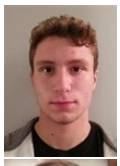
CDIO 2

02312, 02313 og 02315
INDLEDENDE PROGRAMMERING, UDVIKLINGSMETODER TIL IT-SYSTEMER OG VERSIONSSTYRING OG TESTMETODER

Dato: 1. November 2019

LAVET AF



Oliver Poulsen
s185112

Yassine Eljilali
s195390

Thomas Hohnen
s195455

Frederik Koefoed
s195463

Jacob Eriksen
s195471

Dánjal Dávidsson
s195472

Abstract

This report describes the process of making a software-based dice game using object-oriented programming, UML-tools and JUnit-testing. The central feature of the game is that it includes two players who take turn rolling two dice. The sum of the dice determines how many points they will obtain or lose. The first player to reach 3000 points wins.

Models and diagrams within the universal modeling language (UML) have been used throughout the project and are also shown and described in this report. UML-models with different levels of detail have shown to be a great way to analyze the problem and design code before writing it. Additionally, the models can be updated along the way making it easier to understand the written code.
The software code is written in the coding language Java which is centered around object-oriented programming. By dividing elements and features of the game into classes it is easier to understand as it resembles real life objects.

The conclusion for the report is that the tools used in this project are quite suitable to resolve the given software problem. With them it has been possible to analyze the problem, design the solution and implement it as code. The program created runs without any problems and with all demands implemented.

Indhold

1 Indledning

I denne rapport gennemgås udviklingsprocessen for et terningespil til en fiktiv kunde. Kundens krav til spillet inkluderer f.eks., at spillet er mellem to personer, som slår med to terninger, og forsøger at opnå 3000 point før modstanderen. Rapporten indeholder både beskrivelse af analysen, kravene, designet af programmet og implementeringen af koden. For at sikre kvaliteten af spillet, er det også blevet testet.

Projektet er tværfagligt og derfor er der flere formål med projektets udførelse. Overordnet er formålet at vise duelighed indenfor objekt orienteret programmering. Dette indeholder brug af forskellige modeller og diagrammer fra UML, planlægning og samarbejde i gruppen, brug af versionsstyrings-værktøjet github og brug af metoder til testning af programmet. Projektet er kodet i programmet IntelliJ med det objektorienterede kodesprog Java. Testningen af projektet er lavet med JUnit-tests.

2 Analyse og Krav

2.1 Kravspecifikation

Funktionelle Krav

1. Der skal være 2 spillere.
2. Spillet slutter når en af spillerne har opnået en pengebeholdning på 3000.
3. Spillerens balance må ikke kunne gå i minus.
4. Terningens sum bestemmer hvilket felt man lander på.
5. Ved hvert felt vindes eller tabes et antal point

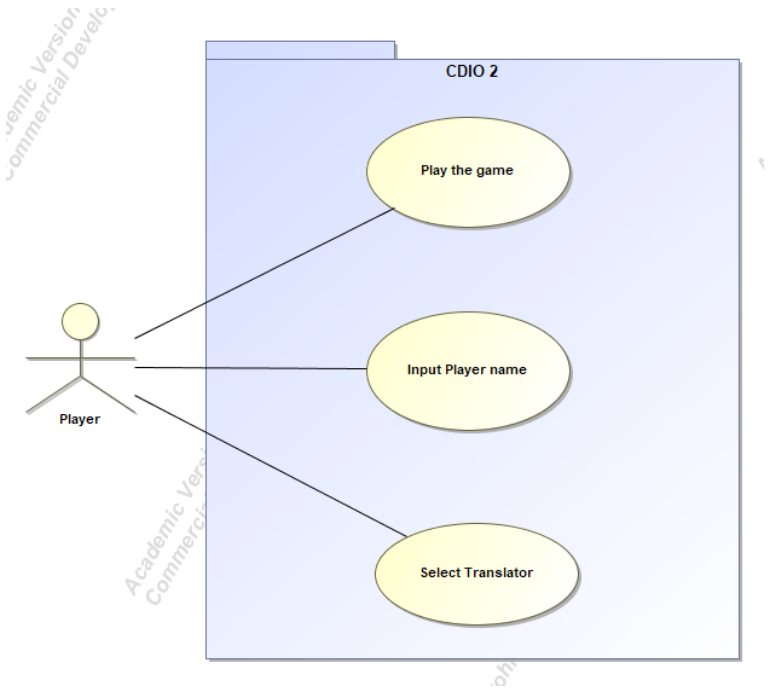
Ikke Funktionelle Krav

1. Spillerne skal kaste med 2 terninger.
2. Spillet skal let kunne spilles på andre sprog.
3. Spillerne starter med en pengebeholdning på 1000.
4. Hvert felt skal udskrive et scenarie for, hvorfor man vinder eller taber point.

2.2 Use cases

Lise over Use cases med en brief beskrivelse

1. **Play the game:** Spilleren slår med terningerne og derefter giver turen videre til den anden spiller. (Bliver også beskrevet i fully dressed form)
2. **Input player name:** Spilleren indtaster sit navn i systemet, så der kan kendes forskel på spiller 1 og spiller 2
3. **Select Translator:** Først skal spillerne enes omkring hvilket sprog spillet skal forgår på, dansk, engelsk, fransk osv.



Figur 1: Use Case Diagram

Til dette projekt er blevet lavet nogle forskellige Use cases. For at ilustrere dette er der også blevet lavet et Use case diagram. Til et terninge spil som dette er der blevet fundet 3 forskellige use cases. "Play the game", "Input player name"og "Select Translator". Disse 3 forskellie use cases opbygger hvordan spillet spilles for brugerens synsvinkel.

2.3 "Play the game" i fully dressed beskrivelse

Play the Game

Scope: IOOuterActive
Level: User goal
Primary actor: Player
Stakeholders and intrest: Player vil gerne kunne spillet spillet med en ven.
Preconditions: Spilleren starter spillet
Success guarantee: Spillet blev færdigt. Vinder blev fundet.
Main success scenario:

- Spillet starter.

Extenstions:

1. Spillet starter ikke.
2. Spillet bliver lukket ned og åbnet igen.
3. Spillet starter stadig
4. udbyder bliver kontaktet

Special requirements:
Til dette projekt er der ingen specille krav.

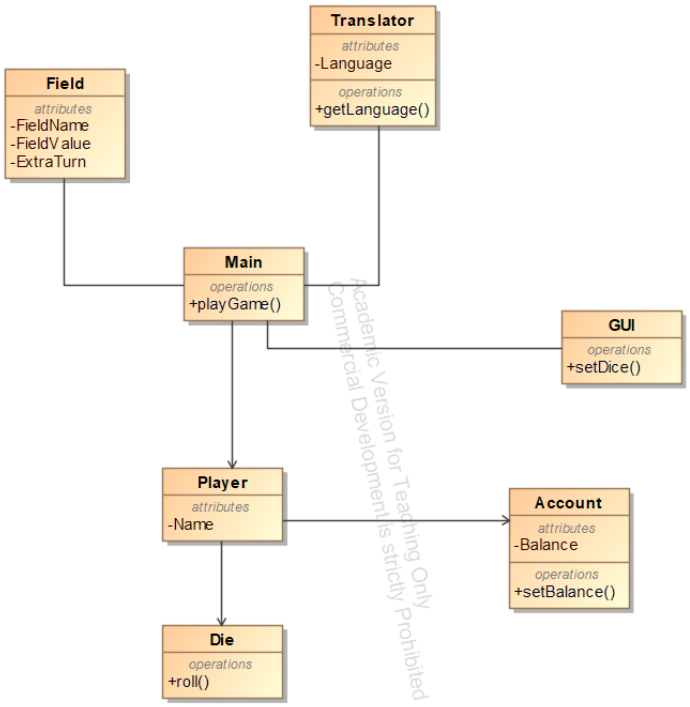
Technology and Data Variations List:
Der er ikke noget til dette projekt.

Frequency of Occurrence:

- Når der bliver slået med en terning
- Hver spillers tur
- Point der bliver lagt sammen

Miscellaneous:
Der er ikke noget til dette projekt.

2.4 Domain model

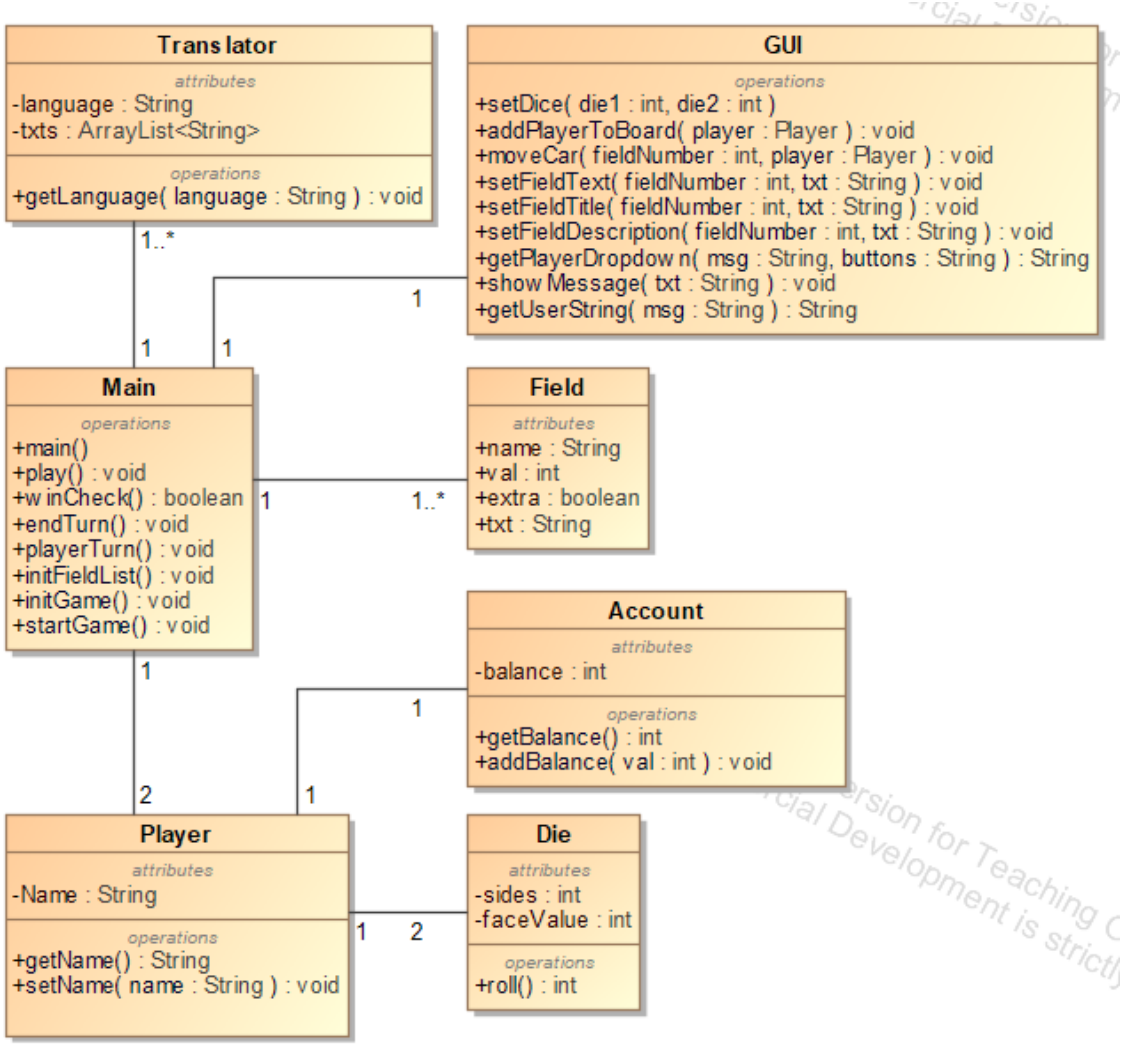


Figur 2: Domain Model

Domænemodellen viser sammenhængen mellem forskellige dele af systemet. I dette tilfælde er der 7 forskellige elementer i modellen: Player, account, die, Main, GUI, fields og translator.
Modellen er ikke særligt dybdegående, men viser alligevel simple relationer mellem elementerne og deres mest tydelige attributter og metoder. Det, der især er brugbart fra modellen, er, at spiller (player) har adgang til konto (account) og terninger (die). Terningernes værdier returneres så til Main, som så tager fat i det felt (Fields), som skal benyttes i turen. Dette giver så main informationer om hvilken historie, der skal printes, hvor mange point, som tabes eller gives, og om der gives ekstra tur. GUI og Translatoren bliver løbende opdateret og brugt.

3 Design

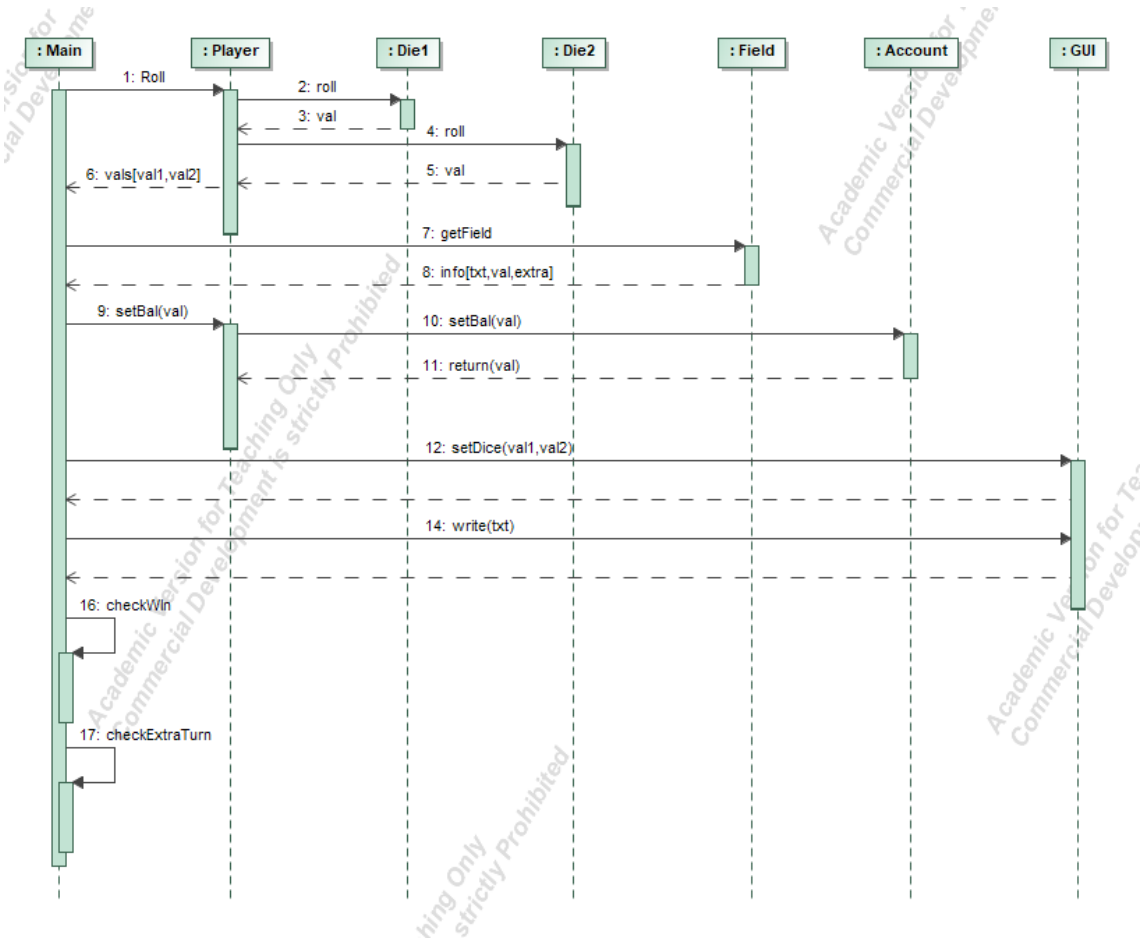
3.1 Klassediagram



Figur 3: Klasse Diagram

Klassediagrammet er en videreudvikling af domænemodellen fra analyseafsnittet. Her er tilføjet relationerne mellem de forskellige klasser. Som ses på diagrammet har én spiller (Player) to terninger (Die). Én spiller (Player) har én konto (account) osv. Der er også lavet en mere grundig beskrivelse af de forskellige klassers attributter og metoder. Diagrammet hjælper til designet af koden på en simpel og overskuelig måde.

3.2 Sekvens Diagram



Figur 4: Sekvens Diagram

Inden der blev kodet noget i dette projekt, blev der lavet et Sekvens Diagram for at holde et overblik over hvordan de forskellige klasser skulle snakke sammen. Og hvilke værdier der skulle sendes frem og derefter returneres. F.eks. kan man se at main klassen ikke snakker direkte sammen med die klassen. Det er blevet lavet således, at main klassen først snakker sammen med player klassen som derefter snakker sammen med die klassen.

Grunden til at der blev lavet et Sekvens Diagram var af nogle forskellige årsager. Det vigtigste var at det gav et bedre overblik over hvordan koden skulle bygges op, og med hvilke klasser der skulle burges til projektet. En anden vigtig grund var for at finde ud af hvordan de forskellige klasser skulle snakke sammen, og i hvor langt til de skulle snakke sammen.

Sekvens Diagrammet gjorde at der var et overblik, over hvilke slags klasser der skulle bruges til programmet, som medførte at der var en struktur i koden da den blev skrevet. Selvom sekvens diagrammet giver at godt overblik er det stadig ikke helt perfekt i forhold til hvordan koden endte med at se ud. Det kan f.eks. ses at der ikke er en "Translator"class i dette diagram. Ideen med translator klassen kom nemlig efter at dette diagram var blevet lavet.

4 Implementering

4.1 Main Class

Main klassen starter og kører vores spil. Det startes ved at lave et GUI-object, lånt fra matadorgui-3.1.6, som er en GUI pakke vi har fået udleveret. Vi bruger vores eget **GUI**-objekt til at give instrukser til denne GUI. Når main metoden bliver kørt, køres der før en masse kode, der laver flere af vores objekter, der skal bruges til at køre programmet. spilleren bliver bedt om at specificere hvilket sprog spillet skal være på, og efter dette bliver **Translator**- og flere **Field**-objekter konstrueret.

Der bliver spurgt om navn på spillerne, og **Player**-objekterne bliver konstrueret og tilføjet til GUI'en. Til sidst køres startGame metoden, der kører resten af spillet.

StartGame kører playerTurn metoden for skiftevis spiller 1 og spiller 2. I denne tur bruges spillerens **Die**-objekter til at slå et slag mellem 2 og 12, hvorefter spillerens **Account**-objekt bliver opdateret med den korrekte værdi fundet i det **Field**-objekt spilleren lander på.

Der bliver kontrolleret om spilleren har vundet, og turen sluttes ved at kontrollere om spilleren skal have en ekstra tur eller ej, og give turen videre til den rigtige spiller.

Når en spiller vinder bliver dette hyldet af spillet og spillet slutter.

4.2 Field Class

I projektet er der også blevet lavet en Field class. Ideen med denne klasse, er at den skal indeholde al information, om felterne spillerne kan lande på. Her er der blevet lavet en konstruktør til at sætte data ind i felterne i spillet. Dermed bliver der for hvert felt indsat, value(værdien af mønter man får eller mister), tekst(beskrivelse af feltets handling) og om man skal have en ekstra tur, når man lander på feltet. Navnet på feltet bliver også gemt, og denne bliver fundet i starten af teksten for feltet.

4.3 Player Class

I Player Classen bliver der oprettet 3 objekter til den tilhørende spiller, Die1, Die2 og guiPlayerObj. Die1 og Die2-objekterne er terningerne. Disse to objekter bruges i roll funktionen. guiPlayerObj kommunikere med GUi'en, og er måden Gui'en kender spilleren på.

For hele tiden at kunne opdatere spillerens pointscore, blev en implementering af metoden for getGuiPlayerObj() oprettet. Her bliver spillerens point opdateret efter hvert slag. For at dette kan lade sig gøre bruges guiPlayerObj.setBalance(this.acc.getBalance()), som opdaterer guiPlayerObjs balance, som GUI'en så kan vise. Derefter returneres guiPlayerObj, da den så kan bruges i Main klassen.

For at spilleren kan benytte terningerne, blev metoden roll() benyttet. Her laves en liste, hvor der derefter rulles med terningerne, som til sidst returnerer resultatet.

4.4 Die Class

Terningen (Die) er meget simpelt sat op. Den indeholder en konstruktør og en enkelt metode, som benytter en instans af Random-klassen fra java.util-pakken til at genererer en tilfældig integer mellem 1 og 6. Værdien gives til int-variablen faceValue. Returnet er faceValue. Grænserne for den tilfældige værdi afhænger af variablen "sides", som definerer antal sider på terningen. Det er på denne måde nemt at skifte til andre typer terninger.

4.5 GUI Class

I programmet er version 3.1.6 af det udleverede Matador GUI bibliotek blevet anvendt. GUI'en viser en matador spilleplade, hvorpå man kan flytte spillernes spillebrikker, i form af biler, rundt. På pladen kan felternes titel ændres, så de passer til de ønskede felter i dette spil.

Til programmet er blevet konstrueret en ny GUI klasse som håndtere brugen af den udleverede matador GUI. Klassen står for håndtering af objekterne på spillebrættet og indeholder funktioner til brugerinteraktion. Funktionerne som klassen indeholder, er som følgende:

1. setDice : void
2. addPlayerToBoard : void
3. moveCar : void
4. setFieldText : void
5. setFieldTitle : void
6. setFieldDescription : void
7. getPlayerDropbown : String
8. getUserString : String
9. showMessage : void

Ved GUI objektet oprettelse vises en standard matador spilleplade med felternes almindelige navne. På pladen kan de 2 terninger, som spillerne spiller med, vises på pladen med funktionen setDice. Funktionen indeholder 2 parameter, hvori man indtaster antallet af øjne. Ved funktionens kald vises terningerne et tilfældigt sted på pladen.

På pladen bliver spillerne også vist i form af 2 biler, som rykkes rundt mellem spillets 11 felter. Før man kan flytte rundt på de 2 spillere på pladen, skal de først tilføjes til spillebrættet. Dette gøres ved brug af addPlayerToBoard funktionen. Den indeholder en parameter hvori spillerens objekt indtastes. Spillerne bliver herefter vist med deres navn, farven på deres bil og antal af point. Efter hver spillers tur og de er landet på et nyt felt, skal spillebrikken flyttes hen på feltet på spillepladen. Dette gøres ved brug af funktionen moveCar, hvor man i parametrene indtaster det nye felt, og spillerens objekt.

Når GUI objektet oprettes har felterne deres normale matadornavne, som skal ændres og tilpasses til dette terning spil. Til dette formål bruges funktionerne setFieldText, setFieldTitle og setFieldDescription, som alle indeholder parametrene, feltet der ønskes ændret og den nye tekst. Med setFieldText skrives feltets tekst, hvor, i dette spil, skrives antallet af guldmønter man modtager eller mister. setFieldTitle ændre feltets titel. Når man trykker på et felt kommer felter frem i midten af spillepladen, hvor man kan læse historien bag feltet. Når historien skal indsættes i felter bruges funktionen setFieldDescription.

Til brugerinteraktion bruges funktionerne getPlayerDropbown, getUserString og showMessage. Når spilleren i starten af spiller skal vælge sprog bruges getPlayerDropdown funktionen, som kommunikere med GUI biblioteket. Funktionen viser en tekst og en dropdown menu, hvor spilleren skal fortage et valg, som bliver returneret. I funktionen parameter skal man først specificere teksten der bliver vist, og bagefter hvilke valgmuligheder menuen indeholder. Når spilleren, efter at have valgt sprog, skal indtaste deres navne, bruges getUserString funktionen. Den viser tekststreng på spillepladen og en tekstboks, som den returnere værdien af. Funktionen indeholde en parameter hvor man indtaster tekststrengen der vises. Den sidste funktion som bruges til brugerinteraktion er showMessage, som viser en teksten, indtastet i dens parameter, på skærmen og en "OK" knap, hvor brugeren kan forsætte spillet.

4.6 Account Class

Som man kan se fra Sekvens Diagrammet så er der også en Account Class som kan kun kan snakke sammen med Player klassen. Grunden til at denne klasse er blevet lavet er for at kunne have en klasse der holder styr på spillerens account balance. Selve klassen er bygget op meget simpelt med en start værdi på 1000 og balancen bliver opdateret med den værdi der kommer fra Field klassen gennem Main. I denne klasse er der også blevet lavet et if statement der siger at hvis account balancen er under 0 skal den sættes til at være lig med 0.

4.7 Translator Class

Til dette projekt skulle der være valget mellem flere forskellige sprog. For at løse dette blev der lavet en Translator class som kunne skifte mellem dansk og engelsk. Selvom klassen hedder "Translator" kunne man også kalde det for en "Dictionary". Dette er fordi der blev lavet 2 forskellige tekst filer med alt den tekst der skulle bruges til projektet. Dette gjorde at der skulle skrives noget kode der kunne læse den tekst der blev skrevet i en txt fil. For at gøre dette blev biblioteket FileReader brugt, som gjorde at der kunne blive læst en linje af gangen i en tekst. Alle disse linjer blev sat ind i en ArrayList. Det vigtige ved disse txt filer var, at linjerne var det samme sted begge steder. f.eks. ver det vigtigt at "Tillykke du har vundet" var på linje 18, både i den danske og i den engelske fil. Dette gjorde at den ArrayList der blev lavet ville have de samme Strings på de samme Index pladser.

Måden, selve klassen var bygget op på, var med en Constructor og en Method. Dette gjorde at Constructoren kaldte Metoden, som resulterede i at i resten af koden kørte,

skulle der kun kaldes på Constructoren. I selve metoden blev der lavet en ArrayList for det sprog der blev valgt i spillet. Med denne måde at lave klassen på er der også gode fremtidsmuligheder, hvis man gerne vil tilføje flere sprog.

5 Test

5.1 JUnit Tests

Da der er blevet brugt JUnit 4 til at teste de adskillige metoder vil kunden kunne gentage de tests der blev foretaget, og derved kan kunden selv afprøve programmet og være sikker på at vores tests ikke var falske.

1. **Første test:**
   Spillerne skal ikke kunne stå med en negativ balance, uanset hvad spillet giver eller tager fra deres balance.

2. **Næste test:**
   Terningerne skal holde sig indenfor givne værdier, f.x. en almindelig terning har en mindste værdi 1 og en maksimal værdi 6.

I den første test blev en simpel JUnit test klasse lavet for klassen Account, som fik navnet AccountTest. Først laves en getBalance() test, som sikrer at getBalance() metoden hos Account giver den rigtige værdi. Det gør den ved at lave et nyt Account objekt, som altid begynder med en balance værdi på 1000. Derefter benytter den getBalance() metoden hos Account og sammenligner den med fastsatte værdi 1000. Det bliver sammenlignet med JUnit metoden assertEquals(). Hvis begge værdier er ens markerer JUnit testen som "passed".

Når getBalance() hos Account er vist at køre korrekt, laves en test metode for addBalance() ved navnet testNegativ(). I test metoden laves et nyt objekt af Account, med default balance værdien. testNegativ() får den aktuelle balance med getBalance(), lægger "1" til, og gemmer den nye værdi i en variabel bal. Derefter bruges addBalance() metoden, hos Account, for at tilføjer bal værdien som negativ. Det giver en negativ balance, men addBalance() forhindrer balance at gå under "0". Hvis efterfølgende getBalance() giver "0"vil testen være "passed".

Den næste test oprettes en ny JUnit test class DieTest for Die class. For at teste om terningerne holder sig indenfor satte værdier, laves en ny test metode testRoll(). Den har tre integer variabler min, max og rollNum. Variablen min er mindst mulige terning værdi og max er den største. "rollNum" fortæller hvor mange gange terningerne skal kastes. Derefter laves en boolean variable "limit" sat til true og et Die objekt "die". Et for loop kører i rollNum-antal gange, hver gang får en integer "num" en ny terning værdi. En if-sætning ser om "num" er mindre end "min"eller større end "max" d.v.s. "num" har en værdi udenfor satte parametre. Hvis det sker sættes "limit" til false og for loop stoppes med "break", ellers kører for loop videre.

Efter for loopet har kørt igennem upåvirket eller ej, testes metoden med at se om "limit" stadigvæk er true med JUnit metoden assertTrue(). Hvis limit er true, vil det sige terning værdierne var indenfor satte parametre, er testen "passed"

6 Projektplanlægning

6.1 Opstart

Ved projektets start, vidste vi fra det forrige projekt, at vi blev nød til at have noget mere struktur over hvordan koden skulle stilles op. For at undgå rod i koden og misforståelser mellem gruppemedlemmer, lavede vi en masse forarbejde før koden overhovedet blev startet på.

Krav Specifikation

Vi lavede hurtigt et dokument, hvor vi kogte opgavebeskrivelsen ned til bullit-points af krav, så vi havde et bedre overblik over hvad opgaven faktisk gik ud på.

Kravene blev grupperet i forhold til hvilke klasser der skulle opfylde hvilke krav. For eksempel var der en krav om af **Player klassen** skulle have et **Account object**, der holdte styr på spillerens balance, så det blev stillet op som værende:

- Player Klasse
  - skal have en Account, der holder styr på spillerens point
  - ...
- Account Klasse
  - Attribut: int Balance
  - Metode: getBalance()
    - returnerer balance
  - Metode: addBalance(int value)
    - tilføjer værdien til balance
  - ...
- ...

Enkelte dele af opgaven var vi i tvivl om, men dette kunne vi få uddybet af en hjælpelære.

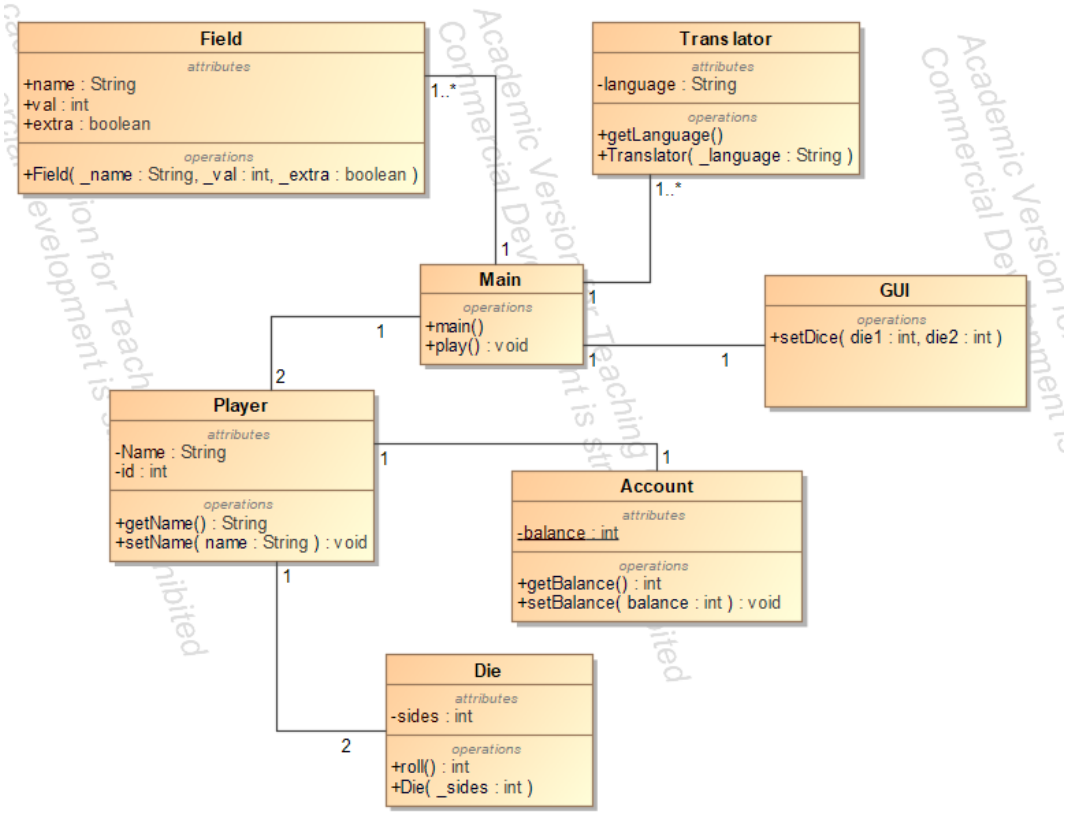
VCS

Vi brugte Git og Github.com til at holde styr på vores udvikling af programmet samt at distribuere vores arbejde mellem os. Vores arbejdshistorik er også blevet gemt på Github og kan findes på https://github.com/Software2019Hold19/19_del2.

Klassediagram

For at få en generel ide om hvordan koden skulle se ud, lavede vi som noget af det første et hurtigt klassediagram. Da vi allerede havde en god ide om hvilke dele der skulle være i projektet, var en skitse af et klassediagram en nem måde at give koden et skelet at bygge videre på.

Vi gennemgik hvad, der sker i spillet, trin for trin og genererede derved en slags pseudo-klassediagram, hvor klasserne og deres mest væsentlige dele blev skrevet op. Her fik vi allerede fastlagt relationerne mellem klasserne samt de største metoder i klasserne som for eksempel *Main.playerTurn()* og *Player.roll()*. Vi kom løbende tilbage til klassediagrammet for at opdatere denne, når der blev brug for flere attributter eller metoder for en klasse.



Figur 5: Tidlig version af Klasse diagrammet

6.2 Project Board

Vi har i Github lavet et projectboard, der har indeholdt vores forskellige arbejdsopgaver samt deres beskrivelse og hvem der har til opgave at løse dem. Vi har med denne nemt kunne holde styr på hvilke opgaver der er i gang, hvilke opgaver de enkelte gruppemedlemmer har til opgave, samt hvilke opgaver der er implementeret.
Dette har gjort det nemt for gruppemedlemmerne at kunne følge med i de andres arbejde, og kunne spørge til råds inden for de enkelte dele af koden, da de nemt kan se, hvem der har skrevet den og dermed har styr på den.

6.3 Arbejdsfordeling

Vi vidste fra sidste projekt, at vi skulle have et skema over den brugte tid på projektet. Dette havde resulteret i en masse gættearbejde med, at hvert gruppemedlem havde arbejdet på projektet, da denne ikke blev oprettet i starten. I dette projekt har vi løbende holdt styr på, hvor mange timer hvert gruppemedlem cirka har brugt på delene af projektet, for at få et bedre resultat samt mindre tidsspild på dette.
Her er skemaet over vores arbejdsfordeling i projektet:

Opgave	Dánjal	Frederik	Jacob	Oliver	Thomas	Yassine
Setup	2.5	3.5	2.5	2.5	2.5	2.5
Kode	7	7.5	6	7	4.5	3.5
Rapport	8	7	9	9	11	7.5
Total	**17.5**	**18**	**17.5**	**18.5**	**18**	**13.5**

7 Konklusion

Der kan konkluderes, at der til dette projekt er blevet lavet et succesfuldt terningespil mellem 2 spillere. Alle krav fra den fiktive kunde er opfyldt. Gennem dette projekt er det blevet arbejdet meget med objekt orienteret programmering. Dette har gjort at koden er blevet meget overskuelig, og det har gjort testen af projektet meget nemmere.

Programmet til dette projekt fungerer som det skal. Der er blevet fundet nogle BUGS, men de er blevet fikset uden yderligere problemer. Der hvor dette projekt er blevet udført rigtig godt var i planlægningen, der blev lavet inden at nogen begyndte at kode noget. Dette gjorde at implementeringen af koden startede sent, men det minimerede til gengæld også de fejl, der kunne opstå. Ved at analysere programmets beskrivelse, og derefter designe det med UML-modeller, har der også været et skarpere fælles overblik i gruppen. Implementationen har derfor ikke været lige så rodet og besværlig, som ellers ville have været tilfældet.

8 Bilag

8.1 Bilag 1: Forklaring af Translator

Language ArrayListen indeholder alt vores tekst til spillet. Indexet i listen refererer til bestemte strings, der skal bruges i bestemte situationer. ArrayListen er .splittet() med "??".

Hvis der er en Variable, skal teksten skrives med metoden:
String.format(String txt, variable(s))

Index	Description	Variables
0-10	Felters beskrivelse (index 0 = felt 2, index 1 = felt 3, osv)	None
11	Velkommen til spillet	None
12	Spiller 1 Navn	None
13	Spiller 2 Navn	None
14	Velkommen spillere	player1.name, player2.name
15	Spiller tur	player.name
16	Du har slået	die1.facevalue, die2.facevalue, dieSum
17	Spiller vinder	player.name
18	Spiller hedder det samme	None