

62532 - Versionsstyring og testmetoder E21

Rapport CDIO Del 2

Projektgruppe: 11

October 29, 2021



Sofie Groth Dige
[s211917](#)



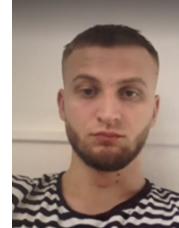
Mathilde Hjorth
[s215835](#)



Anshjyot Singh
[s215806](#)



Nick Tahmasebi
[s195099](#)



Marco Miljkov Hansen
[s194302](#)



Technical University
of Denmark

1 Timeregnskab

Nick	30 timer
Marco	35 timer
Ansh	35 timer
Matilde	15 timer
Sofie	35 timer

Indholdsfortegnelse

1 Timeregnskab	2
2 Indledning	4
3 Projektplanlægning	4
3.1 Krav	5
4 Analyse	6
4.1 Kravspecifikationer	6
4.2 Riskmanagement	8
4.3 P*I Matrix	9
5 Design	10
5.1 Use-case diagram	10
5.2 Domænemodel	13
5.3 Design klassediagram	13
5.4 System Sekvensdiagram	15
5.5 Sekvens diagram	16
6 GRASP-PATTERNS:	16
7 Implementering	18
7.1 Kode:	18
7.2 Konsollen	20
7.3 GUI-implementation	21
8 Test	23
9 Konklusion	27
10 Bilag	27

2 Indledning

I denne CDIO opgave har vi udviklet et program kaldet Junior Matador til vores kunde, som vi har kodet i programmeringssproget Java, i IDE-programmet IntelliJ. Der vil i dette tværfaglige projekt med kurserne Indledende programmering, Versionsstyring og testmetoder, Udviklingsmetoder for IT-systemer tages udgangspunkt i udviklingen af et terningspil. Det er et 2-player spil som handler om at opnå 3000 kr på sin spillekonto først, et spil som forholder sig på et bræt med 11 felter spilbare felter. Hver spiller slår med et raflebæger med to terninger, og rykker antallet af øjne hen til et felt. Hvert af disse felter har enten en positiv eller negativ pengesum som vil agere med ens bankkonto. Vi skal anses som værende udviklere, hvor vi er opdelt i grupper, som hver især skal analysere, designe, implementere og teste denne opgave.

Kundens vision:

Kunden har en vision om, at det skal være et spil mellem 2 personer, der kan spilles på maskinerne i DTU's databarer, uden bemærkelsesværdige forsinkelser. Kravene og de mest centrale punkter fra kunden vil blive yderligere beskrevet senere i rapporten.

3 Projektplanlægning

Før vi, i gruppen gik i gang med selve projektet, så sad vi i fællesskab og sørgede for at klargøre en plan for hvordan vi skulle tilgå denne opgave. En måde hvorpå vi kunne tydeliggøre prioriteringerne i vores projekt, samt lave en intern aftale som lød på hvilke krav der først skulle løses før vi fortsatte arbejdet mod andre krav eller forespørgsler der var til projektet. Herunder ses en NeedToHave og NiceToHave liste som beskriver hvilke krav der SKAL løses og hvilke krav der kunne være FEDT at løse, hvor "Need to have" kravene naturligvis var øverst i vores prioriteringsliste.

3.1 Krav

	Krav	NeedTo Have	NiceTo Have
1	Et spil mellem 2 personer	X	
2	Spillet skal kunne bruges på maskinerne (Windows) i databarerne på DTU	X	
3	Man kan ikke slå 1 med to terninger	X	
4	Spilleren får en ekstra tur hvis spilleren lander på “The WereWall”	X	
5	Afleveringen skal inkludere et link til vores github repo	X	
6	Spillet skal nemt kunne oversættes til andre sprog	X	
7	Spillet er slut når en spiller når 3000	X	
8	Alle almindelige mennesker kan spille det uden en brugsanvisning	X	
9	En afprøvningskode, der sandsynliggør at balancen aldrig kan blive negativ, uanset hvad hæv og indsæt metoderne bliver kaldt med. (JUnit)	X	
10	Der skal gøres brug af UTF-8 som tegnsæt	X	

11	Benytter GUI		X
12	Det skal være let at skifte til andre terninger	X	
13	At lande på felterne med numrene 2-12 har en positiv eller negativ effekt på spillernes pengebeholdning	X	
14	Rafflebægeret virker korrekt	X	
15	Efter hvert slag multipliceres terningeslaget til det gamle landte felt.		X

Figur 1: Viser kravene sorteret i en NeedToHave og NiceToHave prioriteringsliste

Med “Almindelig mennesker” i krav 8, menes der mennesker som ikke adskiller sig fra gennemsnittet, altså f.eks. uden påfaldende egenskaber, skal kunne spille spillet uden brugsanvisning. Vedrørende krav 14, hvor rafflebægeret skal virke korrekt, så menes der, at bægeret skal kunne rumme to eller flere terninger og kunne generere to tilfældige slag.

4 Analyse

4.1 Kravspecifikationer

I denne del af rapporten beskrives vores krav yderligere, samt en risikoanalyse som tager udgangspunkt i de risici som kan forekomme under udviklingen af vores projekt.

Funktionelle krav

ID	Beskrivelse
k1	Et spil mellem 2 personer
k2	Slå med 2 terninger med 1 raflebæger.
k3	Terningerne skal hver have 6 sider, altså to “ standard” terninger.
k4	Spillerne starter med 1000 point
k5	Spiller vinder når de opnår 3000 point
k6	Der skal være felter med numrene 2-12. Disse har en positiv eller negativ effekt på spillerens balance i banken.

Figur 2: Viser vores Funktionelle krav sorteret

Ikke-funktionelle krav

ID	Beskrivelse
k7	Spillerne skal kunne spille spillet uden brugsanvisning.
k8	Systemet skal kunne køre på Windows maskinerne i databarene på DTU.
k9	Det skal gøres muligt for kunden at inspicere koden, og der skal derfor linkes til vores github repo.
k10	Der skal udføres test (JUnit), da det er et krav til reliability.
k11	En afprøvningskode, der sandsynliggør at balancen aldrig kan blive negativ, uanset hvad hæv og indsæt metoderne bliver kaldt med

Figur 3: Viser vores Ikke-Funktionelle krav sorteret

4.2 Riskmanagement

Risici i sammenhæng med udviklingen af Juniormatador:

- | |
|---|
| <p>1. Udskiftning af medarbejdere</p> <p>1.1. Med medarbejdere menes der os som gruppe, der udvikler et program for en virksomhed. Vi kan risikere, at en af os bliver syge og kan misse udviklingen af projektet i et par dage. Dog er risikoen for at der er en reel udskiftning af gruppen meget lille, da der er ingen, der forlader studiegruppen, og vi har derfor valgt en sandsynlighed på 1.</p> <p>1.2. Men hvis det skulle ske, at der i virkeligheden var en udskiftning af en af os i gruppen, ville det påvirke helheden af projektet en del, da vi alle har betydningsfulde roller, samt antallet af vores gruppe: 5, som gør at der ikke er mange medarbejdere at tage af. Vi har derfor valgt en påvirkning på 3.</p> <p>1.3. Denne risiko kan dog nemt forebygges, ved f.eks. at have en specialist på området og sørge for at flere medarbejdere kan varetage samme opgave. Derved bliver sandsynligheden også mindre.</p> |
| <p>2. Systemet underpræsterer</p> <p>2.1. Da vi alle i gruppen bruger MacBooks, samt er vant til det, kan vi have problemer i forhold til at skulle kode programmet på Mac. Kravet er nemlig at det skal fungere på en Windows maskine. Vi har derfor valgt en sandsynlighed på 3.</p> <p>2.2. Vi mener at påvirkningen er meget høj, da vi ikke kan levere et projekt til kunden, der ikke virker eller ikke lever op til kravspecifikationen om at det skal køre på Windows maskinerne i <u>databarerne</u>. Af den grund ligger påvirkningen på 5.</p> <p>2.3. For at forebygge dette kunne man udføre en masse test løbende og kontrollere at programmet virker.</p> |
| <p>3. Dårligt omdømme</p> <p>3.1. Virksomheder med et stærkt ry præstere generelt bedre, samt tiltrækker kvalificerede medarbejdere og øger deres samlede succes. Dog vil der nærmest altid være dårligt omdømme, eksempelvis ift. brugere der ikke har særlig god værdsættelse for produktet. Men hvis der tages udgangspunkt i vores projekt, så mener vi ikke at der er "stor" plads til dårligt omdømme og derfor har vi sat sandsynligheden til 1.</p> <p>3.2. Hvis der fokuseres på selve påvirkningen af det dårlige omdømme, så vil påvirkningen være rimelig mild. Generelt vil dårligt omdømme ikke medvirke til den største påvirkningen på en virksom, og det vil det specielt</p> |

heller ikke i vores projekt hvor sandsynligheden for at der forekommer dårligt omdømme er lav i forvejen. Netop derfor har vi valgt at sætte påvirkningen til 1.

4. Misforståelse af kundes vision

- 4.1. Da det er meget nyt for vores gruppe at fortolke og forstå kravspecifikationer, samt at vi har opdaget kundens vision ikke har været specifik nok indtil videre, har vi sat sandsynligheden på 2.
- 4.2. For et hvilket som helst softwareudviklingsprojekt vil det være katastrofalt at misforstå kundens vision og deres kravspecifikationer, og det samme gælder for vores. Hvis vi ikke forstår hvad vi skal levere til kunden, er det praktisk talt umuligt at få en glad kunde. Af de grunde har vi vurderet påvirkningen til at være 5.

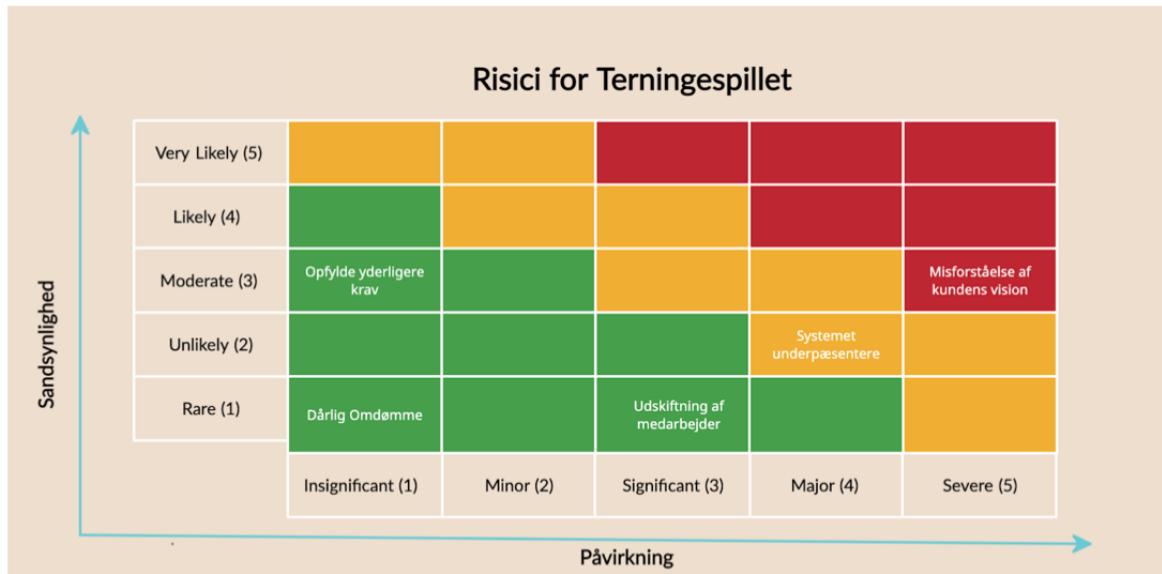
5. Opfyldede yderligere krav

- 5.1. Vi kan komme ud for at vi ikke har nok ressourcer og/eller tid til at udvikle ekstra funktioner for at tilfredsstille kunden uover kravene. Sandsynligheden for dette er middel, da det er vores andet projekt, og vi kender endnu ikke til hvordan vi er mest effektive endnu. Vi har derfor valgt en sandsynlighed på 3.
- 5.2. Påvirkningen er derimod meget lav, da det ikke har en betydning for kunden om vi leverer ekstra-opgaver eller ej. Kunden bliver tilfreds med de grundlæggende krav, og vi har derfor sat påvirkningen til 1.
- 5.3. For at forebygge dette, kan det være nødvendigt at være bedre til at prioritere vores tid, så man dermed har tid til at udvikle ekstra features til programmet. Dette sænker sandsynligheden for denne risiko og det kan give en mere tilfreds kunde.

Figur 4: Viser vores Riskmanagement

4.3 P*I Matrix

Vi har tidligere beskrevet de adskillige risici der kan forekomme i vores projekt, og hvor stor sandsynligheden er for at den forekommer samt beskrevet om hvorvidt påvirkningen af risiciene vil have en stor indflydelse på vores projekt - ved at rangere dem fra 1-5. Herunder ses resultaterne af vores risici ift. hvordan de opstilles i vores P*I Matrix. Vi kan altså visuelt gennemskue hvilken slags handling vi skal tage, hvis vi er ude for problemer.

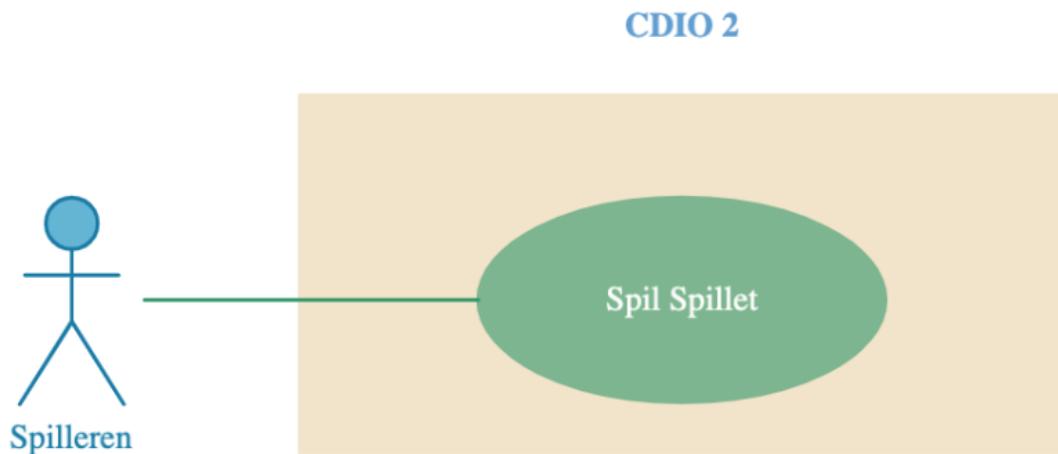


Katasprofisk	STOP
Uønsket	HANDLING
Ønskværdigt	INGEN HANDLING

Figur 5: Viser vores P*I Matrix

5 Design

5.1 Use-case diagram



Figur 6: Viser vores Use-case diagram
Use-case diagrammet består af en primær aktør, som er 'Spilleren' og et enkelt use-case "Spil spillet". Det er kort og kontant, da der kun er én use-case.

Brief description:

Spillet består af 12 felter, hvor hvert felt enten giver en positiv eller negativ pengesum igen. Spillet består af to spiller der på skift skal slå med et raflebæger med to terninger. Den første spiller der opnår 3000 kr har vundet.

Primary actors:

Spiller 1

Spiller 2

Secondary actors: Ingen**Preconditions:**

Hver spiller starter med 1000 kr i banken.

Der skal være et raflebæger med to terninger, to spillere og. Hver spiller starter på start-feltet,

Main flow:

1. Spiller trykker på "Start"
2. Systemet starter et nyt spil
3. Spiller indtaster sit navn
4. Systemet viser "Spiller" + "navn"
5. Spilleren trykker på "enter" for at slå med bægeret

5.1 Systemet viser resultatet af terningekastet

5.2 Spilleren slår to tilfældige tal med terningerne.

5.3 Systemet rykker spilleren summen af terningerne frem på brættet.

5.4 Spilleren lander på et felt, som passer til antallet af øjne på terningen.
Startposition + antallet af øjne.

5.5 Systemet indsætter "x" antal kr på spillerens konto.

5.51 Spillerens konto opdateres

6. Systemet afslutter spillerens tur.
7. Systemet beder den anden spiller om at slå med bægeret

Scenariet fortsætter fra punkt 5 indtil man opnår en balance på 3000.

8. En spiller opnår 3000 kr på deres ~~spillerkonto~~

8.1 Systemet viser “Spiller “x” har vundet!”

8.a Spiller vælger “Spil igen”

Scenarie fortsætter fra punkt 2

8.b Spiller vælger “Afslut spil”

8.b1 Systemet afslutter spillet.

Postconditions:

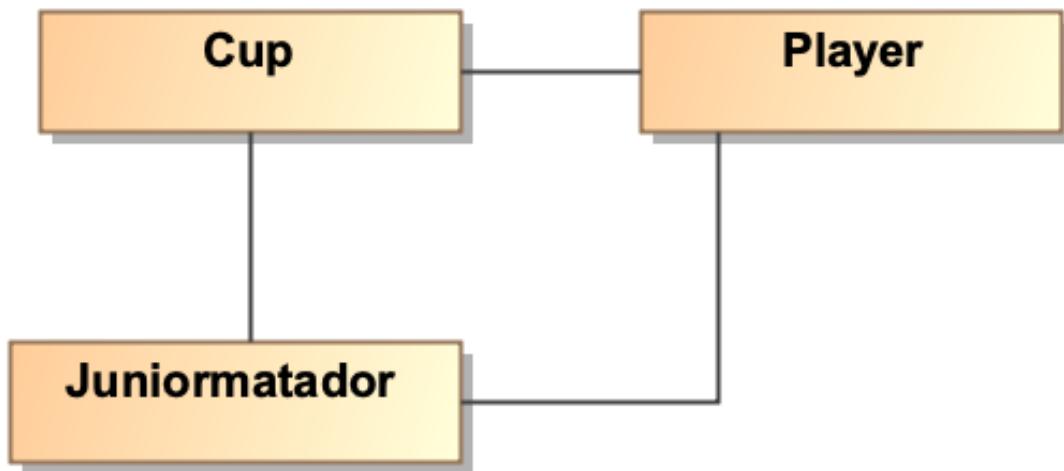
Når enten spiller 1 eller spiller 2 opnår 3000< penge på deres konto har de vundet.

1. “Player “x” won”

Alternative flows: ingen

5.2 Domænemodel

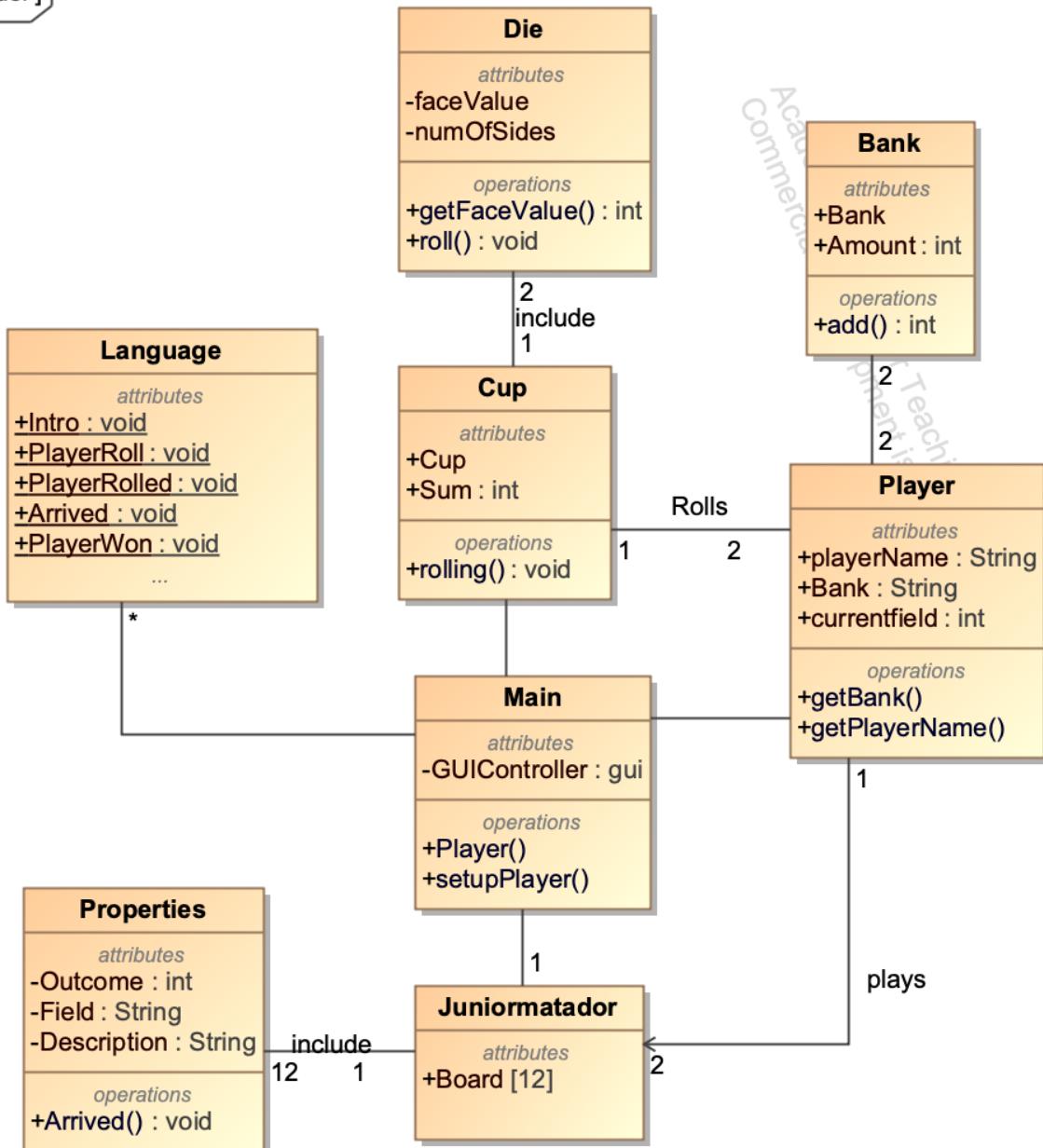
Vores domæne model består af 3 klasser, "Player", "Cup", og "Juniormatador". Spilleren røfler med et bæger, og spiller Juniormatador. Bægeret ("cup") er inkluderet i spillet, da terningerne ikke kanstå alene, når der skal kastes med to. Vores domæne model ser ud som følgende:



Figur 8: Viser vores Domænemodel

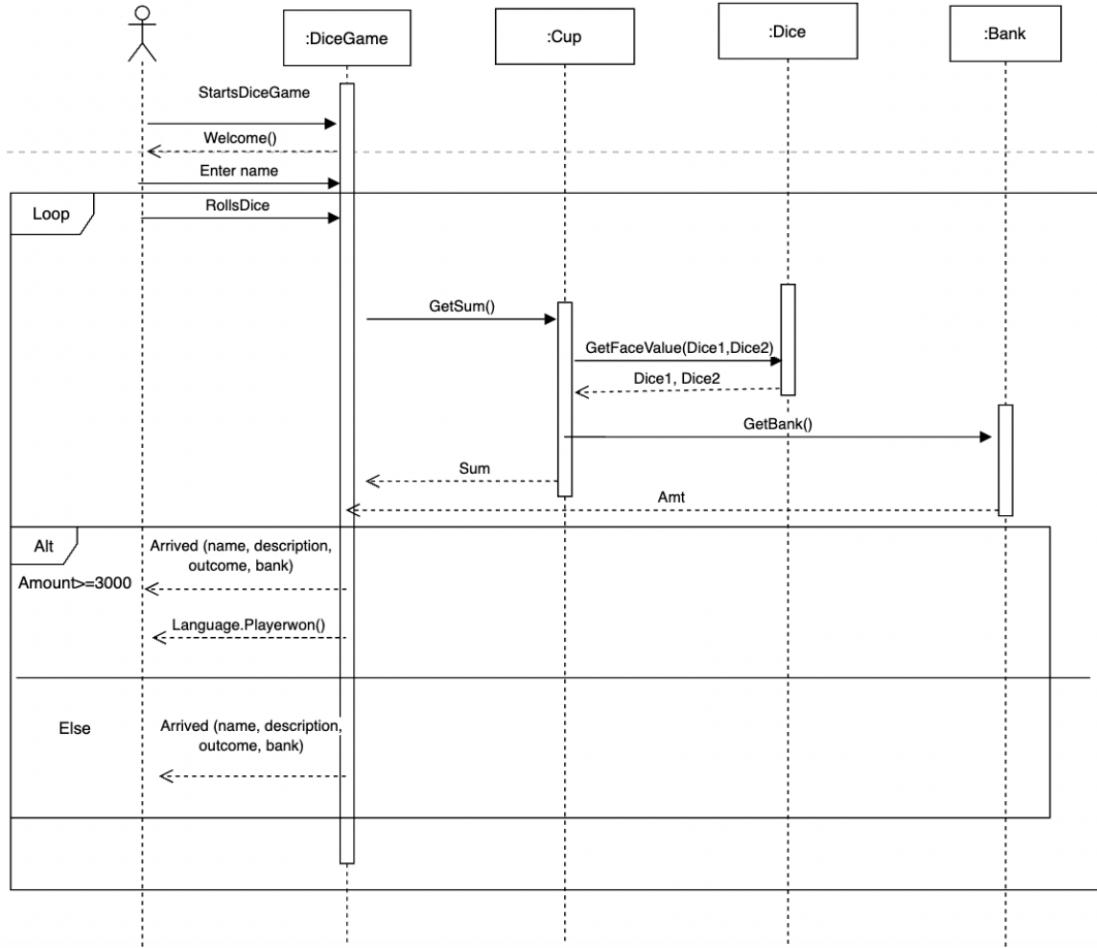
5.3 Design klassediagram

Vores design klassediagram er bygget op omkring vores main, som er tilknyttet board, cup, player og language. Det kan ses at Player-klassen er to spillere, der er koblet til banken, som har en bank til hver spiller. Die er tilkoblet cup, da cup kalder på die-klassens metoder 2 gange og lægger værdierne af de to tilfældige slag sammen. Klassen Board består af 12 felter, samt hvert felts beskrivelse, som er tilknyttet Properties. Properties viser hvilket felt spilleren lander på og resultatet i forbindelse med bankkontoen.



Figur 9: Viser vores Design klassediagram

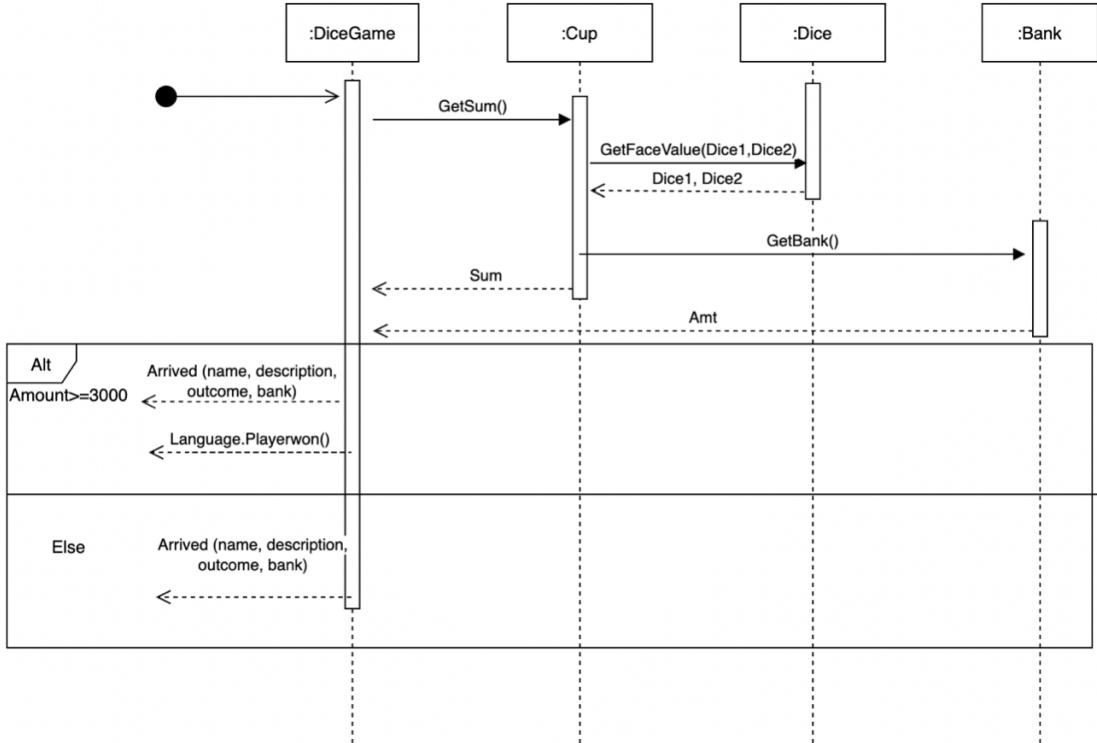
5.4 System Sekvensdiagram



Figur 10: Viser vores System Sekvensdiagram

Her kigger vi på hvordan aktøren interagerer med systemet, systemet er simplificeret og centrale operationer er illustreret. Aktør starter spillet og modtager derefter welcome metoden som er prints og bliver bedt om at taste navn, derefter ruller aktøren terningen og modtager arrived som er et print der viser balancen på aktørens konto og hvilket felt aktøren landede på. Vi har introduceret et if -else statement hvor systemet printer hvilken spiller der har vundet hvis beløbet er 3000 eller over, og et loop fra RollsDice og ned som viser at processen fortsætter indtil ønskede værdi er nået.

5.5 Sekvens diagram



Figur 11: Viser vores Sekvens diagram

Sekvensdiagrammet er simplificeret da det ellers ville blive for uoverskueligt, vi har valgt at fokusere på terningeslaget hvor vi kan se hvordan nogle af klasserne kommunikerer med hinanden. Først ses initialiseringen af processen. Systemet vil hente summen som cup klassen genererer, cup klassen henter så 2 genererede terningeslag fra dice klassen, cup klassen beregner summen af de to terninger og returnerer denne, banken returnerer den værdi som spilleren havde. Hvis hele processen skulle illustreres skulle vi også have introduceret board klassen, properties klassen, player klassen og language klassen. Men forestil at de ligger under :DiceGame Arrived findes originalt i player klassen da den er tildelt til hver spiller, og i language klassen hvor prints ligger. Til slut har vi illustreret den if statement som vi bruger til at kontrollere om spillerens amount er over 3000, hvis dette er tilfældet så printer den både spillerens samlede score og won-metoden fra language klassen.

6 GRASP-PATTERNS:

GRASP-PATTERNS: Creator er et GRASP-pattern, som fortæller hvilken klasse der skal være ansvarlig for at oprette en ny forekomst af en klasse. I vores tilfælde kunne vores **Dice**-klasse blive anset som en "creator", da man gennem Dice-klassen og metoder herunder (Eksempelvis. roll metoden) åbner op for Cup-klassen hvor metoderne skal bruges i forbindelse med at sammensætte de to terninger. Et andet eksempelvis kunne være, når **Player**-klassen formår at oprette Bank-klassen. Der kan ikke oprettes en Bank-klasse uden en Player, og dermed er dette en creator for klassen. Derudover kan vores Main-klasse anses som en creator for alle andre klasser, da inputs/methods fra alle andre klasser bliver sat i forbindelse sammen gennem vores Main. Denne forbindelse skaber funktionaliteten i programmet, og dermed ikke til at undvære - da den er med til "oprette" alle klasser.

Information expert er et af de GRASP patterns som også indgår i projektet. Properties og Board klasserne. Et princip, der bruges til at bestemme, hvor ansvarsområder såsom metoder skal fordeles til. Vores Properties-klasse initialisere netop vores Board-klasse som beskriver alle felterne og dermed er det naturligvis også information-expert - da den burde have ansvaret for den klasse. Ingen, så vil vores

Main-klasse også anses som værende information-expert - da som beskrevet er en creator for de andre klasser og holder dermed på informationer fra alle klasser - hvilket også betyder ansvaret for at programmet kan kører.

Low-coupling kunne muligvis ses i nedenstående eksempel. To elementer er netop "coupled", hvis et element har en aggregation/composition "association" med et andet element eller hvis et element implementerer/udvider et andet element. I vores Properties-klasse defineres der adskillige variabler som senere sammensættes med Board-klassen. Variablerne udvider netop beskrivelserne til hver af disse properties.

```
public class Properties {
    // variables
    private int Outcome;
    private String Field;
    private String Description;

    public Board()
    {
        properties[2] = new Properties( Outcome: +250, Field: "Tower", Description: "You got the Tower, now make it Shower");
        properties[3] = new Properties( Outcome: -100, Field: "Crater", Description: "You fell in the Crater, Sorry, See you later");
        properties[4] = new Properties( Outcome: +100, Field: "Palace Gates", Description: "You arrived at the Palace Gates");
        properties[5] = new Properties( Outcome: -20, Field: "Cold Desert", Description: "You are stranded in the Cold Desert");
        properties[6] = new Properties( Outcome: +180, Field: "Walled City", Description: "You just arrived at Walled City");
        properties[7] = new Properties( Outcome: 0, Field: "Monastery", Description: "You arrived at the Monastery, nothing happened");
        properties[8] = new Properties( Outcome: -70, Field: "Black Cave", Description: "You saw the Black Cave, go buy a torch");
        properties[9] = new Properties( Outcome: +60, Field: "Huts in the Mountain", Description: "You arrived at the Huts in the Mountain");
    }
}
```

Figur 12: Viser dele af Properties og Board class

Polymorphism handler om, at objekter af forskellige klasser har operationer med den samme signatur/attribut, men forskellige implementeringer. Dette kunne sættes i forbindelse med vores 12 felter. Felterne har samme signatur/attribut, men yder forskellige implementeringer til systemet ift. hvor pengesum der tilføjes eller trækkes fra ens bankkonto.



Figur 13: Viser polymorphism ift. feltene

7 Implementering

Vi har valgt at tage udgangspunkt i udvalgte dele af vores projekt.

7.1 Kode:

Dice: Vi har valgt at definere vores tilfældige terningeslag i Dice klassen. Den består af to private integers, en dice metode hvor vi definerer siderne på terningen, en roll klasse der viser matematikken bag det tilfældigt genererede slag, men vi har valgt ikke at definere sidelængden på terningen i denne klasse. Vi har til sidst en public int getFacevalue som kan senere skal kaldes på for at regne summen sammen. Vi har gjort således at det er nemt at ændre terningens sideantal, da vi først i cup klassen definerer at det er en seks-sidet terning og det kan til hver en tid ændres da matematikken bag det tilfældige slag gælder for enhver værdi som terningen bliver sat til.

```

package game;

public class Dice {

    private int numOfSides;
    private int faceValue;

    public Dice(int numberofSides)
    {
        numOfSides = numberofSides;

        faceValue = (int)(Math.random()*numOfSides) + 1;

    } // Constructor ends

    public void roll() // Simulation af the dice throw
    {
        faceValue = (int)(Math.random()*numOfSides) + 1;

    }

    public int getFaceValue() // Returns the value of dice
    {
        return faceValue;
    }
}

```

Figur 14: Viser vores Dice kode

Vi har valgt at implementere og designe vores **Properties** ved at definere vores variabler, Outcome, Field og Description i dens klasse. Dette bruges så i vores Board klasse. Her beskriver vi, hvert feltude fra de tre, tidligere nævnte variabler. Dette er et stort fordel for os, da vi kan genbruge koden til fremtidige projekter såsom Matador projektet. Det frembringer et overskuelig billede af koden, hvor man kan netop kan se felterne, og variablerne for sig selv.

```

Properties [] properties = new Properties[13];

public Board()
{
    properties[2] = new Properties( Outcome: +250, Field: "Tower", Description: "You got the Tower, now make it Shower");
    properties[3] = new Properties( Outcome: -100, Field: "Crater", Description: "You fell in the Crater, Sorry, See you later");
    properties[4] = new Properties( Outcome: +100, Field: "Palace Gates", Description: "You arrived at the Palace Gates");
    properties[5] = new Properties( Outcome: -20, Field: "Cold Desert", Description: "You are stranded in the Cold Desert");
    properties[6] = new Properties( Outcome: +180, Field: "Walled City", Description: "You just arrived at Walled City");
    properties[7] = new Properties( Outcome: 0, Field: "Monastery", Description: "You arrived at the Monastery, nothing happened");
    properties[8] = new Properties( Outcome: -70, Field: "Black Cave", Description: "You saw the Black Cave, go buy a map");
    properties[9] = new Properties( Outcome: +60, Field: "Huts in the Mountain", Description: "You arrived at the Huts in the Mountain");
    properties[10] = new Properties( Outcome: -80, Field: "The Werewall", Description: "You got to the werewolf wall, you have to run away");
    properties[11] = new Properties( Outcome: -50, Field: "The Pit", Description: "Oh Shit, You fell in the Pit.");
    properties[12] = new Properties( Outcome: +650, Field: "Goldmine", Description: "Jackpot, You hit the Goldmine, now you can buy anything you want");
}

```

Figur 15: Viser vores Board kode

Language: Vi har lavet en language klasse for at kunne kalde på diverse system.out.prints når der er behov for dem, det gør også at det er lettere at skifte sprog til fremtidige programmer eller

ved fremtidige opdateringer af dette program. Language klassen består af metoderne: Intro, PlayerRoll, PlayerRolled, Arrived og PlayerWon som indeholder de prints som programmet udskriver til brugeren i løbet af spillets gang.

```

public static void Intro(Player player1, Player player2) {
    System.out.println("Welcome " + player1 + " and " + player2 + " to Matador!");
    System.out.println("The first player to get 3000, wins. Good luck!");
}

public static void PlayerRoll(Player player) {
    System.out.println("\n"+player.PlayerName+", press Any Key + Enter on the keyboard if you want to roll");
}

public static void PlayerRolled(Player player, Cup cup){
    System.out.println(player + " rolled " + cup.sum());
}

public static void Arrived(String name, String description, int outcome, Bank bank){
    // Using the constructor from properties class to add the outcome of the dice throw with the system.out.println
    System.out.println("You landed on: "+name);
    System.out.println(description);
    System.out.println("The effect on your bank-account: "+ outcome);
    if (bank.amount < 0) {
        bank.amount = 0;
        System.out.println("\n"+TEXT_RED +"Your bank balance is now: " + bank.amount+ TEXT_RESET);
    }
    else {
        System.out.println("\n" + TEXT_RED + "Your bank balance is now: " + bank.amount + TEXT_RESET);
    }
}

public static void PlayerWon(Player player) {
    System.out.println("\n"+player.PlayerName+TEXT_CYAN+" WON!"+TEXT_RESET);
}

```

Figur 16: Viser vores Language kode

7.2 Konsollen

Konsollen udskriver en masse informationer vedrørende terningeslaget og feltene som der landes på, samt en opdatering på en “bank balance”. Desuden vil man til sidst i konsollen se hvem der har formået at vinde spillet. Herunder gøres brug af Language-klassen ved indsættelsen af metoder fra andre klasser såsom Board.java og Properties.

```
Welcome Ansh and Jesper to Matador!
The first player to get 3000, wins. Good luck!
```

```

Jesper, press OK on the keyboard if you want to roll
Jesper rolled 12
Total Balance: 1340
You landed on: Goldmine
Jackpot, You hit the Goldmine, now its Showtime.
The effect on your bank-account: 650

Your bank balance is now: 1340

Ansh, press OK on the keyboard if you want to roll
Ansh rolled 9
Total Balance: 1780
You landed on: Huts in the Mountain
You arrived at the Huts in the Mountain, you found some money in the Fountain
The effect on your bank-account: 60

Your bank balance is now: 1780

```

```

Ansh, press OK on the keyboard if you want to roll
Ansh rolled 2
Total Balance: 3000
You landed on: Tower
You got the Tower, now make it Shower!
The effect on your bank-account: 250

Your bank balance is now: 3000

Ansh WON!

```

Figur 17: Viser vores Console udprintning

7.3 GUI-implementation

Vi har valgt at implementere en GUI i vores JuniorMatador spil som skal fremvise et brætspil med 12 felter i alt, som alle tager udgangspunkt i de givet feltnavne og effekt (effekt ift. pengesummen man får trukket eller får tilføjet til ens bankkonto ved de forskellige felter). Vi har implementeret matadorgui-3.1.6.jar filen som konstruerer den overordnede layout for vores brætspil, dog er selve brætspillet blevet tilpasset med vores antal felter og vores feltnavne. Der gøres brug af new GUIPlayer og addPlayer metoden for at implementere vores spillere, hvilket implementeres som spil-brikker (biler)

```

public void addPlayers(Player[] players) { // Creates the player in the GUI
    players[0].GUIplayer = new GUI_Player(players[0].PlayerName, players[0].bank.amount, new GUI_Car(Color.RED,
    players[1].GUIplayer = new GUI_Player(players[1].PlayerName, players[1].bank.amount, new GUI_Car(Color.BLUE,
    board.addPlayer(players[0].GUIplayer);
    board.addPlayer(players[1].GUIplayer);
}

```

Figur 18: Viser vores AddPlayer-GUI metode

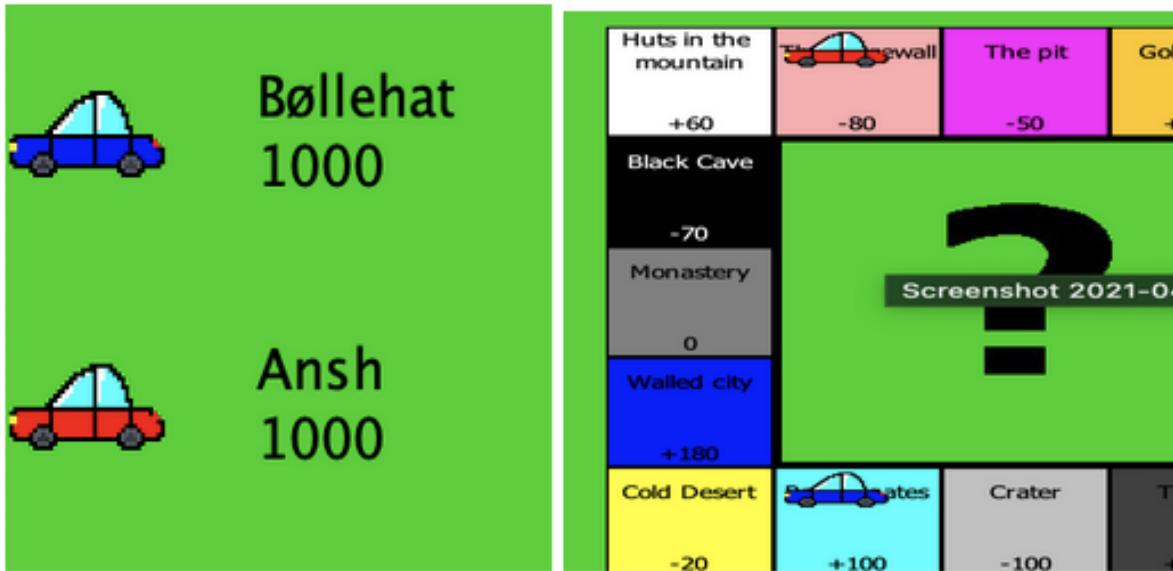
Herefter gøres der desuden brug af setCar som er sat ind i forbindelse med felt-positionerne i en MoveCar metode, som ses herunder:

```

public void MoveCar(Player player, int fieldId) { // Makes the players/cars movable in GUI
    fields[player.currentField].setCar(player.GUIplayer, hasCar: false);
    fields[fieldId].setCar(player.GUIplayer, hasCar: true);
    player.currentField = fieldId;
}

```

Figur 19: Viser vores MoveCar-GUI metode



Figur 20: Viser vores Player/Biler på GUI'en

Der gør til sidst brug af tre separate metoder, den ene opdatere vores bank saldo efter hvert terningekast, den anden opretter to sekssidet terninger og den sidste opretter en "Insert your name" knap som gemmer spillerens navn. Dette er gjort gennem setBalance(), setDice() og GetUserString() metoderne.

```

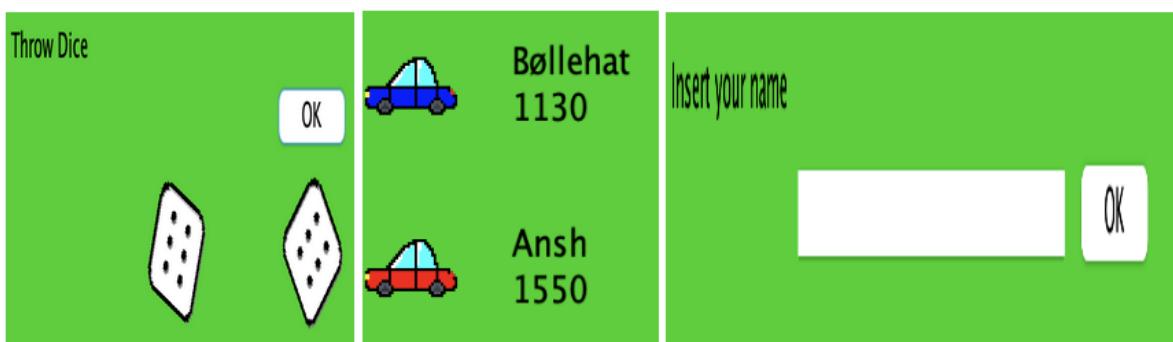
public void GUIBalance(Player player){ // updates the balance for the players after each dice throw in GUI
    player.GUIplayer.setBalance(player.bank.amount);
}

public void GUIDice(Cup cup){ // creates two dice in GUI
    board.setDice(cup.dice1.getFaceValue(), cup.dice2.getFaceValue());
}

public String GUIName(){ // Makes a Insert your name button in GUI
    return board.getUserString( msg: "Insert your name");
}

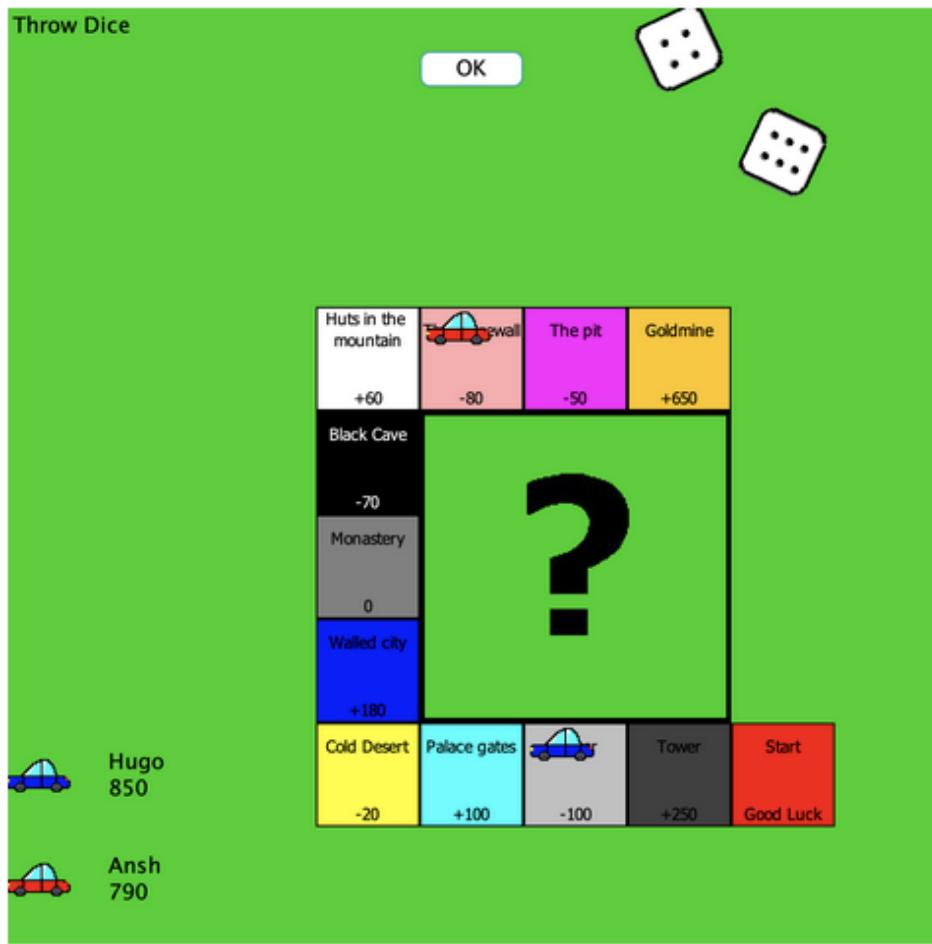
```

Figur 21: Viser tre metoder i GUIController kode vedrørende Dice, Name og Balance



Figur 22: Viser de tre metoder i GUI'en vedrørende Dice, Name og Balance

Et af vores NeedToHave krav som vedrører at spilletts felter multipliceres sammen og ud fra det bestemmer feltets nye position, altså istedet for at bilen/spillebrikken positionere sig udelukkende ud fra felt nummeret fra ens terningekast. Dette fik vi ikke opnået, men i stedet rykker spilleren sig kun fra sammen af ens terningespil og dermed forekommer der ikke et "flow" i spillet. Desuden fået vi heller ikke implementere en udprintning på vores GUI som viser hvilken spiller der har vundet, men i stedet udprintes det i konsollen.



Figur 22: Viser et overordnet syn på vores GUI

8 Test

I alle 4 test, gøres der brug af JUnit med imports vedrørende assertEquals og fail som bruges til at sammenligne vores "expected" og "actual" resultater, og for at udskrive hvis testen er forkert med fail. **Test T1:** Nedenstående test, tester om hvorvidt vores terning er begrænset. Begrænset ift. at der skal konstrueres et raflebæger med to 6-sidet terninger - hvilket naturligvis mindst kan slå 2 til højst 12. Derved gøres der brug af if-statements for at beskrive at, hvis resultatet er over 12 eller mindre 2 - så skal det udskrive en fail-statement som fremviser at der er fejl i testen. Det ender med at min assertEqualsResult og actualResult er ens, og dermed er testen korrekt.

```

package testing;
import game.Cup;
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.assertEquals;
import static org.junit.jupiter.api.Assertions.fail;

public class CupTest {
    /**
     * Testing the Cup and dice, the method should roll two dices and its combined sum should be limited from 2-12, as it is: two six sided dice
     */
    @Test
    public void test() {
        Cup cup = new Cup();
        int result = cup.sum();
        if (result > 12 || result < 2) {
            fail("The dice has not been limited");
        }
        int expectedResult = 8; //Just to declare a variable which
        if (result >= 2 && result <= 12) {
            expectedResult = result;
        }
        int actualResult = result;
        assertEquals(expectedResult, actualResult);
    }
}

```

Run: Main × CupTest.test ×

Test Results 16 ms /Library/Java/JavaVirtualMachines/jdk-11.0.1-fcs-jdk11-u11-b14-2019-07-11_12-04-b00/Contents/Home/jre/lib/amd64/jli/libjli.so

Test Results 16 ms

CupTest 16 ms

test() 16 ms

Process finished with exit code 0

Figur 23: Viser Cup-testen

Test T2: testingAddMethod() tester vores add-metode fra vores Bank-klasse, som er sat i perspektiv med et antal pengesum som indsættes i bank-kontoen når der landes på et specifikt felt. Der defineres et int variable med en værdi på 2000, hvilket skal indsættes i vores bankkonto som i dette tilfælde har værdien 0. Jeg forventer her, at resultatet er lig med 2000 og bruger igen JUnit's assertEquals for at sammenligne min forventet resultat med min faktiske resultater. Det ender med testen er korrekt, da både min forventede og faktiske resultater er ens, hvilket også betyder, at Total Balance = 2000.

```

package testing;
import org.junit.jupiter.api.Test;
import static org.junit.jupiter.api.Assertions.assertEquals;
import game.Bank;

public class BankTest {
    public BankTest() {}

    /**
     * Testing the add-amount method in the Bank class, adding an amount to the overall amount.
     */
    @Test
    public void testingAddMethod() {
        System.out.println("addEffect");
        int number = 2000;
        Bank amount = new Bank(0);
        int expectedResult = 2000;
        int actualResult = amount.add(number);
        assertEquals(expectedResult, actualResult);
    }
}

```

Figur 24: Viser Bank-testen (add amount method)

Test T3: Testen herunder tager udgangspunkt i kravet fra kunden, hvor der skulle udarbejdes en test der sandsynliggør at balancen aldrig kan blive negativ, uanset hvad hæv er. Der bruges her igen et if-statement for at sikre at hver gang vores int variable "amount" er negativ - at den ændre amount til 0. Dermed sættes min forventet resultat til 0 og min faktiske resultat til vores variable amount, og dermed vil det kunne ses om testen er korrekt - hvis begge resultater er lig 0. Hvilket det kan ses i nedenstående billede,

```

/**
 * Testing the methodArrived & Bank, the amount/balance can't be negative - no matter what the withdrawal is
 */
@Test
public void TestingArrived(){
    int amount = -6000;
    int expectedResult = 0;
    if (amount < 0) {
        amount = 0;
        System.out.println("\n"+ "Your bank balance is now: " + amount);
    } else {
        System.out.println("\n"+ "Your bank balance is now: " + amount);
    }
    int actualResult = amount;
    assertEquals(expectedResult, actualResult);
}

```

```

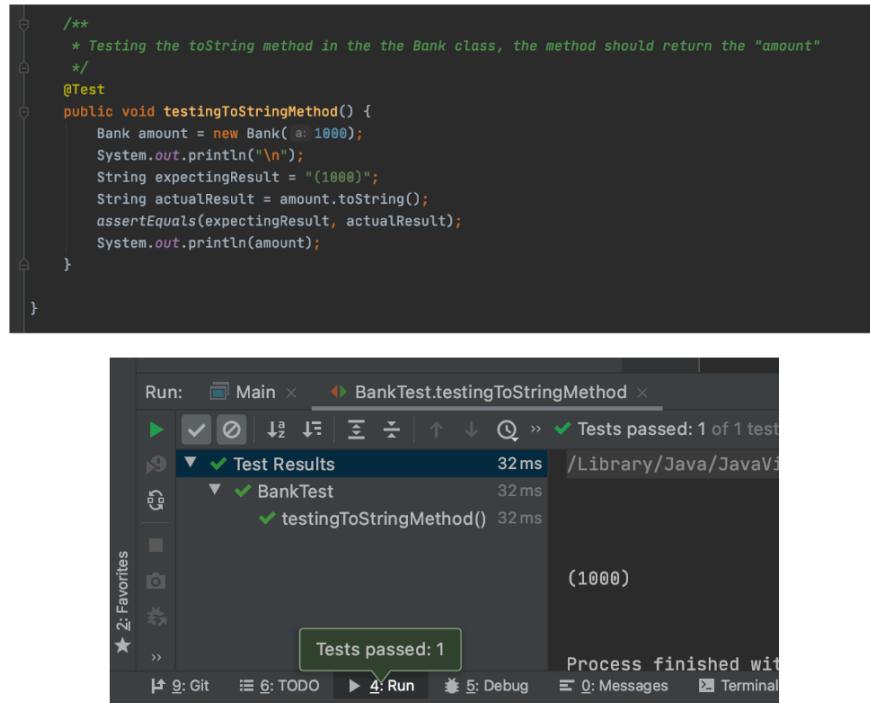
Run: Main × BankTest.TestingArrived ×
[green checkmark] Tests passed: 1 of 1 test – 19 ms
Test Results 19 ms /Library/Java/JavaVirtualMachines
  BankTest 19 ms
    TestingArrived() 19 ms
      Your bank balance is now: 0
      Process finished with exit code 0

```

Figur 25: Viser Bank.testen (tester negative banksaldo)

Test T4: Den sidste test tager udgangspunkt i vores toString metode fra Bank-klassen som skal returnere ens amount, altså en bank saldo. Herunder indsættes vores assertEquals endnu en gang, som

tester om hvorvidt vi får udskrevet 1000, da bank kontoen er sat til have værdien 1000. Vi har formået at få testen til at angive det samme resultat for både forventet resultat og vores faktiske resultat.



The figure consists of two screenshots. The top screenshot shows a portion of a Java test class:

```
    /**
     * Testing the toString method in the Bank class, the method should return the "amount"
     */
    @Test
    public void testingToStringMethod() {
        Bank amount = new Bank( a: 1000);
        System.out.println("\n");
        String expectingResult = "(1000)";
        String actualResult = amount.toString();
        assertEquals(expectingResult, actualResult);
        System.out.println(amount);
    }
}
```

The bottom screenshot shows the execution results in an IDE's run window:

Run	Test Results	Time	Path
Main	BankTest.testingToStringMethod	32 ms	/Library/Java/JavaVi
	Test Results	32 ms	
	BankTest	32 ms	
	testingToStringMethod()	32 ms	

Output pane:

```
(1000)
```

Message pane:

```
Tests passed: 1
```

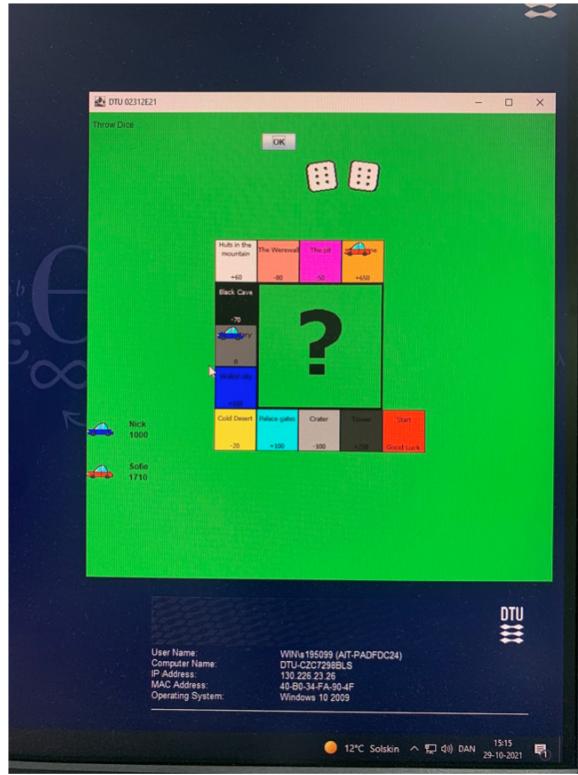
Process finished with exit code 0

Figur 26: Viser Bank.testen (tester `toString` return metode

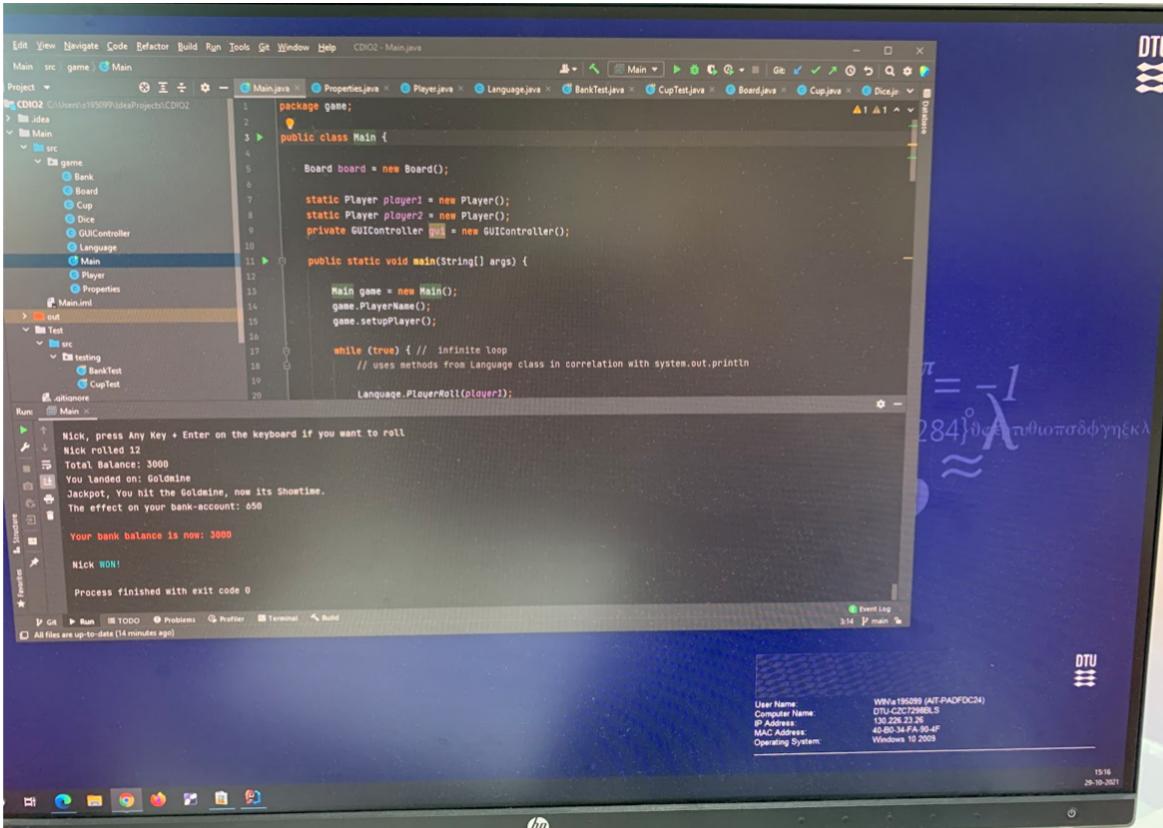
Databar-test Vi har yderligere importeret vores kode inde på databaren, på DTU. Det har været et krav, at vores program skulle kunne køre på Windows maskinerne i databarene. Vores expected result, er at programmet ville kunne åbne GUI'en og køre, indtil en af spillerne har vundet. Det kan ses på figuren, at programmet opfylder vores expected result. Vi kan konkludere at programmet kører fejlfrit på databarene

på

DTU.



Figur T5: Test af GUI på Windows-maskinerne



Figur T6: Test af programmet på Windows-maskinerne

Vi har yderligere importeret vores kode inde på databaren, på DTU. Det har været et krav, at vores pro-

gram skulle kunne køre på Windows maskinerne i databarene. Vores expected result, er at programmet ville kunne åbne GUI'en og køre, indtil en af spillerne har vundet. Det kan ses på figuren, at programmet opfylder vores expected result. Vi kan konkludere at programmet kører fejlfrit på databarene på DTU.

Traceability matrix											
Test case	Requirement number k1-k11										
	K1	k2	k3	k4	k5	k6	k7	k8	k9	k10	k11
T1		X	X							X	
T2										X	
T3										X	X
T4				X						X	
T5	X					X	X	X			
T6	X				X		X	X			

Figur 29: Viser Traceability matrix

9 Konklusion

Det kan nu konkluderes, at vi har opnået at udvikle et fungerende Juniormatador for 2-personer, bestående af 12 felter. Vi har valgt at fokusere på objekt-orienterede programmering og dermed består koden af ni forskellige klasser der hver har en afgørende rolle for programmets kørsel. Vi har formået at opfylde de opstillede krav til projektet, såsom at spillet skal kunne bruges på Windows maskinerne i databarerne på DTU, en afprøvningskode, der sandsynliggør at balancen aldrig kan blive negativ og, at alle "almindelige" mennesker kan spille vores spil uden en brugsanvisning. Vi har desuden implementeret en "NiceToHave" funktion ved at benytte GUI'en, for at give et visuelt billede til os selv og kunden. En "NicetoHave" funktion vi gerne ville have haft med var punkt nr. 15 at "Efter hvert slag multipliceres terningeslaget til det gamle landte felt", istedet vil vores Player lande på feltet der matcher terningeslaget hver gang.

10 Bilag

Her er linket til vores github: <https://github.com/MarcoHansen7/CDI02.git>