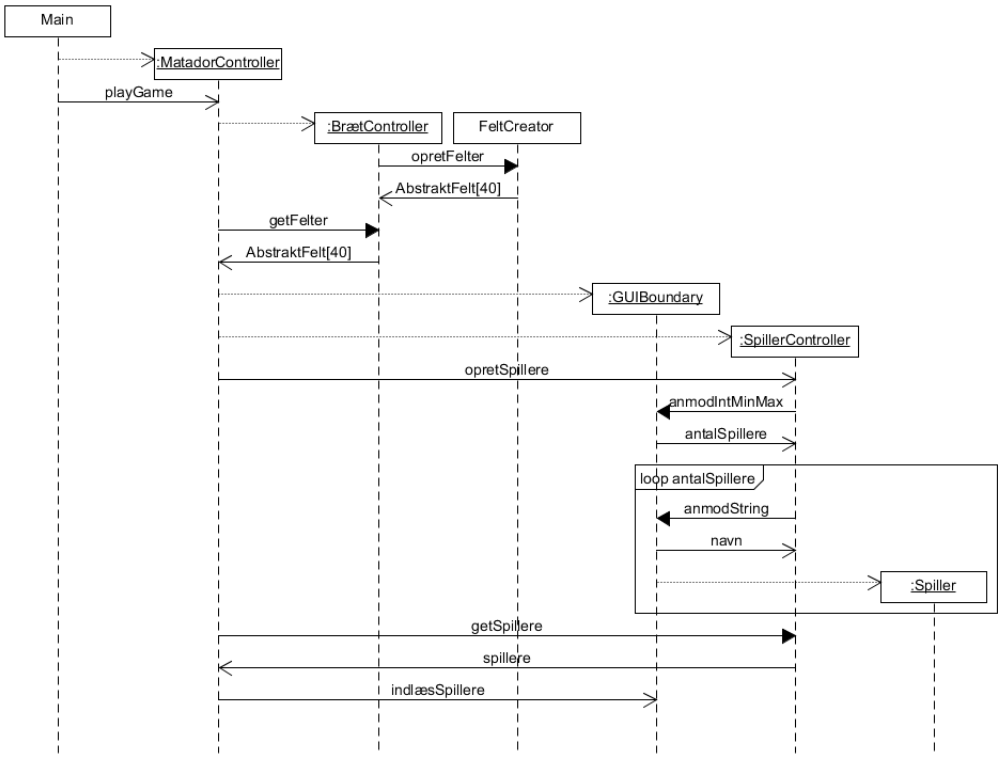


Domænemodellen viser forholdet mellem de forskellige klasser og multipliciteterne imellem. Man kan ud fra domænemodellen også se hvilke klasser der nedarver fra andre klasser, samt hvilke abstrakte klasser der er med i systemet.
Vi har oprettet to abstrakte klasser. En for ejendomme og en for felterne generelt. Disse abstrakte klasser indeholder metoder som alle felter og ejendomme har brug for. Disse metoder indgår i alle felterne via nedarvning, og derfor er det muligt at bruge dem når der henvises til en bestemt ejendom eller et bestemt felt.

3 Designmodeller

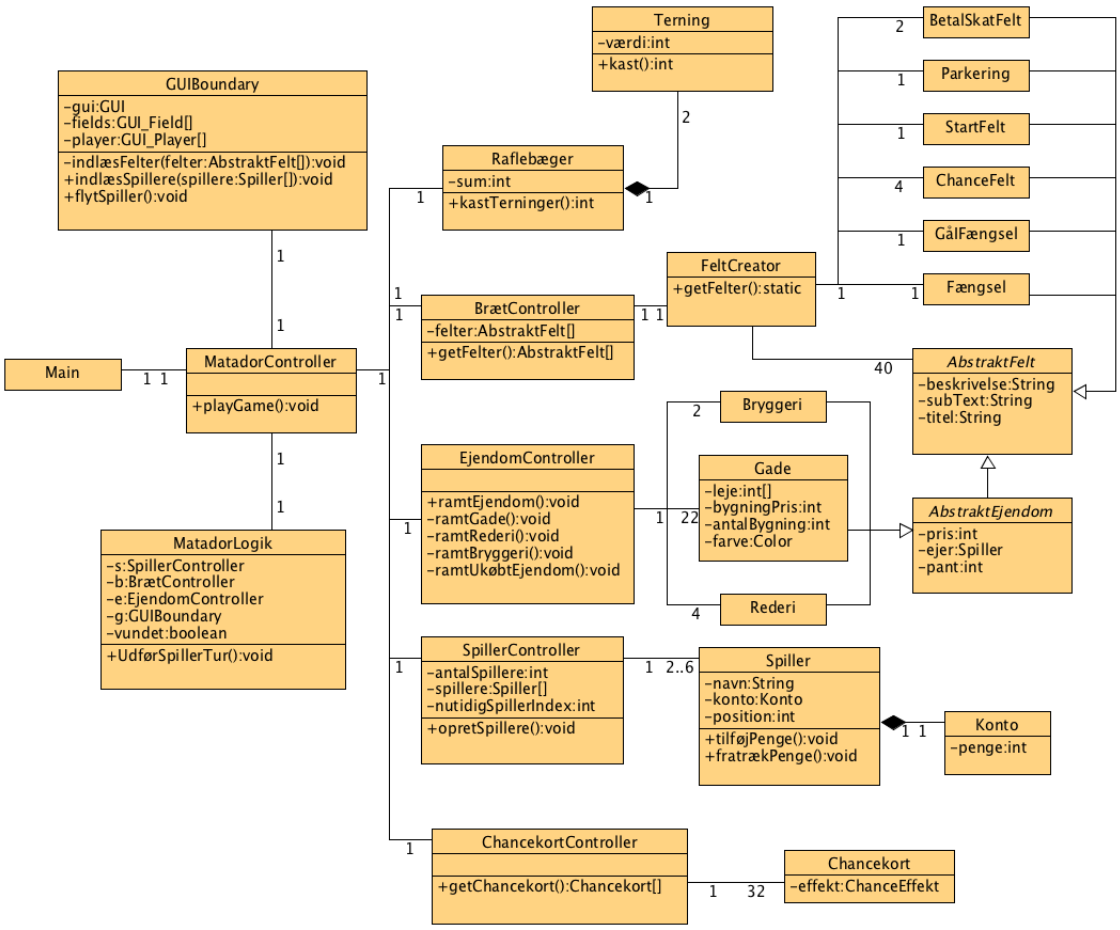
I denne sektion gennemgås de designmodeller, som tilhører systemet. De viser de centrale dele af systemet og deres proces, som skaber funktionaliteten af programmet.

3.1 Sekvensdiagram



Vi har valgt at lave et sekvensdiagram af processen, der tager sted når systememt startes op. Som der kan ses på vores sekvensdiagram, så bliver der dannet instanser af de forskellige klasser løbende. Spillet startes fra main klassen, hvor der bliver dannet en matador controller, hvor der så bliver kaldt på en "play game" metode, for at kunne starte spillet op. Det er herfra, hvor instanserne af de andre controllere og creatorer bliver dannet. Vi har også tilføjet en string ved processen af oprettelse af en spiller, da man skal indtaste navnene på alle spillerne.

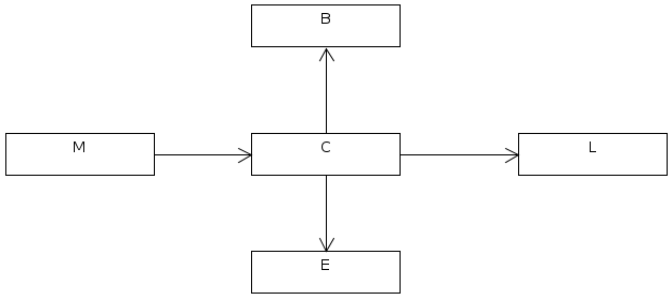
3.2 Designklasse-diagram



Vi kan ud fra designklassediagrammet se hvilke metoder og attributter der indgår i systemets klasser. Samtidig kan vi se helhederne og hvilke der er afhængige af hinanden. Her er det for eksempel tydeligt at felterne, som nedarver, nedarver metoderne fra de abstrakte klasser.

4 Implementering

Ved implementeringen gennemgås en række af udvalgte kodestykker fra programmet. Formålet med afsnittet her, er at uddybe evt. kompleks kode, eller blot forklare valg af metoder og strategier.
Først og fremmest har vi lagt vægt på at benytte BCE arkitektur til opbygningen af systemet. Dette er med til at bidrage og sikre lav kobling og høj binding for højere effektivitet af programmet og dets indre struktur. Vi går udfra følgende model.



Formålet er, at spillerne interagerer med boundary objektet, boundary objektet informerer controllerne om brugerens input, fx. en streng-anmodning eller en valgknap. Controllerne sender muligvis forespørgsler tilbage til boundary omkring yderligere information fra spilleren, fx ved betaling af skat, hvor spilleren kan vælge enten 10% indkomstskat eller blot betale 200kr. Dette forespørges efter spilleren har kastet terningerne. Derefter opdaterer controllerne de entities, som findes.

Dykker vi ned i en af vores centrale controllers, EjendomController, kan man se hvordan vi opdeler koden i metoder for, at holde det så enkelt og struktureret som muligt. Et eksempel er ramtEjendom-metoden som ser således ud.

```
public void ramtEjendom(AbstraktEjendom ejendom, Spiller spiller) {
    if (ejendom.getEjer() == null)
        ramtUkøbtEjendom(ejendom ,spiller);

    else if (ejendom instanceof Gade)
        ramtGade((Gade) ejendom, spiller);

    else if (ejendom instanceof Rederi)
        ramtRederi((Rederi) ejendom, spiller);

    else if (ejendom instanceof Bryggeri)
        ramtBryggeri((Bryggeri) ejendom, spiller);

}
```

På denne måde vil en meget central metode fremkomme ganske simpel kun ved brug af metoder. Fokuserer vi på ramtRederi-metoden, vil man se, at samme fremgangsmåde bruges her.

```
private void ramtRederi(Rederi ejendom, Spiller spiller) {
    if (ejendom.getEjer() != spiller)
    {
        betalLejeRederi(ejendom, spiller);
    }
    else
        g.sendBesked(spiller.getNavn() + " er landet på en gade som De selv ejer");
}
```

Da lejen for rederier ændre sig alt efter hvor mange af disse spilleren har, skal der mere kode til end blot fratræk af penge.
Vi kan endelig tage et kig ned i betalLejeRederi-metoden og se koden bag.

```
private void betalLejeRederi(Rederi ejendom, Spiller spiller)
{
    int ejersAntalRederier = 0;
    for (int i = 0; i < ANTAL_REDERIER; i++)
    {
        if ( rederier[i].getEjer() == ejendom.getEjer() )
            ejersAntalRederier++;
    }

    int leje = 25;
    for (int i = 1; i < ejersAntalRederier; i++)
    {
        leje *= 2;
    }

    spiller.fratrækPenge(leje);
    ejendom.getEjer().tilføjPenge(leje);

    g.sendBesked(spiller.getNavn() + " lander på " + ejendom.getTitel() + " som er ejet af "
            + ejendom.getEjer().getNavn() + ". " + spiller.getNavn() + " betaler " + ejendom.getEjer().getNavn()
            + " " + leje + "kr.");

    g.opdaterAllesPenge();
}
```

Her kan man se, at programmet tjekker om ejeren af de to rederier er ens. I det tilfælde vil ejerens antal af rederier gå en op. Har spilleren et rederi er lejen 25, har spilleren to, vil lejen blive fordoblet. Penge fratrækkes den nutidige spillers konto og denne leje tilføjes til ejerens konto. GUI'en sender en en besked og alle penge opdateres grafisk på brættet.

5 Test

Først og fremmest testes Raflebæger-klassen, der håndterer terningkast og summen af de to terninger som er i spil. Det som ønskes testet er værdien af de øjne der slås. Der ønskes altså ikke et output, der er mindre end 2 eller større end 12. det testet ved hjælp af dette testscript.

```
@Test
public void raflebæger_slag_test() {
    r = new Raflebæger();
    int sum = 0;
    for(int i = 0; i < 10000; i++) {
        r.kastTerninger();
        sum = r.getSum();
        boolean condition = sum >= 2 && 12 >= sum;
        assertTrue(condition);
    }
}
```

Testen går ud på at slå med to terninger, i en tilpassende mængde af gentagelser, hvor der tjekkes om, summen af de to terninger ligger indenfor intervallet [2, 12]. Dernæst testes de centrale metoder i Spiller-klassen, altså tilføj- og fratrækPenge-metoderne. Testscriptet ser således ud.

```
@Test
public void tilføj_penge_test() {
    s.tilføjPenge(500);
    int expected = 1500 + 500;
    int actual = s.getPenge();

    assertEquals(expected, actual);
}

@Test
public void fratræk_penge_test() {
    s.fratrækPenge(500);
    int expected = 1500 -500;
    int actual = s.getPenge();

    assertEquals(expected, actual);
}

@Test
public void negativ_test_0() {
    s.fratrækPenge(1501);
    boolean condition = s.getPenge() == 0;

    assertTrue(condition);
}
```

De to første teststykker går blot ud på, at teste om metoderne fungerer. Det tredje tesstykke, er en negativ test, som skal teste om det er muligt for en spillers pengebehold at gå i minus. Vi regner med at betingelsen (condition) er sand, da der allerede er implementeret kode i fratrækPenge-metoden, som holder styr på at pengebeholdet ikke går i minus.
For at teste de implementerede metoder har vi oprettet en teststub af Raflebægeret. Idéen er at vi har kontrol over hvor på brættet spillerne befinder, så metoderne for betalt leje af flere ejede bryggerier og rederier testes. Vi laver et array med simuleringer af terningekast. På en måde kan vi beslutte hvilke felter

spillerne skal ramme.
Der er også oprettet teststubbe af Raflebægeret, som tester hhv. fængselregler og tab af spil. Et eksempel på et array fra teststubbene, ser således ud.

```
public class RaflebægerTestStub extends Raflebæger {

    private int[] testArray = {
            1,4, // Køb Rederi Øresund
            4,1, // Leje 25

            5,5, // Rederi Ø.K.
            5,5, // 50

            5,5, // Rederi Bornholm
            5,5, // 100

            5,5, // Rederi D.F.D.S.
            5,5, // 200

            // Passer start på henholdsvis hver deres tur.
            5,1, // Køb Rødovrevej
            2,4, // Leje 2

            6,3, // Fængselsbesøg
            3,3, // Chancekort (Ikke implementeret, kræver yderligere teststub)

            1,1, // Køb Tuborg
            2,3, // Leje Tuborg
            6,6, // Udregning af leje m. et bryggeri, (6+6)*4 = 48

            4,4, // Helle
            4,3, // Køb Strandvej af anden spiller

            3,5, // Køb Carlsberg
            6,3, // Leje Carlsberg
            1,1, // Udregning af leje m. begge bryggerier, (1+1)*10 = 20

            };
```

6 Konklusion

Vi kan ud fra rapporten konkludere, at brug af BCE arkitektur strukturerer programkoden, og giver et overblik. Det er ud fra systemkrav, og designklassediagrammer også muligt at følge programmets proces.

Vi kan også konkludere at teststubbe er en effektiv måde at overtage kontrollen på, og dermed teste de implementerede metoder.

Det er altså ved hjælp af disse metoder muligt at udvikle et klassisk matadorspil, med en række af de klassiske regler. Samtidig må vi indse, at grundet tidsmangel har vi måtte udelukke nogle af vores "good-to-have" og "nice-to-have" punkter. Alle "must-have" punkterne er blevet implementeret.

Litteraturliste

- Lewis Loftus. (2015). "Java Software Solutions, Seventh Edition", Pearson Education, Inc., ISBN 10: 1292018232
- Craig Larman. (2005). "Applying UML and Patterns An Introduction to Object-Oriented Analysis and Design and Iterative Development 3rd edition" Pearson Education, Inc. ISBN 0-13-148906-2