Danmarks Tekniske Universitet

62531 Development Methods for IT Systems
62532 Version control and test methods
02312 Introductory Programming

CDIO 2

29th October 2020

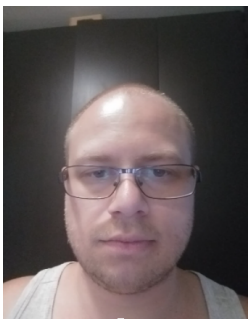
Lucas Arleth Lykke
s205447

Mike Patrick Nørlev Andersen
s205417

Sebastian Andreas Almfort
s163922

Martin Koch
s182935

Anne Sophie Bondegaard Petersen
s194582



Jan Engers Møller Pedersen
s205419

Resumé

På baggrund af vores tidligerer projekt, har **IOOouterActive** andmodet om at udvikle et nyt system. Her skal vi tage udgangspunkt i vores tidligere projekt, da det anmodet system skal være et spil der benytter sig af terninger igen.

Indholdsfortegnelse

1 Timeregnskab

Timer	
Patrick	10
Martin	10
Sebastian	10
Lucas	10
Jan	10
Anne	10

Tabel 1: Time tabel

2 Indledning

I dette projekt har vi fået til opgave, at designe, implementere og teste et brætspil mellem to personer.
Under udarbejdelsen af projektet vil vi benytte UML til at udforme kravsspecificering, Use Case beskrivelser, artefakter og CASE teori, til at danne et godt overblik over udviklingen af produktet, og for at kunne udvikle dette spil bedst muligt. Dette lader også således opfylde de tildelte krav bedst muligt.

3 Kravspecifikation

3.1 Krav liste

I dette kapitel er de tildelte krav blevet analyseret, gennemgået og sat op, således at der er bedre overblik over hvilke krav der er opfyldt og hvordan. Dette fortæller både os og klienten at vi har gennem udviklingen, fået testet og gennemgået alle kravene, således at det anmode system opfylder klientens behov.

Krav	Kommentar	Overholdt / Ikke overholdt
R1	Kan bruges på Windows computer	✓
R2	Spil mellem 2 personer	✓
R3	Spillet går ud på at man slår med et raflebæger med to terninger. Lander derefter på et felt med numering mellem 2 til 12, hvor der sker noget specifikt for det pågældende felt. Skal se resultatet med det samme.	✓
R4	Der skal stå på hvert felt, hvad det pågældende felt gør. Vælger selv, men det skal havde noget at gøre med den overskrift for feltet der er givet i opgavebeskrivelsen.	✓
R5	Spillerne slår på skift	✓
R6	Vinderen er den, der opnår 3000 i pengebeholdning	✓
R7	Kravene for de forskellige felter er alle overholdt	✓
R8	Spiller starter med pengebeholdning på 1000	✓
R9	Spillet skal let kunne oversættes til andre sprog	✓

Krav	Kommentar	Overholdt / Ikke overholdt
R10	Skal let kunne skifte imellem forskellige terninger	✓
R11	Spillerne og pengebeholdningen skal let kunne bruges i andre spil	✓
R12	Spilleren kan ikke slå med 1 terning af gangen	✓
R13	Hvis en spiller slår 10. Så mister spilleren 80 point, men får en tur mere.	✓

Tabel 2: Liste over krav

3.2 Krav matrix

For at få overblik over hvorvidt vores specifikke cases opfylder de tildelte krav, har vi konstrueret en tabel, der markerer hvilke cases opfylder hvilke krav. Her kan ses at alle kravene bliver opfyldt. Eftersom vores cases gør mere eller mindre det samme, giver det meningen at største delen af kravene bliver opfyldt samtidigt

Requiments/ Use cases	U1 ID: 1	U2 ID:1.1	U3 ID: 1.2
R1			
R2			
R3			
R4			
R5			
R6			
R7			
R8			
R9			
R10			
R11			
R12			
R13			

Tabel 3: Requirements Tracing Matrix

4 Analyse

4.1 Use Cases overblik

For at kunne udarbejde vores Use Cases har vi opstillet en tabel, for at få overblik over de aktører, der interagerer med vores system. Ved hjælp af dette har vi skabt overblik og struktur over vores systems opbygning.

Use cases / aktører	Spillere (Abstrakt)
Ligge terninger i bærger	X
Ryste bærger	X
Slå med terningerne	X

Tabel 4: Aktører i spillet

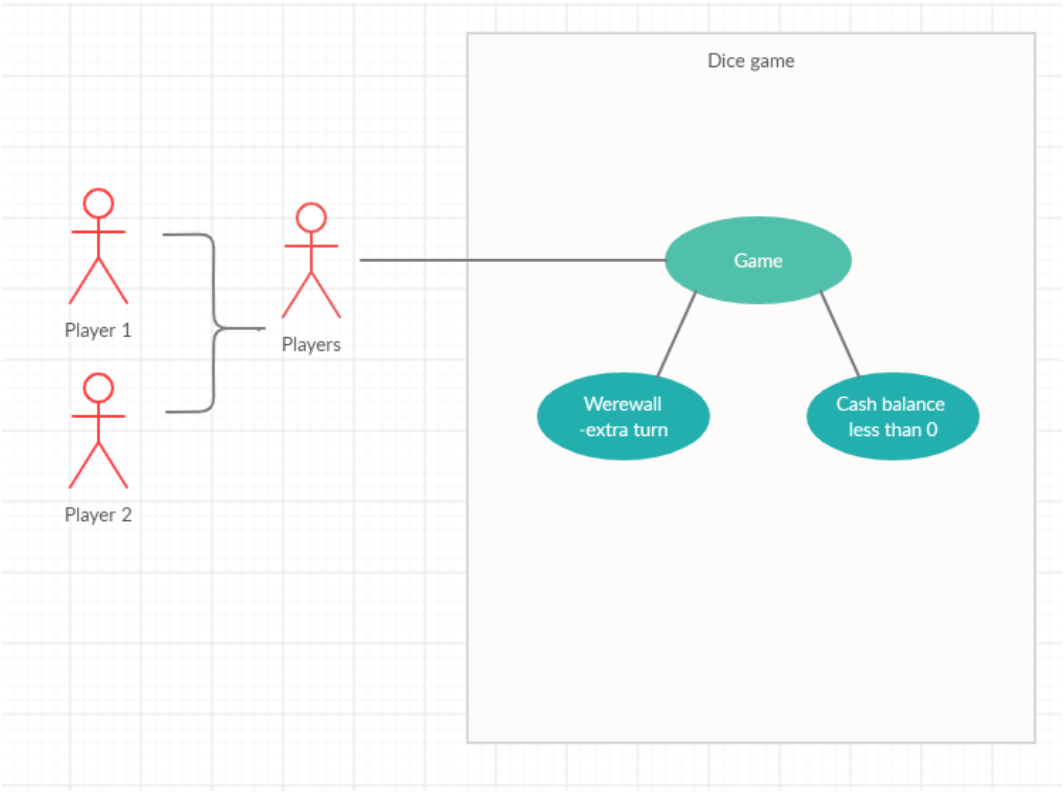


Figure 1: Use Case Diagram

4.2 Use Cases

Use case: Spil
ID: 1
Kort beskrivelse: Spillerne spiller spillet til en af spillerne vinder
Primary actors: Spiller 1 Spiller 2
Secondary actors: Ingen
Preconditions: Spillerne har 1 bæger og 2 terninger
Main flow: 1. Spiller 1 slår med terningerne (a) Spiller 1 lander på et felt og Spiller 1's sum korrigeres efter det pågældende felts værdi (b) Spiller 1 giver bægret og terningerne videre til Spiller 2 2. Spiller 2 slår med terningerne (a) Spiller 2 lander på felt og Spiller 2's sum korrigeres efter det pågældende felts værdi (b) Spiller 2 giver bægret og terningerne videre til Spiller 1 3. Dette gentages indtil en af spillerne har fået 3000 point eller over
Postconditions: Ingen
Alternative flows: 1. Spiller lander på felt 10 2. Spilleren lander på et felt og mister flere penge der er i pengebeholdningen

Tabel 5: Use Case for spillet

Alternativ flow: Spiller lander på felt 10
ID: 1.1
Beskrivelse: En spiller slår et slag, hvor summen af øjnene giver 10. Spilleren mister derved 80 point, men får samtidig en ekstra tur
Primary actors: Spiller 1 Spiller 2
Secondary actors: Ingen
Preconditions: Spilleren er landet på det 10'ene felt
Alternativ flow: 1. Dette alternative flow påbegyndes efter en af spillerne er landet på felt 10. 2. Spilleren der er landet på felt 10 mister 80 point, men får en ekstra tur
Postconditions: Spillet fortsætter herfra som beskrevet i main flow fra punkt 1 eller 2, alt efter hvem der har fået en ekstra tur

Tabel 6: Alternativ flow 1

Alternativ flow: Tom pengebeholdning.
ID: 1.2
Beskrivelse: En spiller lander på et felt, hvor spilleren mister flere penge end spilleren har i sin pengebeholdning.
Primary actors: Spiller 1 Spiller 2
Secondary actors: Ingen
Preconditions: Spillerens pengebeholdning bliver tømt eller forbliver tom.
Alternativ flow: 1. Spilleren slår med terningerne. 2. Spilleren lander på et felt, hvor spilleren mister flere penge, end spilleren har i sin pengebeholdning. 3. Spillerens pengebeholdning er 0
Postconditions: Spillet fortsætter i main flow punkt 1 eller punkt 2. Hvis spilleren landede på felt 10, så fortsætter spillet i alternativ flow ID 1.1 punkt 1.

Tabel 7: Alternativ flow 2

4.3 Domænemodel

Domænemodellen definerer de overordnede elementer i spillet.

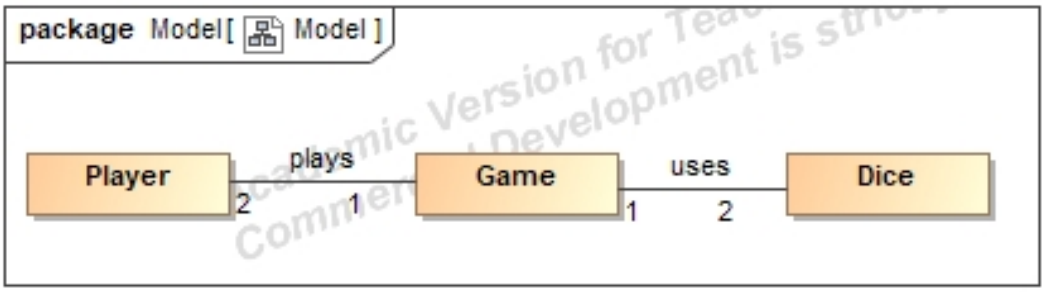


Figure 2: Domænemodel

4.4 Systemsekvensdiagram

Vi har udformet et systemsekvensdiagram, der giver et overordnet overblik over systemets elementer.

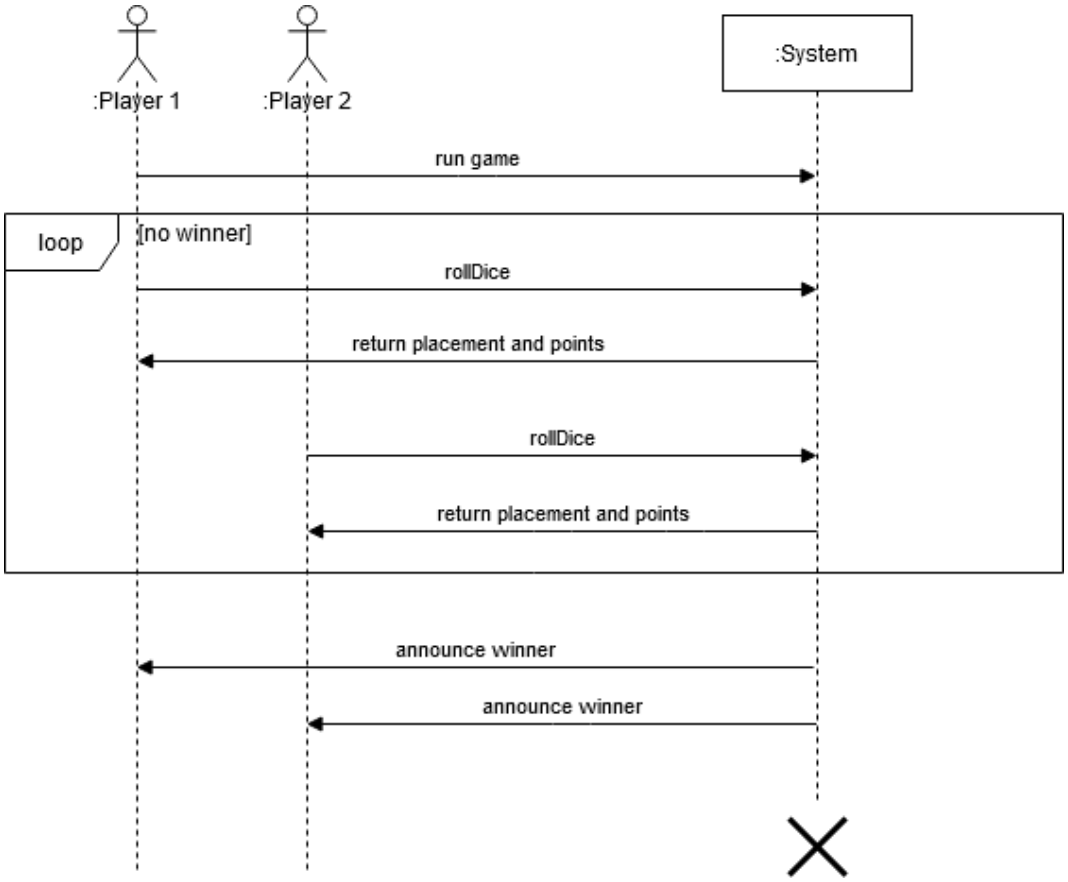


Figure 3: Systemsekvensdiagram

Det ses, at spillet startes af en spiller, og spillerne derefter på tur ruller terningerne, og pointbeholdningen, samt placering returneres af systemet.
Når spillet slutter ved en bestemt pointbeholdning, annonceres vinderen, og spillet slutter.

5 Design

Under udvilking af vores system har vi taget udgangspunkt i systemets design, flow og struktur, til at gøre det nemmere at skabe det ideelle system.

5.1 Sekvensdiagram

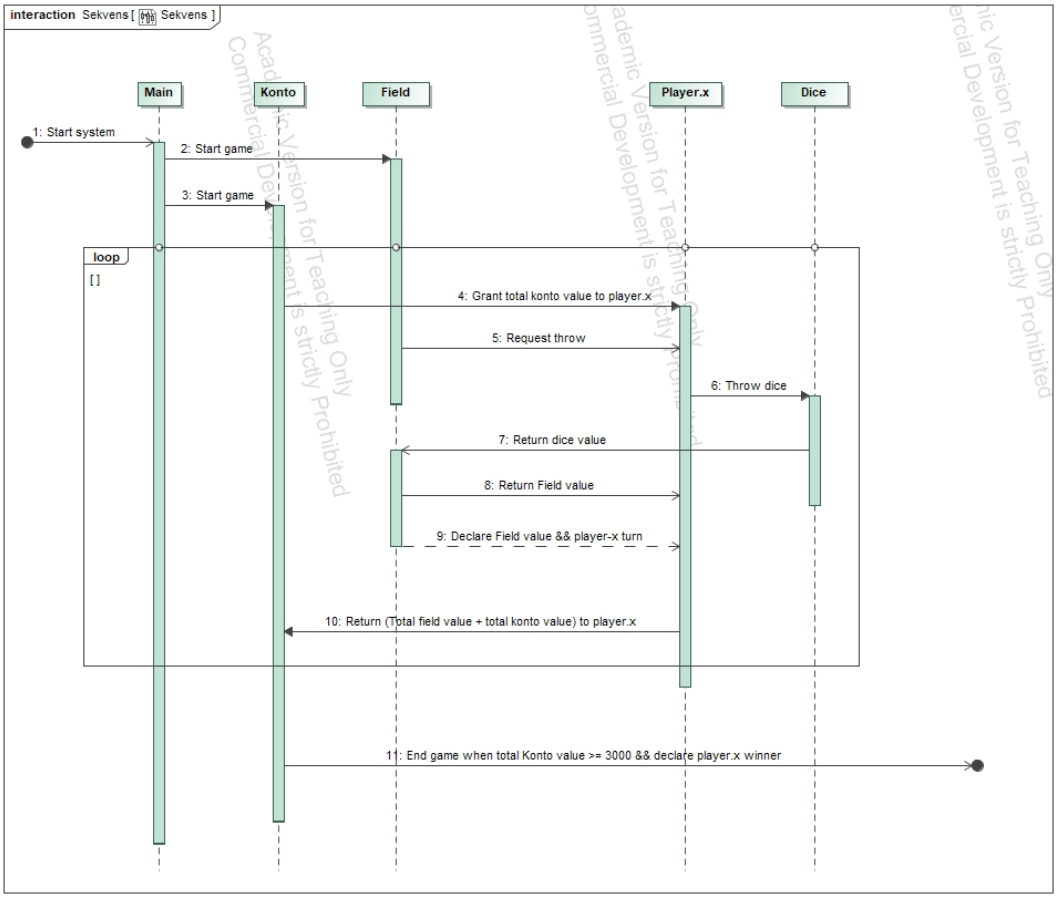


Figure 4: Sekvensdiagram

Ved hjælp af sekvensdiagrammet, på Figure 4, bliver der skabt et overblik over hvad interagerer med hinanden hvornår. Som eksempel kan det ses at de der interagerer mest med hinanden er Field og Player.x
Eftersom spillet bygger på at en aktør, som vi kalder "Player", netop spiller spillet ved at benytte sig af "Dice", hvis rolle er at lade spilleren interagere med spillets fundamentale objekt "Field", giver det mening at størstedelen af interaktionerne foregår mellem de to "Player.x" og "Field".

5.2 Designklassediagram

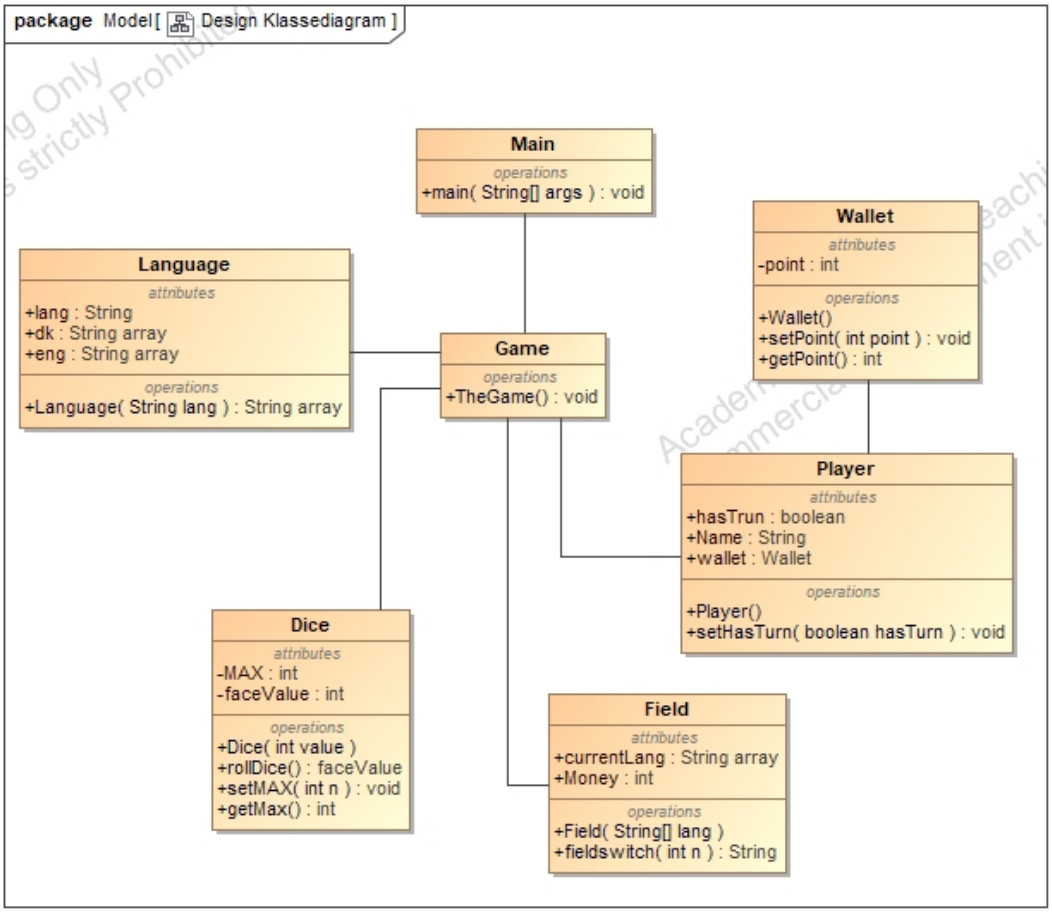


Figure 5: Design klassediagram

5.3 G.R.A.S.P

Ved at lægge G.R.A.S.P konceptet ned over modellerne som koden er skrevet ud fra, kan designet af modellerne og derved også koden gennemgås, og det kan give en mulighed for at se hvad der bør forbedres, for at programmet kommer til at virke mere optimalt.

5.3.1 Creator

I systemet er der to klasser der kan beskrives som værende creator for objekter. Disse to klasser er **Game** og **Player**.
Game står for at oprette objekter for klasserne **Language**, **Dice**, **Player** samt **Fields**. Det er **Game** som alle andre klasser går igennem når de kommunikerer med hinanden. Den sidste klasse **Wallet** har klassen **Player** som creator. For at kunne komme til

Wallet og for at objekterne til denne klasse kan blive createt, er det nødvendigt at gå igenne klassen **Player**.

5.4 Information expert

Da koden er skrevet før gruppen har haft viden om GRASP, har gruppen ikke været skarpe nok omkring hvilket klasser der står for at fuldfører specifikke funktioner. Internt i gruppen vides nu, at G.R.A.S.P. er noget der skal tænkes ind i et projekt fra starten.

5.4.1 Coupling

System har high coupling. Dette ses ved at alle klasser på nær **Wallet** er afhængige af klassen **Game**. Som systemet er bygget op nu, går alt kommunikation i system via **Game**.
Ulempen ved at havde high coupling i et system, er at udvidelser af systemet bliver det sværer at implementere nye funktioner. Hvis man ønsker at tage dele af sin gamle kode og overførere det til at andet projekt, er det svært at gører, da alt i det nuværende system er så sammenflettet at en klasse alene ikke kan kører en funktion.

5.4.2 Cohesion

Systemet har low cohesion. Dette ses ved at ansvaret for de forskellige klasser er svært at definere. Programmet er skrevet på en måde, således at flere klasser arbejder sammen om funktioner. Det vil sige at de fleste af de funktioner som systemet arbejder med, ikke kan udføres af en enkel klasse.
Derfor er det også svært at lave en endelig definition på, hvilke klasser der har det overstående ansvar for de forskellige funktioner.
Til et fremtidigt projekt er det nødvendigt at klart definere hvilke klasser der har hvilket ansvar, inden at man begynder at implementere koden.

5.4.3 Controller

I dette system er controller klassen **Game**. Det er denne klasse der håndtere alle input samt står for kommunikationen mellem alle klasserne. Grundet at alt kommunikation kun går igennem **Game** klassen, er dette også med til at gøre, at system har en high coupling.
Dette vil gøre, at hvis man for redigeret controlleren således, at der kommer til at være flere controllere der står for input, og hvor kommunikationen kommer igennem, vil man også automatisk kommer over imod at få en lower coupling, hvilket også vil kunne gøre det nemmere at definere hvilke klasser der står for hvilket ansvar som i sidste ende derved

ville kunne give en higher cohesion.

6 Implementering

I den første del af implementerings delen vil der blive drøftet om hvordan opgaven er håndtaget på baggrund af de punkter der er uklare i opgaveformulering. De uklare ting har været oppe og vende med hjælpelæren hvilket beskrives i dette afsnit og hvordan de er blevet håndteret.

• Det står uklart til om hvorvidt spilleren rykker tilbage til felt nummer 0 eller bliver på det felt spilleren lander på.
I implementering er der blevet valgt at spilleren rykker tilbage til felt nummer 0. Eksempel: Du står på felt nummer 0. Du slår en 6'er. Du rykker frem til felt nummer 6 får pengene fra feltet. Du rykker tilbage til felt nummer 0.

• Det skal være let at skifte til andre terninger problem 1.
Når man kan skifte til andre terning går vi ud fra at terningerne kan have X sider.

```
d1.setMAX(toint); //initiates how many sides the dices in the game should have.
d2.setMAX(d1.getMAX());
```

Figure 6: terningeside initalisering

Dice klassen indeholder en setter og en getter for Max værdien af terning, på den måde kan spilleren selv vælger hvor mange sider terningerne skal have. Det skal pointeres at terningerne altid har samme værdi.
Ydermere er der lavet en funktion som sørger for at spillet ikke crasher når spilleren indtaster hvor mange sidder terningerne skal have. Inden implentationen ville spillet crashe, hvis man skrev bogstaver, tegn eller alt for høje tal. Vores løsning til dette problem er i følgene kode:

```
while (true)
{
     diceScan = scan.nextLine();
    try {
        toint = Integer.parseInt(diceScan);
        if (toint > 0 && toint <= 1073741823)
        {
            break;
        }
        else
        {
            System.out.println(currentLang[21]);
        }
    }
    catch (NumberFormatException e)
    {
        System.out.println(currentLang[20]);
    }
}
```

Figure 7: bruger input checker

Maks værdien for en enkelt terning er 1073741823. Der bruges to terninger så 1073741823*2 = 2.147.483.646 som er den absolutte højeste værdi for en integer. Med denne implementation vil der ikke forekomme overflow lige meget hvad brugeren har tænkt sig at indtaste. Hvis brugeren indtaster en "værdi" forskelligt fra et "numeric" tal returner den en NumberFormatException og brugeren får lov til at prøve igen.

• Det skal være let at skifte til andre terninger problem 2.
Problem forklaring:
Med implementering hvor man selv kan vælge hvor mange sider terningen skal have kommer problemet at vi når "out of bounce" med hensyn til spilbrættet. Brættet indeholder som sagt kun 12 felter men nu har vi mulighed for at slå for eksempel 40.
Problem løsning implementering:
Når man slår et tal over 12, rykker man tilbage til felt nummer 2. Eksempel: Jan slår 25. Jan lander på felt nummer 2.

```
if (dicevalue > 12) { //it is possible to
    if (dicevalue % 12 == 0) {
        dicevalue = 12;
    }
    else {
        dicevalue = (dicevalue % 12) + 1;
    }
}
```

Figure 8: Modulus af terningesum

Som man kan se i koden ovenover tjekkes summen for de 2 terninger. Hvis terningesummen er over 12 bliver den nye sum modulus af den gamle sum +1. +1 er til fordi brættet ikke indeholder felt nummer 1. Hvis man slår 24, 36, 48 osv.. lander du på felt nummer 12.

• Vi vil gerne have at I holder jer for øje at vi gerne vil kunne bruge spilleren og hans pengebeholdning i andre spil.
Vi har gemt spillerens pengebeholdning i spiller klassen, så man kan se hver spilleres pengebeholdning. Koden i klassen som laver et nyt waller objekt når en spiller bliver initaliseret: Wallet wallet = new Wallet();.

• Spillet skal let kunne oversættes til andre sprog.

Der er lavet en metode som returner en array-string med det valgte sprog.

```
public Language(String lang) { this.lang = lang; }
//Method that takes in a language choice as a string and
public String[] returnLang() {
    switch (this.lang){

        case "en":
            return this.eng;

        case "dk":
            return this.dk;

        default:
            return this.eng;
    }

}
```

Figure 9: metode for valg af sprog

Med denne implementation gøres det nemmere at enten tilføje flere sætninger til spillet på samme sprog, eller tilføje et helt nyt sprog. Det er nemmere da arrayet indeholder en consol output for hver index, så hvis man vil tilføje et nyt sprog skal man bare oversætte hver index i arrayet 1:1, mere behøver man ikke. Nedenfor ses et eksempel for arrayet som bliver initaliseret hvis man vælger "eng".

```
//array that store strings in english
String[] eng = {"Please enter the amount of sides you wish your dice to have: ", "Enter player 1'
        "Enter player 2's name: ", " Enter: 'roll' to roll the dice", "roll", " Rolls: ", " Point
        "You win!", "End of game.", "You fell in a crater and have to pay to get out. You pay 100
        "You stand in front of the palace gates and find a gold coin. You sell it for 100",
        "You ended up in a cold desert. To keep warm you buy a blanket for 20",
```

Figure 10: Engelsk array

7 Test

7.1 Sandsynligheds test

Da terningen klassen er identisk med den brugt i CDIO1. Så vil vi referere til testen af denne i foregående projekt.

7.2 Minus test

Vores pengebeholdning håndteres af Wallet klassen. Den har vi testet for at den ikke kan gå i minus. Dette gøres gennem dens setPoint() metode. Den metoden modtager et heltal og lægger dette til pengebeholdningen. Den er testet for ikke at gå i minus, ved at man lægger et negativt tal til, hvis numeriske værdi er større end den nuværende pengebeholdning. Hvis det virker så kan man forvente et ouput på 0. Dette har testen bekræftet.

7.3 Dice test

Terningen er blevet testet i forhold til, at den ikke ruller uden for dens ønskede talmængde. Altså skal terningen ligge fra og med 1 til og med MAX. Dette er gjort ved at teste 100 kast med en assertTrue fra junit. input til assert true er blot et boolsk udtryk, der er sandt hvis den ligger inden for talmængden. Videre har vi forsøgt at gøre mængden blot 1 tal for småt og der fejler den hver gang. Dette var for at sikre at testen fungerede som forventet.

7.4 Field test

Field klassen i vores software er ansvarlig for at tildele det rigtige antal point til et specifikt slag. Vi har 11 forskellige mulige slag. De 11 slag repræsentere 11 forskellige ækvivalensklasser som alle skal testes. Dette er gjort ved at indsætte et slag i fieldswitch() metoden og sikre at det stemmer overens med forventningen. Dette er gjort for samtlige muligheder. Da vi har en switch til at styre det, så vil alle andre værdier blot ramme default casen. Vi har brugt modulus operatoren til at undgå at komme uden for den ønskede talmængde.

7.5 Sprog test

Language klassen er ansvarlig for at returnere et String[] som indeholder alle strings, der bruges i programmet på et givet sprog. For at sikre at den returnerer det forventede array, har vi kopieret de arrays som language klassen indeholder, og derefter tjekket dem 1 efter en så man kan sikre sig at metoden returnLang() gør det forventede. Jeg har sammenlignet de 2 arrays string for string og talt en counter op hvis de ikke var ens. Derefter har jeg brugt assertEquals(0, counter). Denne ville give false hvis en af sammenligningerne fejlede.

7.6 Player test

Vi har en spiller klasse som creator til Wallet. Den står for at holde styr på hvilken spiller der har tur. Den har metoden setHasTurn() som modtager en boolsk værdi og ændre en intern variabel, kaldet hasTurn. Denne er testet ved at oprette en spiller, som altid har false i den hasTurn. Først med assertTrue(false, hasTurn), derefter kalder man setHasTurn(true). Så bruger man endnu en assertTrue(true, hasTurn). Hvis begge disse går gennem så virker både konstrukter samt metoden setHasTurn().

8 Konfiguration

8.1 Minimums Systemkrav

- CPU: Pentium 2 266 MHz processor
- Ram: 128 MB
- Lagerplads: 5 MB
- Programmer: Java JRE 8, Git, Intellij Idea 2020.2.1

8.2 Installation

Første trin til brug af dette program vil være at hente det fra git.
Dette kan gøres ved at man henter en klon fra https://github.com/zPaddyz/CDIO2 ved hjælp af Intellij, det skal siges at brugeren skal have adgang til projektet.

8.3 Kompilering

Programmet kompilere ved hjælp af Java Virtual Machine som bruger klasse filerne til at oversætte det til maskinesprog derfor vil Java JRE 8 være en nødvendighed.

8.4 Afvikling

I git repositoriet er der en .Jar som man kan gøre brug af for at starte programmet, dette kan enten gøres igennem enten Intellij eller kommando prompt-en.

- Intellij:
  For at køre programmet igennem Intellij højreklikkes CDIO2.jar filen og der trykkes run.
- Kommando Prompt:
  For at køre programmet igennem kommando prompt-en så skal man først navigere til .jar-filens placering ved brug af CD (Indsæt filepath). Derefter skal der skrives java -jar CDIO2.jar for at starte programmet.

Alternativt kan programmet startes ved brug af Main-klassen, for at gøre brug af denne løsning højreklikkes Main-klassen og der trykkes run eller man kan åbne Main-klassen og trykke Shift + F10 i IntelliJ.

9 Projektplanlægning

Gennem forløbet af projektet har vores intern planlægning og opgave fordeling været bygget på fleksibilitet og kommunikation. Her har vi som ekesmpel udnyttet mulighed for langdistangses kommunikation ved bla. at benytte applicationer så som Discord og Messenger. Eftersom det ikke er alt gruppe arbejde der fungerer optimalt over længerer distancer, har vi også arrangeret møder, hvor vi har benyttede os af muligheden for at reseverer lokale i DTU's bibliotek.
Som udgangspunkt benyttede vi os af selve projektets foreslået rækkefølge af løsningerne til opgaverne. Det vil sige at vi startede med at udarbejde diverse analytiske og design messige aspekter af projektet, for derefter benytte dette fundament til at udvikle vores system.

10 Konklusion

Alt i alt har vi udviklet et system der virker og som overholder alle de opgivet krav. Ved gennemgang af projektet med G.R.A.S.P. kan det ses, at der i systemet er plads til optimering på flere aspekter.
Yderligere har vi benyttet diverse test metoder til at dokumenterer at systemet forbliver velfungerende.