



02312, 62531, 62532

INDLEDENDE PROGRAMMERING, UDVIKLINGSMETODER TIL IT-SYSTEMER OG
VERSIONSSTYRING OG TESTMETODER

CDIO 2

Krzysztof Kisiel
s192884



Oliver Olsen
s205443

Johannes Jensen
s205413



Gustav Fauser
s205446

Jonathan Hansen
s205415



Alaa Hadood
s205477

Timeplan

Tid i timer	Krav og analyse	Design	Dokumentering af Implementering	Test	Programmering
Navn					
Gustav Fauser	3	2	1		
Oliver Olsen	2	2	3		
Krzysztof Kisiel	3	4	1	1	11
Johannes Jensen	2	2	1	3	
Jonathan Høj	2	2			
Alaa Hadood	2	2			3

Summary

We have to create a new dice game, which contains at least 2 players. Each player starts with 1000 virtual coins. Each space on the board has its own advantages. Some spaces give the player more points/coins, removes coins or even gives the player an extra turn. The game is won when a player reaches 3000 coins.

Indholdsfortegnelse

Timeplan	1
Summary	2
Introduktion	5
Krav	5
Krav til spillet	5
Krav til kodningen	5
Krav til arbejdsprocessen	6
Analyse	7
Use cases	7
Domænemodel	11
Systemsekvensdiagram	12
Design	13
GRASP	13
Klasser	13
Game	13
Logic	13
Field	13
FieldList	13
Board	14
GameGUI	14
Dice & Die	14
Account	14
Player	14
Player List	14
Translator	14
Model - Klasser	15
Account	15
Player	17
PlayerList	18
Board	22
FieldList	22
Controller - Klasser	23
Game	23
Logic	25
Grafisk brugerflade - Klasser	27
Game GUI	27
Translator	28
Konfiguration	29
Test	30

Dice	33
Nemt at oversætte	34
Projektplanlægning	35
Konklusion	36
Kilder	37

Introduktion

Vores opgave er at skabe et spil i de tre kurser Indledende programmering, Udviklingsmetoder til IT-systemer og Versionsstyring & testmetoder. Udvikleren IOOuterActive (os) skal opdatere vores første, simple spil og gøre det mere avanceret ved at tilføje felter og mulighed for flere spillere.

Krav

Kunden som har bestilt spillet har stillet en række krav, som vi skal efterleve i vores spil. Nogle af kravene er fra kunden, mens andre er defineret ud fra kundens beskrivelser og ønsker.

Krav til spillet

1. Systemet skal starte spillet med 1000 point i hver af spillernes pengebeholdning
2. Systemet skal udskrive en tekst for det felt spilleren lander på
3. Systemet skal ændre spillerens pengebeholdning i positiv eller negativ retning i overensstemmelse med det felt spilleren er landet på
4. Systemet skal afslutte spillet når en af spillerne har opnået 3000 point
5. Spillerens pengebeholdning må ikke gå i minus
6. Felterne på spillepladen skal være nummereret fra 2-12

Krav til kodningen

7. Spillet skal kunne spilles på maskinerne i DTU's databarer
8. Spillet skal let kunne oversættes til andre sprog
9. Systemet skal kunne eksekvere spillet uden forsinkelser
10. Det skal være let at skifte til andre terninger
11. Spilleren og dens tilhørende pengebeholdning skal programmeres så den kan implementeres i andre spil fremover
12. Systemet skal kunne oprette en feltliste hvor navn, farve og andre værdier specificeres
13. Spillebrættet skal være fyldt ud med felter fra feltlisten (Lave Text klasse som kan tildele text til alle scenarier)
14. Oprette en spillerliste som kontrollerer spillerens tur
15. Spilleren skal kunne bevæge sig rundt på spillebrættet uden stop
16. Systemet skal være i stand til at give spilleren en ekstra tur
17. Der skal være mulighed for op til 6 spillere at spille samtidigt
18. Systemet skal kunne håndtere spillernes pengebeholdning
19. Systemet skal opdatere spillernes pengebeholdning i overensstemmelse med det felt spilleren står på
20. Spillernes balance kan ikke gå i minus

Krav til arbejdsprocessen

21. Alt arbejde på spillet skal versioneres så der kan følges med i hvordan udviklingen er foregået
22. De enkelte gruppemedlemmers/programmøres bidrag skal fremgå af versioneringen
23. Versioneringen skal forekomme i formatet .git
24. Projektet som helhed skal leveres i form af en zip-fil

Analyse

Spillet har en række Use cases som vi har defineret og efterfølgende analyseret. Use case er defineret ud fra kundens beskrivelse og ønsker omkring spillet.

Use cases

Ud fra beskrivelsen er der blevet defineret 4 forskellige Use cases. De 4 Use cases er efterfølgende beskrevet enkeltvis.

ID	Use case
1	Spil spil
2	Spil runde
3	Kast terning
4	Slut spillet

Use case	Spil spil
ID	1
Brief description	Spilleren påbegynder spillet, bevæger sig rundt på spillepladen, når 3000 point og afslutter spillet
Primary actors	Player
Stakeholders and interests	None
Preconditions	Spilleren har indtastet sit navn og valgt farve til sin bil
Success Guarantee	Succes er garanteret
Main flow	<ol style="list-style-type: none"> 1. Use casen starter når spilleren åbner spillet 2. Spilleren indtaster det ønskede antal spillere 3. Spilleren indtaster navnene til det ønskede antal spillere 4. Terningerne kastes 5. Summen af terningerne vises 6. Spilleren rykker det samme antal felter som summen af terningerne 7. Systemet udskriver teksten for det felt som spilleren er kommet til 8. Hvis spilleren er kommet til et felt med et negativt beløb <ol style="list-style-type: none"> a. Det negative beløb på feltet fratrækkes spillerens pengebeholdning 9. Hvis spilleren er kommet til et felt med et positivt beløb <ol style="list-style-type: none"> a. Det positive beløb på feltet tillægges spillerens pengebeholdning 10. Hvis spilleren er kommet til et felt hvor der tildeles en ekstra tur <ol style="list-style-type: none"> a. Gentag trin 2 11. Hvis spilleren opnår 3000 point, kåres spilleren til vinder og spillet afsluttes
Extensions (or Alternative Flows)	
Postconditions	Spillet er slut
Alternative flows	Ingen

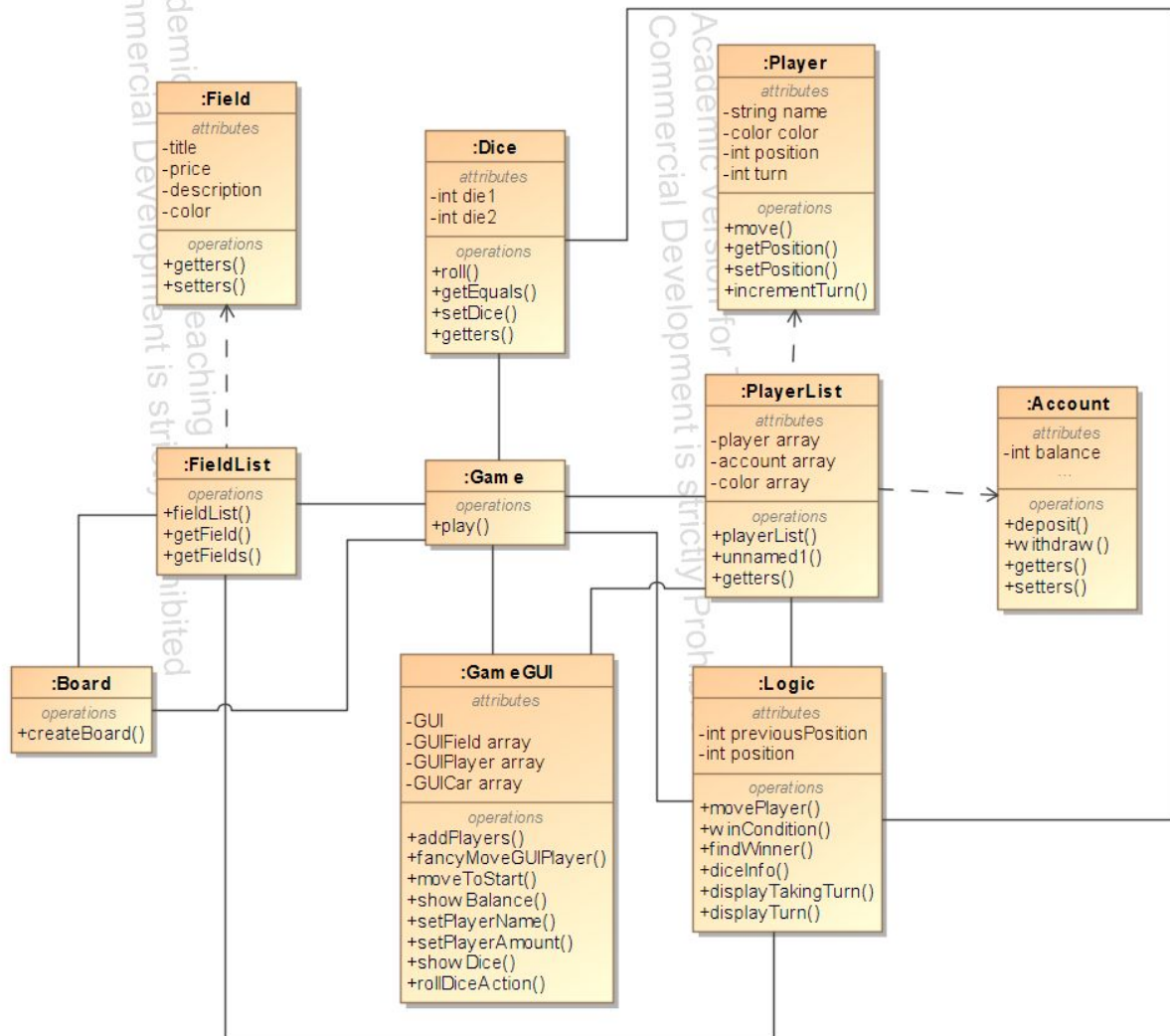
Use case	Spil runde
ID	2
Brief description	Der er den pågældende spillers tur, til at slå og flytte.
Primary actors	Player
Stakeholders and interests	None
Preconditions	Det er spillerens tur
Success Guarantee	Spilleren har flyttet sig på brættet
Main flow	<ol style="list-style-type: none"> 1. Spilleren kaster terningerne 2. Bilen(spillerens grafiske repræsentation) bliver flyttet på brættet (GUI) 3. Terningerne vises 4. Turen går videre til den næste spiller
Extensions (or Alternative Flows)	
Postconditions	Turen går videre til den næste spillere som gentager Use casen
Alternative flows	Ingen

Use case	Kast terning
ID	3
Brief description	En spiller skal kaste med terningerne
Primary actors	Player
Stakeholders and interests	None
Preconditions	Det er spillerens tur
Success Guarantee	Spilleren har flyttet sig på brættet, efter summen af terningerne
Main flow	<ol style="list-style-type: none"> 1. Spilleren kaster terningerne 2. Bilen(spillerens grafiske repræsentation) bliver flyttet på brættet (GUI) 3. Terningerne vises 4. Turen går videre til den næste spiller
Extensions (or Alternative Flows)	
Postconditions	Turen slutter
Alternative flows	Ingen

Use case	Slut spillet
ID	4
Brief description	Spillet er slut
Primary actors	Player
Stakeholders and interests	None
Preconditions	En spiller har opnået 3000 point
Success Guarantee	Spillet er slut
Main flow	<ol style="list-style-type: none"> 1. Spilleren har opnået 3000 point 2. En besked viser hvem der har vundet
Extensions (or Alternative Flows)	
Postconditions	Spillet slutter
Alternative flows	Ingen

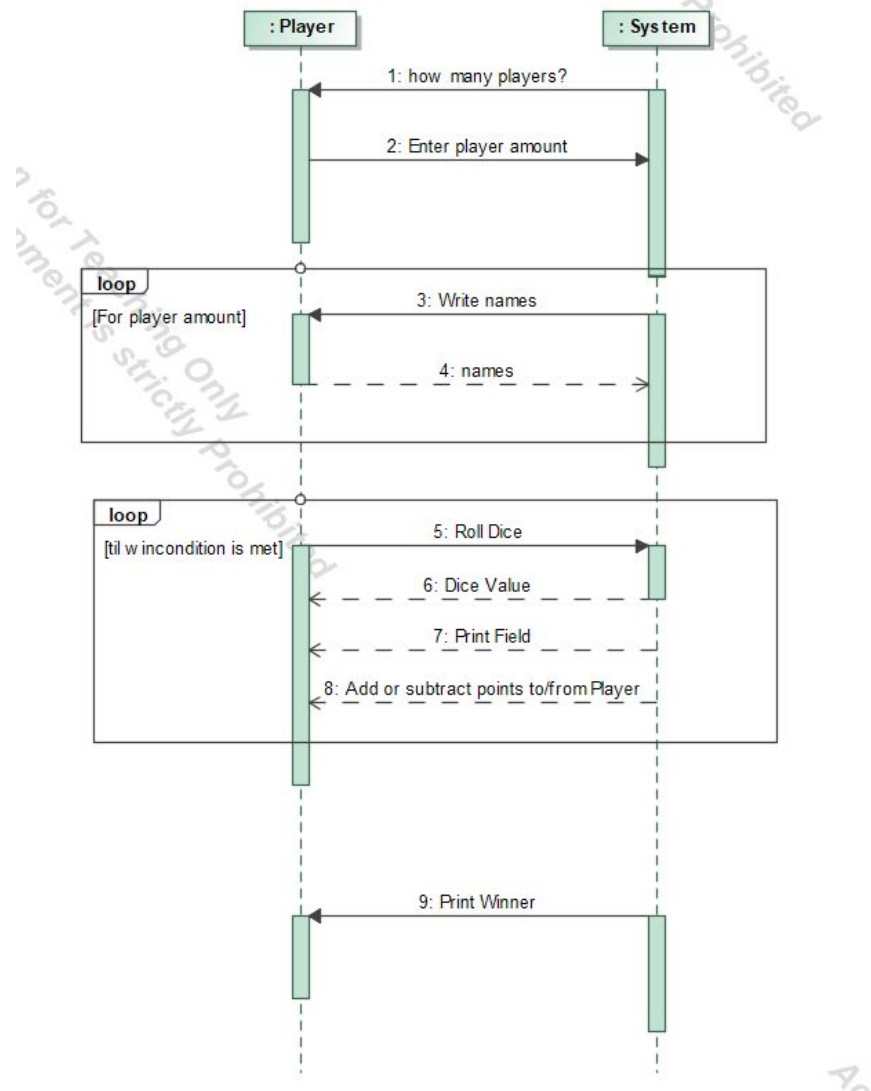
Domænemodel

Spillet er bygget op af en række klasser. De forskellige klasser arbejder sammen indbyrdes, for at skabe et komplet spil. Domænemodellen nedenfor beskriver sammenhængende og relationerne mellem de forskellige klasser.



Systemsekvensdiagram

Sekvensdiagrammet viser hvordan hoveddelen af spillet fungerer. Det viser interaktionen mellem Player klassen og Logic klassen (System). Sekvensdiagrammet kan man dele op i 3 overordnede dele: start, spil og slut. Starten er hvor de enkelte spilleren vælger antallet af spillere og indtaster navne. Spil er hvor selve spillet bliver spillet. Spil delen kører over og over igen indtil en spiller har opnået 3000 point. Slutningen forekommer umiddelbart efter at en spiller har opnået 3000 point. Her får spillerne at vide hvem der har vundet.



Design

I dette afsnit vil vi definere de klasser, som vi vil få brug for ud fra kravene, samt vores analyse, specielt fra domænemodellen.

GRASP

Der er anvendt en række GRASP-mønstre til at allokere ansvar til de enkelte klasser. Et mønster der er blevet anvendt er Lav kobling (Low Coupling), spillet er sammensat af en række af klasser som har et veldefineret ansvarsområde f.eks. Account og Player.

Derudover er der blevet anvendt Creator mønster, dvs der er en klasse, der står for brugerinput og den grafiske brugerflade. Klassen i spillet er GameGUI.

Programmet er delt op i 2 overordnede dele: Logik og den grafiske brugerflade. På den måde er det nemmere at teste logikken i spillet, uden at skulle teste den grafiske brugerflade.

Klasser

Vi har vurderet, at følgende klasser skal oprettes, for at lave en overskuelig kode til spillet:

Game

I "Game" klassen, bliver alt vores kode taget i brug. Den "kalder" på de andre klasser, som gør brug af forskellige metoder. Det er altså i denne klasse selve spillet bliver kørt. - Det er her logikken foregår.

Logic

"logic" klassen er ansvarlig for spillereglerne, så som at spillet afsluttes når en af spillerne når 3000 point og at spillerne på skift får deres tur. "Logic" klassen er også ansvarlig for at bestemme hvilke felter der skal trække penge fra spillerens konto og hvilke der skal lægge penge til spillerens konto.

Field

"Field" klassen indeholder informationerne vedrørende hvert felt. Når man lander på et bestemt felt, vil feltets tilhørende beskrivelse, pengeværdi, navn og nummer blive vist til spilleren. Det er i denne klasse at alle disse attributter bliver bestemt. Når en spiller lander på et givent felt, vil informationen vedrørende dette felt altså blive overført fra felt klassen.

FieldList

I "FieldList" klassen har vi et array af de felter, som i Field klassen blev oprettet og

defineret.

Board

Boardklassen skal oprette spillebrættet med informationer fra GUI og fieldlist.

GameGUI

GameGUI klassen skal have ansvar for at opdatere den grafiske brugerflade til spillet. Klassen indeholder derfor metoder der opdatere GUI, med information fra selve logikken i spillet. GameGUI klassen skal ikke holde styr på regler, point osv. Men blot have ansvar for den grafiske repræsentation af spillet, for spillerne. Gui klassen holder styr på felter, spiller og biler der bliver brugt til at bevæge spilleren. GameGUI bruger metoder fra "Matador GUI" version 3.1.7.(Budtz ,C. et al. 2019)

Dice & Die

I "Dice" klassen, slår man med terningerne i spillet. Terningernes værdi skal være et tilfældigt genereret tal, som kan blive slået med terningerne. I kravene står der at man let skal kunne skifte til andre terninger. Terningens sider skal kunne bestemmes, så man kan slå med to terninger med seks sider eller f.eks. 12 sider. Det sker i "Die" klassen. Terningernes sum, samt individuelle værdi skal kunne bruges af de andre klasser, for at vurdere hvor spilleren skal lande, samt om terningerne er ens.

Account

I "Account" klassen, kontrolleres spillernes pengebeholdninger. Pengebeholdningen skal ændres hver gang spillerne lander på et nyt felt og der enten skal trækkes penge fra, eller tillægges penge til pengebeholdningen. De andre klasser skal kunne tilgå spillernes pengebeholdninger. Bla. er "Game" klassen afhængig af pengebeholdningen, da en pengebeholdning på 3000 er kravet for at vinde spillet.

Player

I "Player" klassen skal der holdes styr på spillernes informationer. Deres navn, farve, position og hvilken tur spillerne er i gang med.

Player List

I "PlayerList" klassen, skal der holdes styr på antallet af spillere, med deres tilhørende navn, farve account, tur og position. Denne klasse gør brug af de informationer, som "Player" klassen holder styr på.

Translator

Translator klassen gør det nemt at oversætte spillet til forskellige sprog. Klassen kan tage imod en sprog forkortelse f.eks. da eller en, og derefter indlæse den korrekte sprogfil og oversætte programmet.

Implementering

I dette afsnit vil vi dokumentere hvordan vi har implementeret koden. Koden består af følgende klasser: Account, Player, PlayerList, Board, Dice, Die, Field, FieldList, Game, GameGUI og Logic.

Model - Klasser

Account

Account klassen holder styr på spillerens virtuelle penge. Vi har brug for en integer, som kan holde styr på spillerens penge:

```
public class Account {  
  
    private int balance;
```

Vi har lavet en konstruktør, som er følgende:

```
public Account(int balance) {  
    this.balance = balance;  
}
```

Denne konstruktør sætter klassens "balance" til det samme som konstruktørens balance. Nu når konstruktøren er på plads, så kan vi lave nogle metoder. Vi har brug for en metode, som kan hæve/trække penge fra spillerens account/konto. Dog må kontoen ikke gå i minus, så hvis det bliver trukket et beløb der er større end hvad spilleren har tilgængeligt, så skal spillerens antal penge sættes til 0:

```
public void withdraw(int amount) {  
    setBalance(getBalance()-amount);  
    if (getBalance()<0) {  
        setBalance(0);  
    }  
}
```

Vi har også brug for en metode, som kan deponere penge til spillerens konto:

```
public void deposit(int amount) {  
    setBalance(getBalance()+amount);  
    if (getBalance()<0) {  
        setBalance(0);  
    }  
}
```

Her er der også lavet et simpelt if-statement, som tjekker om spillerens konto er i minus, så den kan sættes til 0. I metoden "deposit" lægges "amount" (mængden af penge) til spillerens konto (balance).

Når spillet starter, så skal hver spiller starte med 1000 virtuelle penge. Det gøres med med følgende metode:

```
public int getStartingBalance(){
    setBalance(1000);
    return balance; }
```

Den sætter spillerens "balance" til 1000, når spillet starter. Integer'en "getStartingBalance" er importeret fra klassen PlayerList, og vil blive dokumenteret i dens klasse senere i rapporten. Spillerens mængde penge skal også kunne ses. Det gøre med følgende metode:

```
public int getBalance() {
    return balance;
}
```

Her kan de andre klasser "kalde" på getBalance, hvor metoden, så vil returnere balance, som er spillerens mængde penge.

De andre metoder i klassen, skal kunne bestemme/ændre mængden af penge. Det gør de ved at bruge "setBalance". Denne metode skal være den aktuelle sum af penge. Derved kan vi definere setBalance følgende:

```
public void setBalance(int balance) {
    this.balance = Math.max(balance, 0);
}
```

Player

Player klassen holder styr på spillernes navn, farve, position og hvor mange ture spilleren spilleren har været igennem.

Til det har vi da oprettet følgende:

```
public class Player {  
  
    private String name;  
    private Color color;  
    private int currentPosition;  
    private int turn;
```

De informationer skal da kunne kobles på en spiller. Det har vi gjort følgende:

```
public Player(String name, Color color){  
    this.name = name;  
    this.color = color;  
    int turn = 0;  
}
```

Her bliver spillerens information opbevaret. Spilleren får "tildelt" navn, farve og turen bliver sat til 0, da spillet lige er startet.

For at holde styr på spillerens position på pladen, så har lavet følgende metode:

```
public void move(int position, FieldList fl){  
    int fieldLength = fl.getSize();  
    this.currentPosition = (currentPosition + position) % fieldLength;  
}
```

Her er "fieldLength" og "fl.getSize()" importeret fra klassen FieldList. De bruges til at holde styr på spillepladens længde. De vil blive gennemgået i FieldList længere nede. Den sidste linje holder styr på hvor på pladen spilleren lander, eller om spiller kan komme hele vejen rundt. Her er "position" det antal felter, som spilleren skal rykke frem, hvor "currentPosition" er der hvor spilleren står lige nu.

For at Player klassen ved hvor spilleren står, hvilken tur det er, hvilken farve spillerne har valgt og spillernes navne, så lavet følgende metoder:

```
public int getCurrentPosition() { return currentPosition; }  
public void setCurrentPosition(int position){ this.currentPosition = position; }  
public void incrementTurn(){ turn++; }  
public int getTurn(){ return turn;}  
public Color getColor() { return color; }  
public void setColor(Color color) { this.color = color; }  
public String getName() { return name; }  
public void setName(String name) { this.name = name; }
```

Den første metode (getCurrentPosition) returnere spillerens aktuelle position. Metoden "setCurrentPosiotion" ændrer spillerens aktuelle posiotion, så spilleren rykker frem, når der

bliver slået med terningerne. Metoden "incrementTurn" opdaterer hvilken tur det er, så når den første tur er overstået, så skrifter turn fra at være 0 til at være en. Metoden ""getTurn" returnerer hvilken tur spilleren er i gang med. Metoden "getName" returnerer spillerens navn. Metoden "setName" sætter spillerens navn til at begynde med.

PlayerList

PlayerList holder styr på spillerens navn med spillerens tilhørende farve, samt konto. Det har vi gjort ved hjælp af arrays.

```
Player[] players;  
Account[] accounts;
```

Her oprettes to arrays, som holder styr på antallet spillere med deres navne, samt deres tilhørende konto.

Spillerne skal kunne vælge en farve, samt et navn:

```
private String[] names = {"Peter", "Marcus", "Oliver", "Phill"};  
private Color[] colors = {Color.CYAN, Color.GREEN, Color.WHITE, Color.BLUE, Color.orange, Color.RED, Color.WHITE };
```

Det er også gjort ved hjælp af arrays. Her er fire forskellige navne, som man kan vælge mellem og syv forskellige farver.

```
public PlayerList(int playerAmount){  
    players = new Player[playerAmount];  
    accounts = new Account[playerAmount];  
    for (int i = 0; i < playerAmount; i++) {  
        players[i] = new Player( name: null,getColor(i));  
        accounts[i] = new Account( balance: 0);  
    }  
}
```

På billedet ovenfor, der oprettes antallet af spiller og accounts. Her er "playerAmount" antallet af spiller og accounts, som der skal oprettes. Derefter er der en "for" kommando, som får fat i den farve som spillerne har valgt, og de får en account, som lige nu er tom, men som bliver sat til 1000, når spillet starter.

For at få fat i spillernes farver, account og navn, så sker følgende:

```
public Color getColor(int id) { return colors[id];}  
  
public Player[] getPlayersList() { return players; }  
public Player getPlayerList(int id) { return players[id]; }  
  
public Account[] getAccounts(){ return accounts; }  
public Account getAccount(int id){ return accounts[id];}
```

Dice klassen:

```
public class Dice {  
  
    private int die1;  
    private int die2;  
  
    public Dice(int die1, int die2) {  
        this.die1 = die1;  
        this.die2 = die2;  
        roll();  
    }  
}
```

Dice klassen begynder med, at vi opretter to die og kalder en roll metode på dem.

```
public int roll(){  
    Random r = new Random();  
    die1 = r.nextInt(6)+1;  
    die2 = r.nextInt(6)+1;  
    return getSum();  
}
```

Roll metoden bruger Random metoden, så når terningerne 'kastes' vil de hver have en værdi mellem 1 og 6. Der vil ligeledes blive returneret en sum af de to terningers værdier.

```
boolean getEquals(){ return die1 == die2; }  
public void setDice (int dice1, int dice2) { this.die1 = dice1; this.die2 = dice2;}  
  
//Getters and setters  
public int getDie1() { return die1; }  
public int getDie2() { return die2; }  
public int getSum() { return die1 + die2;}  
public String toString() { return Integer.toString(roll()); }
```

(Bliver ikke brugt) getEquals metoden, viser, når vores terninger har samme værdi. Herefter har vi først en setDice metode, som bruges til at tilvige terningerne en bestemt værdi, og efterfølgende en getDice metode, som kan returnere udfaldet af de to terninger hver især eller deres sum.

Die klassen:

```
public class Die {  
  
    private int faces;  
    private int faceValue;  
  
    public Die(int faces, int faceValue){  
        this.faces = faces;  
        this.faceValue = faceValue;  
    }  
}
```

i Die klassen har vi først lavet en terning, som indeholder et antal sider og antallet af øjne til den tilhørende side.

```
public int roll(){
    Random r = new Random();
    setFaceValue(r.nextInt((faces)+1));
    return faceValue;
}
```

Vi har også her en metode, roll, som igen bruger Random metoden til at bestemme antallet af øjne, terningen viser. Alt efter antallet af sider på terningen, vil Random metoden 'kaste' terningen, og terningens værdi bliver returneret.

```
public int getFaces() { return faces; }

public void setFaces(int faces) { this.faces = faces; }

public int getFaceValue() { return faceValue; }

public void setFaceValue(int faceValue) { this.faceValue = faceValue; }
```

set- og getFaces metoderne bruges til henholdsvis at vælge et antal af sider, som en terning skal have, og til at vise hvor mange sider, en given terning har.

set- og getFaceValue har samme virkning, men i stedet for antallet af sider, bruges de til at vælge og vise hvilken side af terningen, der i øjeblikket vises.

Field

"Field" klassen skal holde styr på felternes attributter, altså ID, titel, pris, beskrivelse og farve. Vi opretter derfor en integer til id og pris, en string til titel og beskrivelse.

```
public class Field {

    private int id;
    private String title;
    private int price;
    private String description;
    private Color color;
}
```

Vi laver følgende konstruktør:

```
public Field(String title, int price, String description, Color color) {
    this.title = title;
    this.price = price;
    this.description = description;
    this.color = color;
}
```

Vi laver get og set metoder til vores forskellige attributter, så andre klasser kan kalde på disse værdier eller sætte værdierne til en ønsket værdi.

```
public String getTitle() { return title; }
public void setTitle(String title) { this.title = title; }
public int getPrice() { return price; }
public void setPrice(int price) { this.price = price; }
public String getDescription() { return description; }
public void setDescription(String description) { this.description = description; }
public Color getColor() { return color; }
public void setColor(Color color) { this.color = color; }
```

Afslutningsvist opretter vi en metode der returnerer alle informationer om et field objekt

```
public String toString(){
    return ("Title:" + getTitle() + "\n" + "Price:" + getPrice() + "\n" + "Description " + getDescription() + "\n");
}
```


Board

“Board” klassen opretter et board ud fra GUI-klassen og Fieldlisten

```
public GUI_Street[] createBoard(FieldList fl) {  
  
    GUI_Street[] gui_fields = new GUI_Street[fl.getSize()];  
  
    for (int i = 0; i < fl.getSize(); i++) {  
        Field field = fl.getField(i);  
        gui_fields[i] = new GUI_Street(field.getTitle(),  
            String.valueOf(field.getPrice()),  
            field.getTitle(),  
            rent: "",  
            field.getColor(),  
            Color.BLACK);  
    }  
    return gui_fields;  
}
```

FieldList

“Fieldlist” er en masse arrays, der definerer ID, titel, pris og farve for hvert felt på brættet

```
fields[0] = new Field( id: 1, title: "This is the first field", price: 200, description: "1", Color.green);  
fields[1] = new Field( id: 2, title: "Tower", price: 250, description: "2", Color.green);  
fields[2] = new Field( id: 3, title: "Crater", price: 100, description: "3", Color.red);  
fields[3] = new Field( id: 4, title: "Palace gates", price: 100, description: "4", Color.green);  
fields[4] = new Field( id: 5, title: "Cold Desert", price: 20, description: "5", Color.red);  
fields[5] = new Field( id: 6, title: "Walled City", price: 180, description: "6", Color.green);  
fields[6] = new Field( id: 7, title: "Monastery", price: 0, description: "7", Color.green);  
fields[7] = new Field( id: 8, title: "Black cave", price: 70, description: "8", Color.red);  
fields[8] = new Field( id: 9, title: "Huts in the mountain", price: 60, description: "9", Color.green);  
fields[9] = new Field( id: 10, title: "The werewall", price: 80, description: "10", Color.red);  
fields[10] = new Field( id: 11, title: "The pit", price: 50, description: "11", Color.red);  
fields[11] = new Field( id: 12, title: "Goldmine", price: 650, description: "12", Color.green);
```

Her har vi en metode der printer felternes informationer til systemet

```
// Prints out all the fields to the console  
public void getAllFields() {  
    for (int i = 0; i < getSize(); i++) {  
        System.out.println(fields[i]);  
    }  
}
```

Controller - Klasser

Game

```
public class Game {  
  
    public void play() throws InterruptedException {  
  
        Board board = new Board();  
        FieldList fl = new FieldList(12);  
  
        Logic logic = new Logic();  
  
        GUI gui = new GUI(board.createBoard(fl), Color.WHITE);  
        GameGUI gameGui = new GameGUI(gui);  
  
        int playerAmount = gameGui.setPlayerAmount();  
        PlayerList pl = new PlayerList(playerAmount);  
  
        Dice dice = new Dice(0,0);  
  
        for (int i = 0; i < playerAmount; i++) {  
            String name = gameGui.setPlayerName();  
            pl.getPlayerList(i).setName(name);  
            pl.getAccount(i).setBalance(pl.getAccount(i).getStartingBalance());  
        }  
    }  
}
```

Game klassen starter med en play metode. Dette sker ved spillets start, hvor et nyt board med felterne bliver sat, og logikken, GUI'en samt de andre klasser bliver implementeret. Spillerne bliver bedt om at indtaste hvor mange de er og derefter vælge et navn hver, hvorefter de får tildelt 1000 point ved start. Når spillerne har indtastet denne information, vil spillet begynde.

```
gameGui.addPlayers(pl);  
  
int playerTurn = 0;  
int preTurn = 0;  
  
for (int i = 0; i < playerAmount; i++) {  
    while(!logic.winCondition(pl)) {  
        preTurn = playerTurn;  
  
        logic.displayTakingTurn(pl, playerTurn);  
        gameGui.rollDiceAction(pl, playerTurn);  
  
        logic.movePlayer(pl, fl, dice, playerTurn);  
  
        logic.diceInfo(pl, dice, playerTurn);  
  
        gameGui.showDice(dice.getDie1(), dice.getDie2());  
        gameGui.fancyMoveGuiPlayer(logic.prePos, playerTurn, dice);  
        gameGui.showBalance(pl, playerTurn);  
  
        pl.getPlayerList(playerTurn).incrementTurn();  
        logic.displayTurn(pl, playerTurn);  
        playerTurn = (playerTurn + 1)%playerAmount;  
    }  
}
```

Efter spillets begyndelse, tilføjes spillerne i GUI'en. Spillet kører i et for-loop, hvor der bliver kaldt en række metoder fra logic klassen og vores GUI gennem hele spillets forløb.

Der vises hvis tur, det er, og terningen kastes. Derefter bliver movePlayer metoden kaldt, og spilleren vil rykke til næste felt. DiceInfo fra logic klassen kaldes, og informationen omkring spillerens placering og kontobeholdning opdateres.

Imens dette sker i logikken, vil GUI'en vise terningerne på skærmen. Når terningerne er kastet, vil terningernes nye værdier vises på skærmen, og GUI'en sørger for, at spillerens bil bevæger sig ét felt ad gangen med fanceMoveGuiPlayer metoden. Den vil også vise den opdaterede kontobeholdning på skærmen for spilleren.

Fra playerlisten har systemet styr på hvem, der lige har slået, og turen vil skifte til den næste spiller på listen. Herefter vil den samme process finde sted for denne spiller. Dette loop fortsætter indtil en spiller når 3000 point.

```
}  
logic.findWinner(pl, preTurn);  
gameGui.displayWinner(pl, preTurn);  
gui.close();
```

Når en spiller når 3000 point vil findWinner metoden blive kaldt fra logic, som finder ud af hvem, der har vundet spillet, og vores GUI vil derefter annoncere det på skærmen. Spillet er hermed slut.

Logic

```
public class Logic {  
  
    public int prePos;  
    public int pos;  
  
    public Logic() {  
    }  
}
```

Vi har en metode der bevæger spilleren ud fra spillerens forrige position, registrerer hvilket felt spilleren er landet på og derefter enten trækker penge eller tillægger penge til/fra spillerens konto.

```
public void movePlayer(PlayerList pl, FieldList fl, Dice dice, int playerTurn) {  
    Player player = pl.getPlayerList(playerTurn);  
    Account account = pl.getAccount(playerTurn);  
  
    prePos = player.getCurrentPosition();  
    player.move(dice.roll(), fl);  
    pos = player.getCurrentPosition();  
  
    switch (pos) {  
        case 0: case 1: case 3: case 5: case 8: case 11:  
            account.deposit(fl.getField(pos).getPrice());  
            break;  
        case 2: case 4: case 7: case 9: case 10:  
            account.withdraw(fl.getField(pos).getPrice());  
            break;  
        case 6:  
        default:  
            break;  
    }  
    System.out.printf("%s landed on the field nr %d %s and you will receive/pay the price of this field %d\n",  
        player.getName(), pos, fl.getField(pos).getTitle(), fl.getField(pos).getPrice());  
}
```

Vi har en metode der afgør at når en af spillerne opnår 3000 point, så opnås vinderbetingelsen.

```
public boolean winCondition(PlayerList pl) {  
    boolean winCondition = false;  
    for (int i = 0; i < pl.getAccounts().length ; i++) {  
        if (pl.getAccount(i).getBalance() >= 3000) winCondition = true;  
    }  
    return winCondition;  
}
```

Vi har en metode som tager vinderen af spillet og printer hans navn og antallet af point han har vundet spillet med.

```
public void findWinner(PlayerList pl, int playerTurn){  
    System.out.println(pl.getPlayerList(playerTurn).getName() + " Has won with the balance of" + pl.getAccount(playerTurn).getBalance());  
}
```

Vi har en metode som printer det der foregår på spillepladen til konsollen. Den printer spillernavn, resultat af kast, position på brættet og point i konto

```
public void diceInfo(PlayerList pl, Dice dice, int PNum) {  
    System.out.println("Name: " + pl.getPlayerList(PNum).getName() +  
        " Dice Results " + dice.getDie1() + "+" + dice.getDie2() + "=" + dice.getSum() +  
        " Position = " + pl.getPlayerList(PNum).getCurrentPosition() +  
        " Balance: = " + pl.getAccount(PNum).getBalance() );  
}
```

Vi har en metode der printer til konsollen hvis tur det er

```
public void displayTakingTurn(PlayerList pl, int PNum){  
    System.out.println("The player " + pl.getPlayerList(PNum).getName() + " is now taking his turn turn");  
}
```

Vi har en metode der printer hvor mange ture spilleren har haft

```
public void displayTurn(PlayerList pl, int PNum){  
    System.out.println(pl.getPlayerList(PNum).getName() + " has taken: " + pl.getPlayerList(PNum).getTurn() + " Turn(s)");  
}
```

Vi laver get og set metoder til vores positioner, så andre klasser kan kalde på disse værdier eller sætte værdierne til en ønsket værdi.

```
public int getPrePos() { return prePos; }  
public int getPos() { return pos; }  
public void setPos(int pos) { this.pos = pos; }
```

Grafisk brugerflade - Klasser

Game GUI

```
public class GameGUI {  
  
    private GUI gui;  
    private GUI_Field[] fields;  
    public GUI_Player[] gui_players;  
    private GUI_Car[] gui_cars;
```

Gui konstruktionen tager felterne der er blevet oprettet i Board klassen og sender dem videre til game GUI

```
public GameGUI(GUI gui){  
    this.gui = gui;  
    this.fields = gui.getFields();  
}
```

Vi har her en metode der tilføjer spiller til spillebrættet. Metoden starter med at initialisere arrays for hvor mange spiller og biler den skal have plads til. Den tager en længden af playlisten til til at sætte længden af arrays for gui biler og spiller.

```
public void addPlayers(PlayerList pl){  
    gui_cars = new GUI_Car[pl.getPlayerList().length];  
    gui_players = new GUI_Player[pl.getPlayerList().length];
```

Til at starte med bliver der defineret en spiller som beskriver hvilken spiller man arbejder videre med.

Nu kan man fylde arrays ud med hvor bilerne, som bliver givet den samme farve som spilleren. På samme måde følger

```
for (int i = 0; i < pl.getPlayerList().length; i++) {  
    Player player = pl.getPlayerList(i);  
    gui_cars[i] = new GUI_Car(player.getColor(), player.getColor(), GUI_Car.T  
    gui_players[i] = new GUI_Player(player.getName(), balance: 0, gui_cars[i]);  
    gui.addPlayer(gui_players[i]);  
    fields[0].setCar(gui_players[i], display: true);  
    gui_players[i].setBalance(pl.getAccount(i).getStartingBalance());  
}
```

Translator

Translator klassens constructor tager imod en tekststreng der specificere hvilken sprogl fil den skal indlæse.

```
public Translator(String localization){
    switch (localization){
        case "da":
            file = readFile( source: "da.txt").split( regex: "\n");
            break;
        default:
            file = readFile( source: "en.txt").split( regex: "\n");
            break;
    }
}
```

Herefter tildeler den hver tekststreng den tilhørende værdi fra tekstfilen.

```
LandedString = file[1];
WonString = file[3];
DiceInfo = file[5];
TakingTurnString = file[7];
DisplayTurnString = file[9];
FieldToString = file[11];
PlayerNameAction = file[13];
PlayerSelectAction = file[15];
RollDiceAction = file[17];
WonTheGameString = file[19];
fieldsName = Arrays.copyOfRange(file, from: 19, file.length);
```

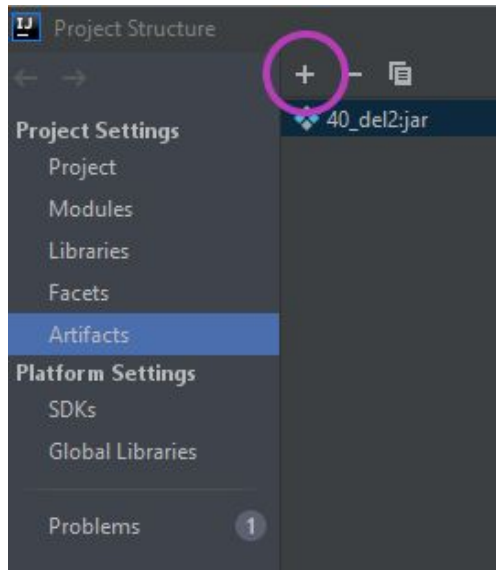
Tekststrengene kan derefter bruges rundt omkring i programmet. f.eks. i Logic klassen, til at udskrive hvor man er landet henne på brættet:

```
System.out.printf(Game.translation.getLandedString(),
    player.getName(), pos, fl.getField(pos).getTitle(), fl.getField(pos).getPrice());
```

Konfiguration

Spillet er testet på java versionen 1.8.0_261. Derfor kan vi kun garantere at spillet kører optimalt på denne version eller nyere versioner af java. Spillet kører også, som det skal på version 1.8.0_271, som lige nu er den seneste version af java til Windows 10. Kompatible windows versioner for den seneste version af java, er ifølge java: 64 bit - Windows Vista SP2, 64 bit -Windows 7, 64 bit -Windows 8 & 64 bit -Windows 10.

For at køre spillet skal brugerne klikke på **File** oppe i venstre hjørne af IntelliJ. Derefter skal man klikke på "Project Structure", så "Artifacts" og klikke på "+"



Derefter skal man vælge "JAR" → "From modules with dependencies". Vælge Main klassen og erstatte "java" i "CDIO2\src\main\java" med "resources", og klikke "ok". Derefter skal man klikke på "Build" oppe i toppen, "Build Artifact" ig så klikke "build".

For at køre spillet, så skal kan man nu finde filen 40_del2.jar, som ligger inde under CDIO2\out\artifacts\40_del2.jar. Spillet kan nu køres med java uden for IntelliJ.


Test

For at kunne se om vores program/spil lever op til de krav, som er blevet stillet og fungere korrekt, er der blevet udarbejdet en række JUnit test, som afprøver og tester enkelte dele af spillet.

Account

klassen Account holder styr på spillerens point. Det er derfor vigtigt at pointbalancen ikke kan blive negativ. Der er 4 metoder i Account klassen som ændrer på balancen. Lige meget hvor mange eller hvor lidt point, spilleren modtager må balancen ikke blive negativ. Det er krav 5 der testes i denne JUnit test.

Den JUnit test som tester metoderne tester 100% af metoderne:

 Account	100% (1/1)	100% (6/6)	83% (15/18)
---	------------	------------	-------------

Test case ID	TC01
Beskrivelse	Test af deposit metode i Account klasse.
Krav	Balancen må ikke være negativ
Præbetingelser	
Postbetingelse r	Balancen ikke er negativ
Test procedure	1. Indsæt beløb. 2. Se om resultatet er negativt.
Test data	Balancen = 1 Beløb der indsættes = -2
Forventet resultat	Balancen = 0
Faktisk result	Balancen = 0
Status	Succes
Testet af	Johannes Jensen
Dato	2020-26-10.
Test Miljø	IntelliJ IDEA 2020.2.2 (Ultimate Edition) Build #IU-202.7319.50, built on September 15, 2020 Runtime version: 11.0.8+10-b944.31 amd64 VM: OpenJDK 64-Bit Server VM by JetBrains s.r.o. Windows 10 10.0 Memory: 953M Cores: 4 Non-Bundled Plugins: org.jetbrains.kotlin on Windows 7 Home Premium Service Pack 1.

Test case ID	TC02
Beskrivelse	Test af withdraw metode i Account klasse
Krav	Balancen må ikke være negativ
Præbetingelser	
Postbetingelse r	Balancen ikke er negativ
Test procedure	1. Hæv beløb. 2. Se om resultatet er negativt.
Test data	Balancen = 1 Beløb der hæves = 2
Forventet resultat	Balancen = 0
Faktisk result	Balancen = 0
Status	Succes
Testet af	Johannes Jensen
Dato	2020-26-10.
Test Miljø	IntelliJ IDEA 2020.2.2 (Ultimate Edition) Build #IU-202.7319.50, built on September 15, 2020 Runtime version: 11.0.8+10-b944.31 amd64 VM: OpenJDK 64-Bit Server VM by JetBrains s.r.o. Windows 10 10.0 Memory: 953M Cores: 4 Non-Bundled Plugins: org.jetbrains.kotlin on Windows 7 Home Premium Service Pack 1.

Test case ID	TC03
Beskrivelse	Test af setBalance metode i Account klasse
Krav	Balancen må ikke være negativ
Præbetingelser	
Postbetingelse r	Balancen ikke er negativ
Test procedure	1. Sæt balancen til et negativt beløb. 2. Se om resultatet er negativt.
Test data	Balancen = -1
Forventet resultat	Balancen = 0
Faktisk result	Balancen = 0
Status	Succes

Testet af	Johannes Jensen
Dato	2020-26-10.
Test Miljø	IntelliJ IDEA 2020.2.2 (Ultimate Edition) Build #IU-202.7319.50, built on September 15, 2020 Runtime version: 11.0.8+10-b944.31 amd64 VM: OpenJDK 64-Bit Server VM by JetBrains s.r.o. Windows 10 10.0 Memory: 953M Cores: 4 Non-Bundled Plugins: org.jetbrains.kotlin on Windows 7 Home Premium Service Pack 1.

Test case ID	TC04
Beskrivelse	Test af addBalance metode i Account klasse
Krav	Balancen må ikke være negativ
Præbetingelser	
Postbetingelse r	Balancen ikke er negativ
Test procedure	1. Indsæt beløb. 2. Se om resultatet er negativt.
Test data	Balancen = 1 Beløb der indsættes = -2
Forventet resultat	Balancen = 0
Faktisk result	Balancen = 0
Status	Succes
Testet af	Johannes Jensen
Dato	2020-26-10.
Test Miljø	IntelliJ IDEA 2020.2.2 (Ultimate Edition) Build #IU-202.7319.50, built on September 15, 2020 Runtime version: 11.0.8+10-b944.31 amd64 VM: OpenJDK 64-Bit Server VM by JetBrains s.r.o. Windows 10 10.0 Memory: 953M Cores: 4 Non-Bundled Plugins: org.jetbrains.kotlin on Windows 7 Home Premium Service Pack 1.

Alle test er succesfulde, kravet er dermed opfyldt.

Dice

Dice klassen, indeholder alt logik der har relation til terningerne i spillet. Det er derfor vigtigt at denne klasse fungerer efter hensigten. JUnit testene tester bl.a. om terningerne bliver slået rigtigt.

Test case ID	TC01
Beskrivelse	Test af roll metode i Dice klasse
Krav	Terningerne skal returnere en sum
Præbetingelser	
Postbetingelser	
Test procedure	1. Slå med terningerne. 2. Se om terningerne er blevet slået.
Test data	To terninger der bliver slået
Forventet resultat	Sum af terning 1 + sum af terning 2
Faktisk resultat	Sum af terning 1 + sum af terning 2
Status	Succes
Testet af	Krzysztof Kisiel
Dato	2020-30-10.
Test Miljø	IntelliJ IDEA 2020.2.3 (Ultimate Edition) Build #IU-202.7660.26, built on October 6, 2020 Runtime version: 11.0.8+10-b944.34 amd64 VM: OpenJDK 64-Bit Server VM by JetBrains s.r.o. Windows 10 10.0 GC: ParNew, ConcurrentMarkSweep Memory: 953M Cores: 4 Non-Bundled Plugins: org.jetbrains.kotlin

Da roll metoden er den eneste interessante metode, som alle de andre metoder i Dice klassen bruger, er der ikke flere metoder der er nødvendige at teste.

Nemt at oversætte

Krav 8. siger at spillet skal være nemt at oversætte. Derfor er der blevet implementeret en klasse, der læser en sprogfil (.txt) ind i det valgte sprog. Sproget kan man vælge ved at give det med som parameter når man kører programmet. Formatet er "da" for dansk eller "en" for engelsk.

Program arguments: + ↵

eller

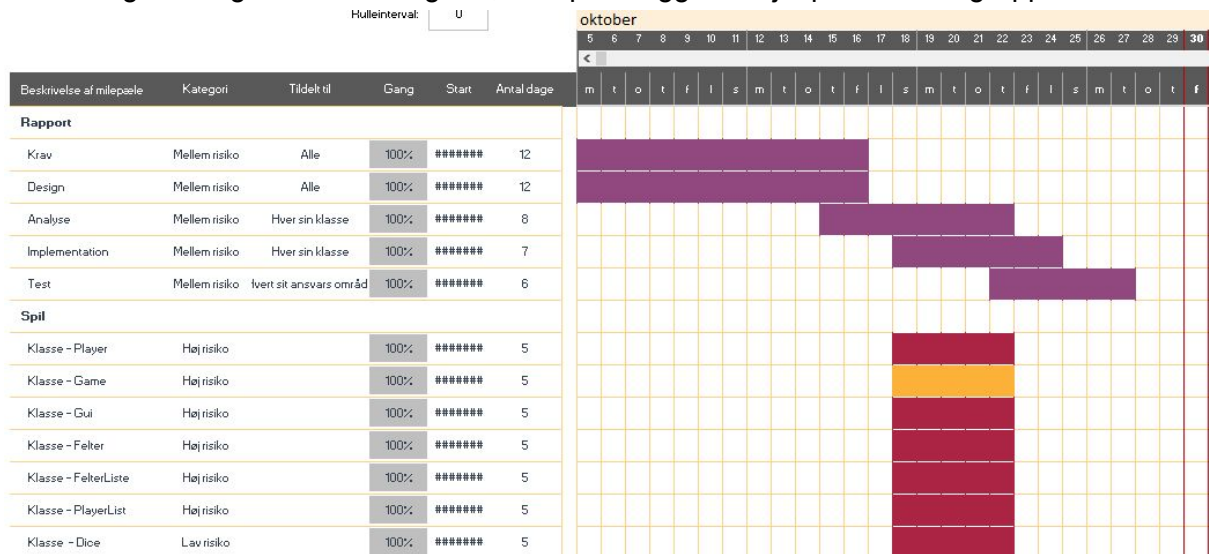
Program arguments: + ↵

Dette kan også specificeres i terminalen, hvis programmet køres derfra. Derudover er det nemt at oversætte til andre sprog f.eks. tysk, her skal man ændre i da.txt og give "da" med som parameter. Det er dog vigtigt at ikke ændrer de bogstaver hvor der % bagved.

```
1  LandedString
2  %s landed på plads nr %d %s og du vil modtage/betale prisen på dette felt %d%n
3  WonString
4  Du har vundet med en balance på:
5  DiceInfo
6  Navn: %s  Terninge resultat: %d + %s = %d Position = %d  Balance: = %s
7  TakingTurnString
8  Spilleren %s tager nu sin tur
9  DisplayTurnString
10 %s har taget: %d Tur(e)
11 FieldToString
12 Title: %s \n Pris: %d \n  Beskrivelse %s \n
13 PlayerNameAction
14 Indtast spiller navn:
15 PlayerSelectAction
16 Vælg 2-6 spillere
17 RollDiceAction
18 vil du slå med terningerne?
19 WonTheGameString
20 Har vundet spillet med en balance på:
21 fieldsName
22 Dette er det første felt
23 Tårn
24 Krater
25 Paladsets porte
26 kold ørken
27 Beskyttet by
28 Kloster
29 Sort hule
30 Hytter i bjergene
31 Vareulvvæggen
32 Hullet
33 Guldminen
```

Projektplanlægning

Vi har valgt at bruge et Gantt-diagram, til at planlægge arbejdsprocessen i gruppen.



I Gantt-diagrammet har vi vurderet opgavernes risiko, og tildelt opgaverne imellem os. Vi har løbende opdatere diagrammet, og har dermed kunne følge med i processen. Vi har derudover holdt ugentlige møder, for at holde hinanden opdateret om eventuelle problemer eller andet.

Konklusion

Der er i opgaven blevet designet og udviklet et spil, ud fra de givne krav. Der er blevet anvendt relevante artefakter og GRASP mønstre til at designe og implementere spillet i Java. Spillet er derefter blevet testet med en række JUnit test, for at dokumentere at spillet lever op til kravene, som var blevet stillet.

Kilder

Budtz ,C. Pedersen, M. Dalsgaard ,R. Nyborg, M. Matador_GUI. 14 november 2019
https://github.com/diplomit-dtu/Matador_GUI

Aflæst d. 30/10-2020

https://www.java.com/en/download/win_64sysreq-sm.jsp

Linket fører til java's side med kompatible windows versioner.