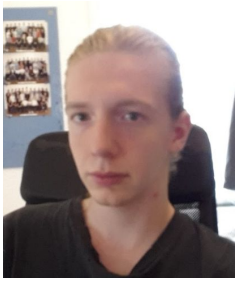
Rapport CDIO - 2

Gruppe nr: 16
Indledende programmering
Versionsstyring og testmetoder
Udviklingsmetoder til IT-systemer

Gruppemedlemmer:



Jens Daniel Kramhøft
s175445

Michael Lund Jarberg
s185091

Rasmus Nielsen
s185101

Helle Achari
s180317

Tamanna Zkria
s185019

Timeregnskab

Overordnet timeregnskab. Se bilag for detaljeret timeregnskab

	Analyse	Design	Kode	Dokumentation	Versionsstyring
Michael	1.3	2	10	6.75	3.5
Jens Daniel	1.75	5.5	15	12	3.6
Helle	4.7	0.5	1	3.5	
Rasmus	2	5	5	2	1
Tamanna	2				
I alt	11.75	13	31	24.25	8.1

Indholdsfortegnelse

Resume

I denne opgave har vi bl.a. arbejdet med at analysere, designe og opstille starten på et matador spil ud fra kravene opstillet i opgavebeskrivelsen. Vi har analyseret de krav, der er blevet stillet til os ved bl.a. at opstille en domænemodel, finde use case og opstille use case diagrammer. Det har ledt til, at vi bl.a. via. design-klasser har fået en guide-line til vores kode. Efterfølgende har vi analyseret vores kode og dens klasser og metoder ift. GRASP-patterns. Sidst, så er der eksempler fra vores kode og test som er kommenteret.

Indledning

I vores opgave, så har vi delt opgaven op i hovedpunkter og underpunkter til hvert punkt. Det er også illustreret i vores indholdsfortegnelse. Vores opgave er opdelt i krav, analyse, design, implementering, test, projektforløb, konklusion og sidst vores bilag.
I vores krav, har vi arbejdet med visionen, kravene der er stillet til os i opgavebeskrivelsen og efterfølgende delt dem op i funktionelle -og ikke-funktionelle krav. Efterfølgende har vi bevæget os videre hen i analysen, hvor vi har analyseret vores use cases. Det har været i form af brief og fully-dress beskrivelser, use-case diagrammer, Sekvensdiagram m.m. Vores analyse har dannet et fuld forståelse for vores use case diagrammer, så efterfølgende er vi gået videre til vores design af koden.
I vores designdel har vi benyttet os af design-klassediagram, sekvensdiagram m.m. Herunder har vi også analyseret vores kode via. GRASP-principper med eksempler fra klasserne i koden.
I implementar afsnittet, så er der eksempler fra vores kode, hvor klasserne, klassernes attributter og deres metoder med tilhørende kommentare.
Det næste afsnit er test, hvor vi tester vores færdige kode. Her er der udplukket de mest interessante eksempler, som er sat ind under afsnittet.
Sidst indeholder vores opgave et projektforløb samt en konklusion.

Krav

Vision

Det skal være et spil mellem 2 personer, der kan spilles på maskinerne i DTU's databarer, uden bemærkelsesværdige forsinkelser.

Spillerne slår på skift med 2 terninger og lander på et felt med numrene fra 2 - 12. At lande på hvert af disse felter har en positiv eller negativ effekt på spillernes pengebeholdning. Derudover udskrives en tekst omhandlende det aktuelle felt.

Spillerne starter med en pengebeholdning på 1000. Spillet er slut når en spiller når 3000. Spillet skal let kunne oversættes til andre sprog. Det skal være let at skifte til andre terninger.

Vi vil gerne have at I holder jer for øje at vi gerne vil kunne bruge spilleren og hans pengebeholdning i andre spil.

Navneords-analyse

- Et spil
- En spiller
- En modstander
- En tur
- En terning
- Et raflebæger
- Et kast/slag
- Et bræt
- Et felt
- En pengebeholdning
- En vinder
- En taber

Kravspecifikation

Funktionelle krav

1. Spillet skal kunne spilles mellem 2 personer.
2. Spillerne skal slå på skift med to terninger
3. To terninger lander på et felt fra 2-12
4. Spilleren bliver ramt af en positiv eller negativ effekt afhængig af feltet.
5. Der skal udskrives en beskrivende tekst når man lander på et felt.
6. Felterne skal udvikles ud fra listen angivet i opgavebeskrivelsen.
7. Enhver spille skal have et startbeløb: 1000 pr spiller
8. En spiller skal vinde hvis hans/hendes konto kommer op på 3000 eller over.
9. Spilleren og hans pengebeholdning skal kunne bruges i andre spil (Lav kobling)
10. Spillet skal spilles uden bemærkelsesværdige forsinkelser.
11. En spillers konto må aldrig kunne gå under 0

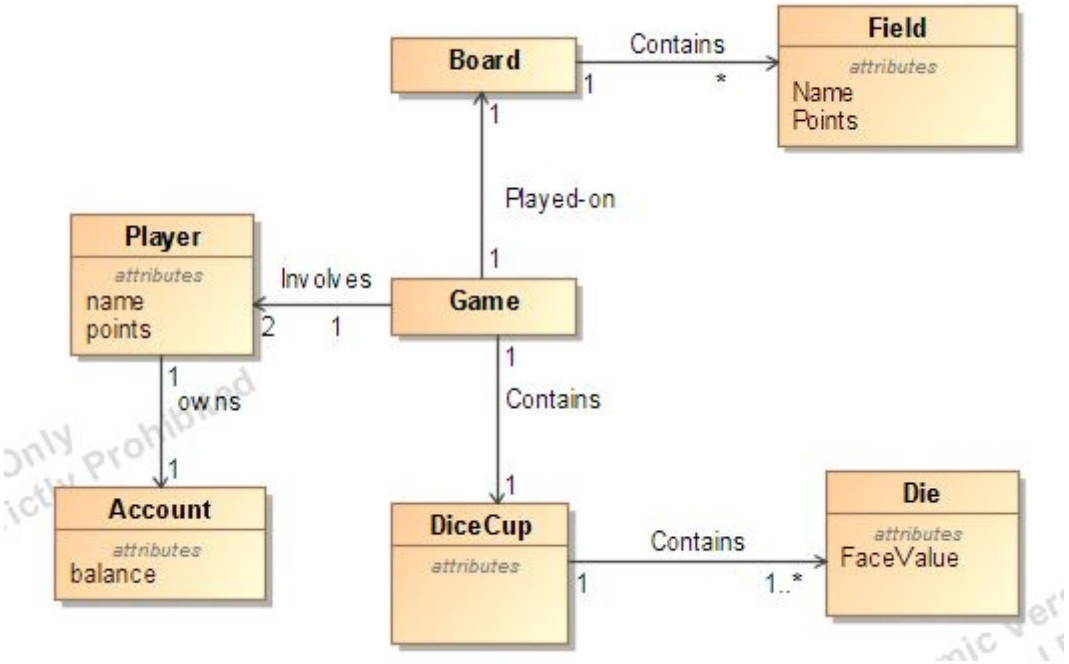
Ikke-funktionelle krav

1. Spillet skal let kunne oversættes til andre sprog.
2. Det skal være let at skifte til andre terninger.
3. Spillet skal kunne spilles på maskinerne i DTU's databarene.
4. Kan spilles på operativsystem Windows.
5. Projektet skal indeholde et .git repository med commits (hvem, hvornår).
6. Projektet skal udvikles på en sådan måde at man kan se de enkelte gruppemedlemmers bidrag og følge med i hvordan udviklingen er foregået.
7. Projektets versionering skal udvikles med hyppige (atomic) commits med beskrivelser.
8. spiller klassen og konto klassen skal holdes adskildt

Analyse

Domænemodel

Nedenstående er vores domænemodel. Klasserne er taget ud fra et kunde-synsvinkel i den virkelige verden. Der er et spil, der bliver spillet på et bræt, som indeholder nogle felter. Spillet involverer to spillere der hver har en konto, og spillet indeholder et raflebæger med to terninger.



Use-cases

Brief

1. "Start spil" - Spiller starter spillet.
2. "Spil tur" - Den aktive spiller udfører sin tur.

Fully-dressed Use case beskrivelser

Vores use case *"Start spil"* sker hver gang to spillere vil begynde et spil. For at kunne begynde et spil, skal brugerne indtaste deres brugernavne. Når de har gjort det, så kan de begynde spillet, og slå terningerne. Turen vil skifte imellem de to brugere.
Spillet er gennemført når en vinder er fundet også kan spillet starte om igen. Her gælder de samme forudsætninger som foroven.

Navn:	Start spil
ID:	1
Kort beskrivelse:	Spiller starter spillet.
Primær aktør:	Spiller
Main flow:	**1.** Spiller starter spillet **2.** Systemet beder om navn fra spillerne **3.** Spiller 1 og 2 indtaster deres navn **4.** Systemet opretter de 2 spillere samt hvad der er nødvendigt for at spille spillet.
Postcondition:	Spillet er startet og det er Spiller 1s tur.
Alternative Flows:	

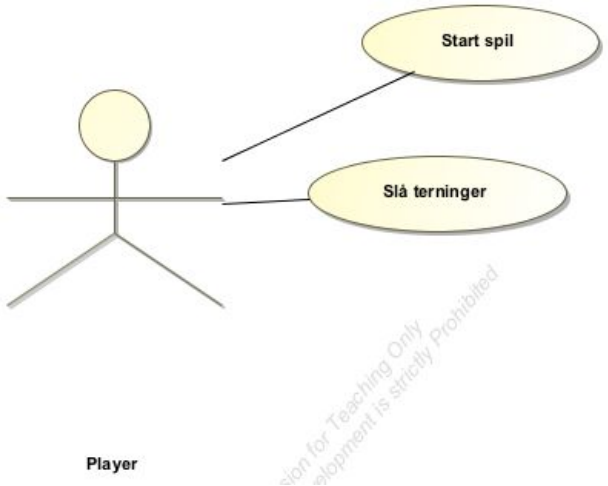
Vores anden use case *"Spil tur"* sker hver gang spillerne har en tur. Her skal spillet være gået igang, og turene skal være gået i gang. Her slås der med de to terninger vi har i vores raflebæger som giver en samlet sum. Denne sum er også tallet på et felt, som spilleren bliver rykket hen på. Hvert felt har en positiv og negativ påvirkning på spillerens pengebeholdning, hvilket til sidste resulterer i en vinder og en taber. Det kræver at turene skifter, altså der bliver slået med terningerne, indtil der er fundet en vinder og en taber.

Navn:	Spil tur
ID:	2
Kort beskrivelse:	Den aktive Spiller kaster med terningerne og får resultatet tilbage
Primær aktør:	Spiller
Precondition:	Det er en Spillers tur
Main flow:	**1.** En spiller kaster med terningerne. **2.** Systemet slår to Terninger og får en samlet værdi. **3.** Spilleren rykker til et felt ud fra terningernes værdi og får uddelt point fra feltet. **4.** Spillerens tur afsluttes.
Postcondition:	Det er en ny spillers tur eller der er fundet en vinder.
Alternative	

Flows:	

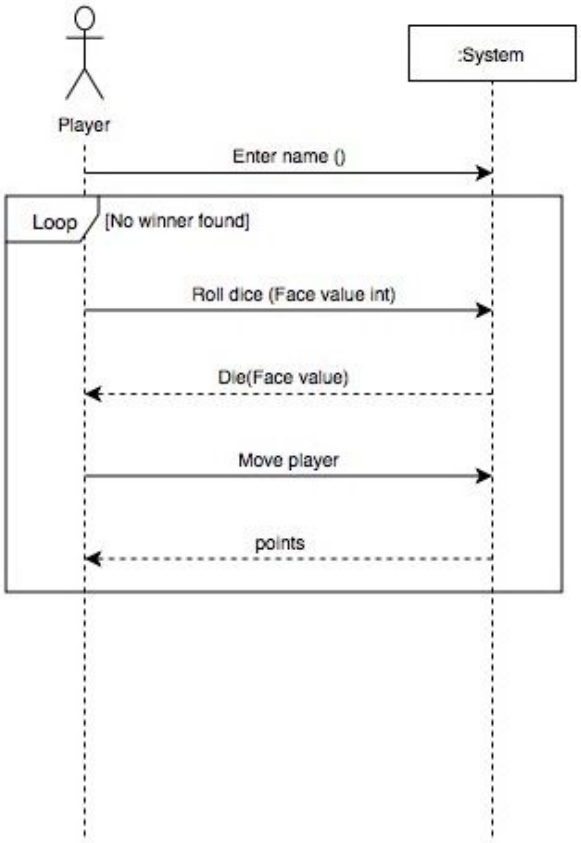
Use case diagram

Nedenstående er der illustreret et use case diagram, hvor vi ser sammenhængen mellem den primære aktør samt use cases. Vores primær aktør er spilleren, og vores use cases er *"Start spil"* og *"Slå terninger"*.



System sekvens diagram

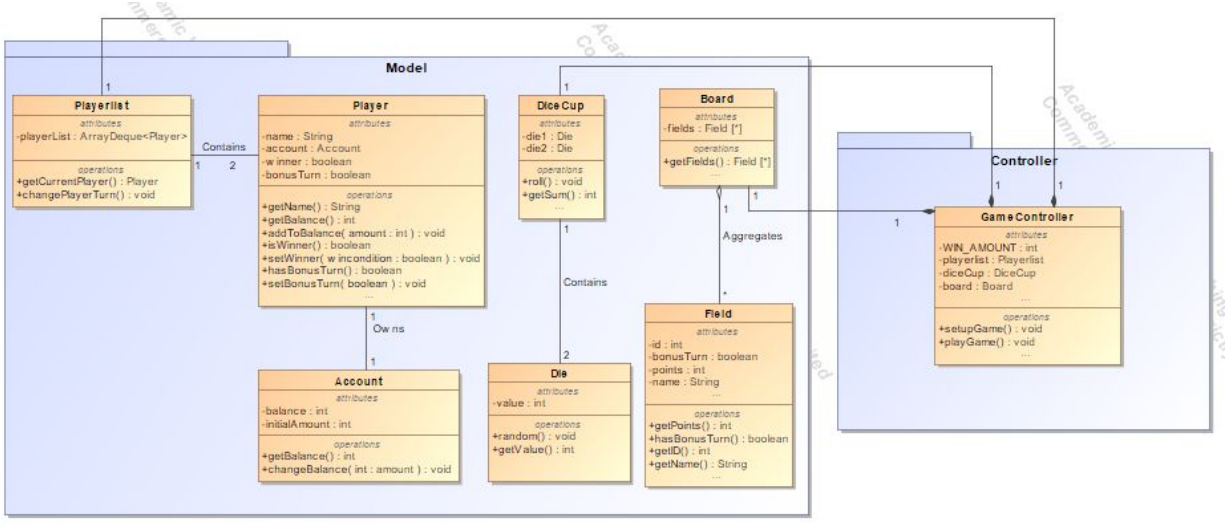
Nedenstående er vores system sekvens diagram, hvor vi ser hvordan spilleren og systemet kommunikerer. Når spilleren "slår terninger", så returnerer systemet en "Die (Face value)" altså en besked til spilleren om den slået terninge værdi.


:System
Player
Enter name ()
Loop
[No winner found]
Roll dice (Face value int)
Die(Face value)
Move player
points

Design

Design klassediagram

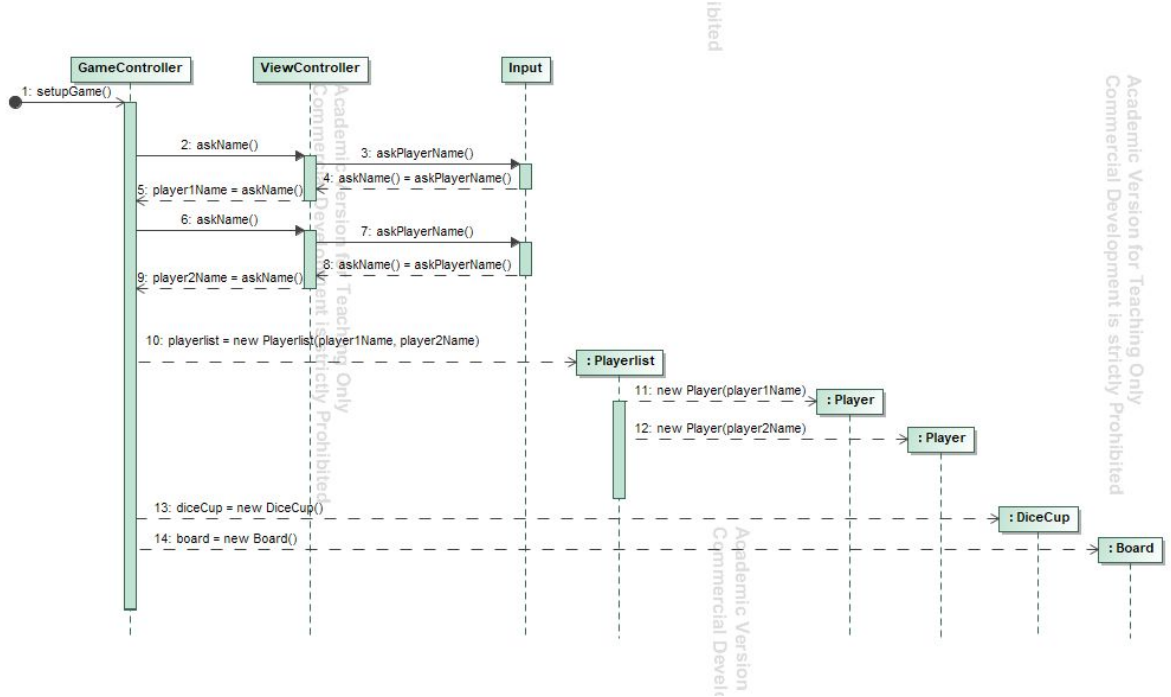
Nedenstående ses et design klassediagram over klasserne i vores Model pakke. Det viser relationerne mellem klasserne, samt klassernes variabler og metoder. Mest notérbart er hvordan relationerne er uddelt for at fremme lav kobling. Diagrammet inkluderer også 'GameController' klassen fra Controller pakken, da den er essentiel i forståelsen for hvordan, og af hvem, de andre klasser bruges.
Det ses fx. at hver instans af klassen 'Player' har en instans af klassen 'Account', og at 'Board' klassen er en samling af 'Field' objekter.



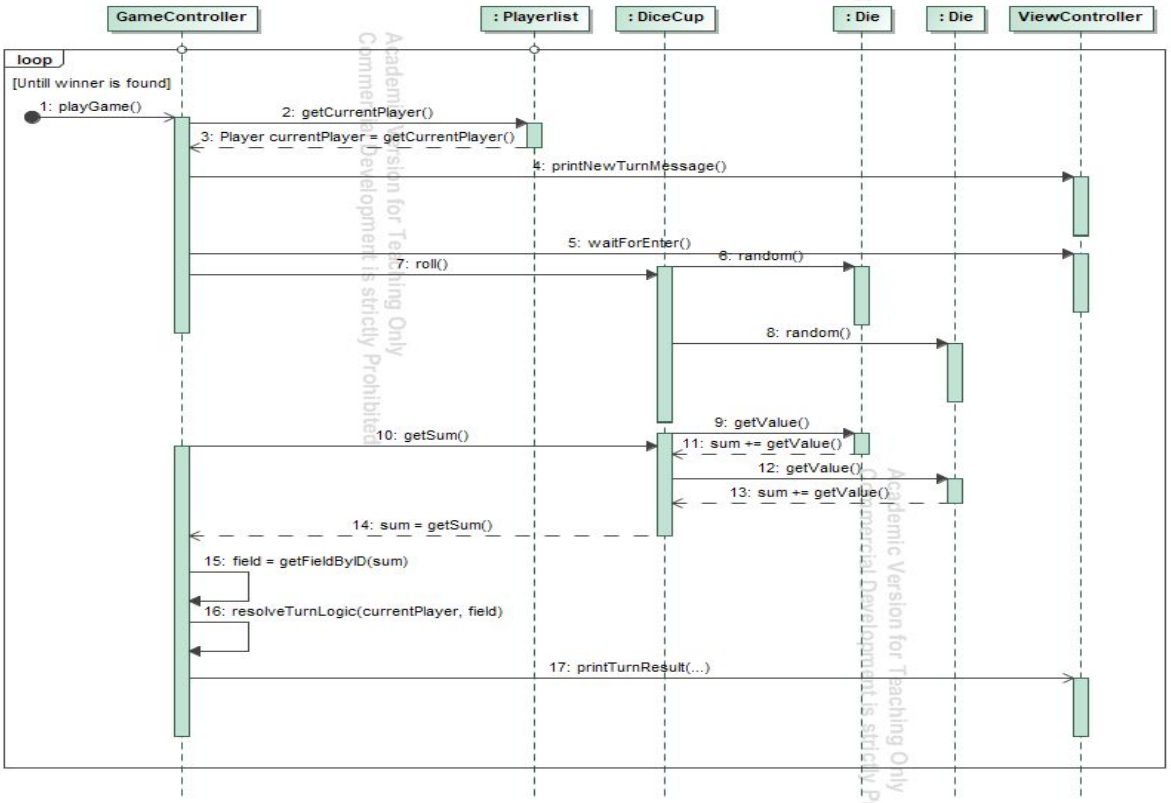
Sekvensdiagrammer

Der er udarbejdet to sekvensdiagrammer, der viser det primære succes-scenarie i vores to use cases, 'Start spil' og 'Spil tur'.

Use case 1 “Start spil”. Her ser vi programmets klasser, og hvilke beskeder der bliver sendt og returneret til hvilke klasser, når spilleren vil starte spillet.



Her er use case 2 “ Spil tur” illustreret, hvor vi kan se hvilke klasser og metoder der bliver kaldt og hvad der bliver returneret tilbage til spilleren.



Pakker og Ansvarsområder

Nedenstående beskrivelse er, hvordan vi har anvendt GRASP-mønstret i vores spil. Derudover, så er det beskrevet klassernes ansvarsområder og deres metoder, og hvornår de bliver kaldt.
Spillets pakker består af den overordnede pakke **DTU.SWT_grp16** som under sig har pakkerne
Application
Controller
Model
View

Application
Pakken indeholder **Main java klassen** hvorfra spillet bliver startet fra.

Controller pakken
Indeholder klasserne **GameController** samt **ViewControlleren** hvor **GameControllerens** hovedansvar er kommunikation mellem klasserne i **Model, Application** og **Controller** pakkerne.

Model pakken
Indeholder alle de afhængige underklasser. Eksempelvis for at starte spillet bliver **GameControlleren** objektet skabt i **Main klassen** og dens Metoder, **setupGame**() og **playGame**() kaldt.

setupGame() gør, kort sagt, at **GameController** objektet skaber andre objekter fra klasserne i **Model** pakken og overføre deres nødvendige informationer fra andre instantierede objekter som **Board** klassen, der skal bestå af en bestemt mængde af **Field** objekter, der alle skal have indlæst parametre som navn, id, points, og bonustur. **Field** objekternes Parametre bliver indlæst af en textfil af Input klassen som kun står for at modtage input fra brugeren samt systemet.
Dette har været hensigten på baggrund af GRASP principperne.

Anvendte GRASP-principper

Høj Binding
Da klasserne kun indeholder de metoder inden for det område de skal bruges i, overholdes princippet I modsætning til at have fks, en input klasse der også har output klasse funktioner. Pakken View med klassen Input er et godt eksempel da denne består af metoderne, der alle enten tager et input fra brugeren eller indlæser filer baseret på brugerens input.
Dets metoder er:

```
askPlayerName
waitForInput
waitForEnter
directoryList
readFileField
getLanguageChoices
```

Controller

Vi har overholdt Controller princippet ved at have vores to klasser der trækker på strengene "GameController" og "ViewController" i vores Controller pakke, og ladet dem være de klasser der styrer showet. Det vil sige at det er vores ViewController der er ansvarlig for håndtering af input og output til brugeren. Der er ikke brug for flere controllers af input og output til dette spil.
Metoderne i ViewControlleren har alle det tilfælles at de outputer til GUI eller laver GUI

Lav kobling

Princippet er også overholdt da der er ingen til lav interaktion internt og mellem **View** og **Model** pakkerne.
Klassen Playerlist skaber og holder styr på instanser af Player klassen, hvor Player klassen holder styr på instanser af Account klassen. Playerlist klassen behøver derfor ikke at vide noget til Account klassen.
Denne opsætning gør der er minimal afhængighed mellem klasser og ændringer betyder mindre for de andre klasser. Dette ses også på vores designklasse diagram.

Creator

'Creator' princippet omhandler hvornår en klasse skal være ansvarlig for at skabe en instans af andre klasser. Løsningen på den problemstilling indebærer at klasse 'B' skal være ansvarlig for at skabe en instans af klasse 'A' hvis B er en samling A objekter (B aggregerer A objekter), hvis B indeholder eller benytter A objekter, eller hvis B har initialiserede data som A behøver når det skabes.
Det ses at dette princip er overholdt i vores kode ved at se på hvornår vi skaber instanser af forskellige objekter.
Eksempler:
Instanser af klassen 'Die' bliver kun lavet af klassen 'DiceCup'. DiceCup må ifølge princippet gerne lave instanser af Die, da den indeholder og benytter Die objekter.
På samme måde bliver instanser af klassen 'Account' kun lavet i forbindelse med skabelse af et 'Player' objekt, da Player klassen indeholder en instans af Account som den benytter til at holde styr på sine point.

Vores ‘Board’ klasse er en samling af ‘Field’ objekter. Derfor har klassen ‘Board’ ansvar for at lave instanser af klassen ‘Field’.
Vores GameController skaber instanser af de øverste klasser fra Model pakken, Playerlist, Board og DiceCup. Det stemmer overens med princippet da det er vores GameController der bruger objekterne i spillet.

Implementering

Systembeskrivelse med pakker og ansvar

I projektet er klasserne tildelt en rolle som enten er Model, View eller Controller, og hver klasse er kommet i den passende pakke med et veldefineret ansvar. Opdelingen er sket for at opnå høj samhørighed og lav kobling, hvilket gør det lettere at arbejde sammen som gruppe på et projekt, lettere at forstå og lettere at vedligeholde ved kun at ændre få filer.
Vi har opbygget det med en Controller 'GameController' der har ansvar for at snakke og skabe instanser af objekterne i 'Model' pakken.
GameController klassen udfører selv spillets logik og ændrer på de passende objekter, og sender den resulterende information til en 'ViewController' som er ansvarlig for at snakke med klasserne i View pakken. View pakkens klasser består bl.a. af en Output klasse der sender information til brugeren, og en Input klasse der læser information fra konsollen og tekstfiler i projektet.
Kort sagt har klasserne i Model pakken ansvar for deres individuelle del af spillet, klasserne i View pakken har ansvar for at kommunikere med brugeren, og klasserne i Controller pakken har ansvaret for at trække trådene.

Objektorienteret kodeeksempel - Player + Account

Player klassen er et enkelt eksempel på simpel objektorienteret implementering i vores projekt. Alt Player klassen indeholder er sine egne private attributter og public metoder samt en konstruktor.

```
package DTU.SWT_grp16.Model;

public class Player {

   private String name;
   private Account account;
   private boolean winner;
   private boolean bonusTurn;

   public Player(String name){
       this.name = name;
       this.account = new Account();
   }

   public String getName() {
       return name;
   }

   public int getBalance(){
       return account.getBalance();
   }
   public void addToBalance(int amount){
       account.changeBalance(amount);
   }
```

Attributterne består af en reference, name, til et String objekt, en reference, account, til en instans af klassen Account, og to booleans (winner og bonusTurn).

Der er passende metoder til anvendelse af disse attributter, kun nogle af dem ses i kodeeksemplet. Den mest interessante metode er måske *addToBalance*, som kalder en metode på spillerobjektes egen reference til et Account objekt. Metoden i Account klassen sørger for at dets private felt 'balance' aldrig bliver negativ vha. noget simpel forgrening, og Player objektet benytter funktionaliteten som 'Account' objektet bringer.
Konstruktøren tager en String som parameter og peger sin egen String reference hen på samme String objekt. Derudover laves en ny instans af klassen Account til spilleren. Det ses i Account klassen af den starter med en balance på 1000.
Gennem projektet har vi holdt samme konvention af private attributter og public metoder der repræsenterer Constructor, private public. Det er relationen mellem dem der giver player dens funktionalitet, som man er interesseret i og derfor bruges.
Lige i vores tilfælde er der ingen toString metoder, da de ikke vurderes nødvendige eftersom anvendelsen af en GUI med getName vurderes nok.

Spillogik - Spillet starter op

Vi har benyttet koncepterne i vores sekvensdiagram til at udvikle nedenstående.
Selve spillet udføres i to public metoder i GameControlleren (Den anden ses længere nede). Den første public metode "*setupGame*" har ansvaret for at starte spillet op, og den anden public metode (ses længere nede i opgaven) "*playGame*" for at spille spillet. Her ses setupGame():

```
public void setupGame(){
   setupLanguageChoice();
   createGameObjectsAndSetupGUI();
   createPlayersWithGUI();
}
private void setupLanguageChoice(){
   view.printLanguageChoiceList();
   currentLanguage = view.setlanguage();
   setFilePath();
}
private void createGameObjectsAndSetupGUI(){
   diceCup = new DiceCup();
   board = new Board(filePath+"\\Fields.txt");
   viewController.guiSetup();
   for(Field field : board.getFields()){
       int fieldID = field.getID();
       String name = field.getName();
       if(fieldID >= 2)
           viewController.addFieldToGUI(fieldID,name);
   }
   viewController.makeGUI();
}
private void createPlayersWithGUI(){
   String player1Name = viewController.askName();
   String player2Name = viewController.askName();
   if(player1Name.equals(player2Name))
       player2Name = player1Name + "#2";
   playerlist = new Playerlist(player1Name, player2Name);
   viewController.addPlayers(player1Name, player2Name);
}
```

Metoden er opdelt i tre private metoder med passende navne så den er lettere at læse: "*setupLanguageChoice", "createGameObjectAndSetupGUI"* og "*createPlayersWithGUI"*. Det giver også mulighed for at man kan kollapsere sine undermetoder i sin IDE når man koder, hvilket hjælper med at danne overblik.

Den første private metode "*setupLanguageChoice"* giver brugeren mulighed for at vælge sprog. Det er designet så man let kan komme ind og tilføje sprogvalgmuligheder, og i nuværende tilfælde er der to: dansk og engelsk. Da brugergrænsefladen bruges til dette bedes ViewControlleren om at skaffe informationen.
Den anden private metode *"createGameObjectsAndSetupGUI"* opretter alle objekterne der ikke er spillere, i dette tilfælde en instans af klassen DiceCup og Board. Brættets felter sættes op ud fra en tekstfil, og vores ViewController sørger for at vores GUI kommer op og køre ved at tilføje alle felterne i et loop.
Den sidste private metode '*createPlayersWithGUI*' bruger den nylavede GUI til at spørge brugerne om spillernavn 1 og 2. Herefter oprettes spillerne som objekter i vores Spillerliste og tilføjes til GUI'en.

Spillogik - Turene spilles (Controller orienteret)

Selve spillets gang styres af den anden public metode i GameController "*playGame"*. Turene bliver udført ved først at den nuværende spiller bliver hentet fra spillerlisten. Herefter slås terningerne, det rigtige felt hentes (gennem den private metode getFieldByID), og resultaterne anvendes i den anden private metode (resolveTurnLogic) med spilleren og feltet som parametre. GUI'en viser resultaterne, og metoden køres rekursivt igen hvis ingen vinder er fundet.

```
public void playGame(){
   Player currentPlayer = playerlist.getCurrentPlayer();
   String currentName = currentPlayer.getName();
   viewController.printNewTurnMessage(currentName);

   diceCup.roll();
   int sum = diceCup.getSum();
   Field currentField = getFieldByID(sum);
   resolveTurnLogic(currentPlayer, currentField);
   int id = currentField.getID();
   int balance = currentPlayer.getBalance();
   boolean bonusTurn = currentPlayer.hasBonusTurn();
   boolean isWinner = currentPlayer.isWinner();
   String playerName = currentPlayer.getName();
   int fieldPoints = currentField.getPoints();
   viewController.printTurnResult(sum, id, balance, bonusTurn, isWinner, playerName, fieldPoints);

   boolean gameIsNotOver = !currentPlayer.isWinner();
   if(gameIsNotOver){
       playerlist.changePlayerTurn();
       playGame();
   }
   viewController.closeScanner();
}
```

Herunder ses spillets logik, hvor effekten fra feltet udføres ved at spilleren har fået et vis antal point til, for at se om brugeren har fået en bonustur eller har vundet. Vi anvender forgrening til at styre dette.

```
private void resolveTurnLogic(Player player, Field field){
   int pointChange = field.getPoints();
   player.addToBalance(pointChange);

   if(field.hasBonusTurn())
       player.setBonusTurn(true);

   if(player.getBalance() >= WIN_AMOUNT){
       player.setWinner(true);
   }
```

```
}
```

Spillerliste - Kort forklaring af ArrayDeque<>() funktionalitet

Projektet anvender en Collection for at holde styr på hvis tur det er. Hvordan Collections virker går ud over pensum, så forklaringen er primært om funktionaliteten. En instans af 'Playerlist' klassen har en privat reference til en implementation af et Deque (Double ended queue) der bliver initialiseret i konstruktøren (Med implementationen ArrayDeque<>()) hvor spillere bliver lavet og tilføjet. Denne form for løsning er valgt da den er let at arbejde med og utrolig fleksibel, så der ville være god mulighed for at forøge fx. mængden af spillere i spillet.

```
public class Playerlist {

   private Deque<Player> playerList;

   public Playerlist(String... playerNames){
       playerList = new ArrayDeque<>();
       createAndAddPlayers(playerNames);
   }
```

Playerlist har ansvaret for at skabe spiller objekter. Konstruktøren modtager et array af referencer til Strings og bruger disse til at lave passende spillere i 'createAndAddPlayers':

```
private void createAndAddPlayers(String... names){
   for(String name : names){
       playerList.addLast(new Player(name));
   }
}
```

Playerlisten har to vigtige metoder, en accessor til den næste spiller, og en til at skifte tur. Her ses acessoren der henter den nuværende spiller:

```
public Player getCurrentPlayer(){
   return playerList.getFirst();
}
```

Den mere interessante metode ses herunder:

```
public void changePlayerTurn(){
   Player currentPlayer = getCurrentPlayer();
   boolean playerHasBonusTurn = currentPlayer.hasBonusTurn();

   if(playerHasBonusTurn){
       currentPlayer.setBonusTurn(false);
       return;
   }
   playerList.addLast(playerList.pollFirst());
}
```

Metoden fungerer ved først at hente den nuværende spiller og undersøge om han/hun har en bonustur. Hvis spilleren har en bonustur bliver spilleren sat til ikke at have en bonustur, og metoden returnerer uden at gøre noget andet. På den måde vil det nu være den samme spillers tur igen. Hvis spilleren ikke har en bonustur trækker metoden spilleren ud fra sin først plads i køen, og smider ham ind bagerst i køen igen. Dette er en dynamisk løsning, da den også vil virke med flere spillere.

Test

Test cases

Høj prioritet

Næste test case var at en spillers point aldrig må gå under 0.

```
@Test
void getBalance() {
    //positiv test
    assertEquals(1000,account.getBalance());
    //negativ test
    assertNotEquals(0,account.getBalance());
}
```

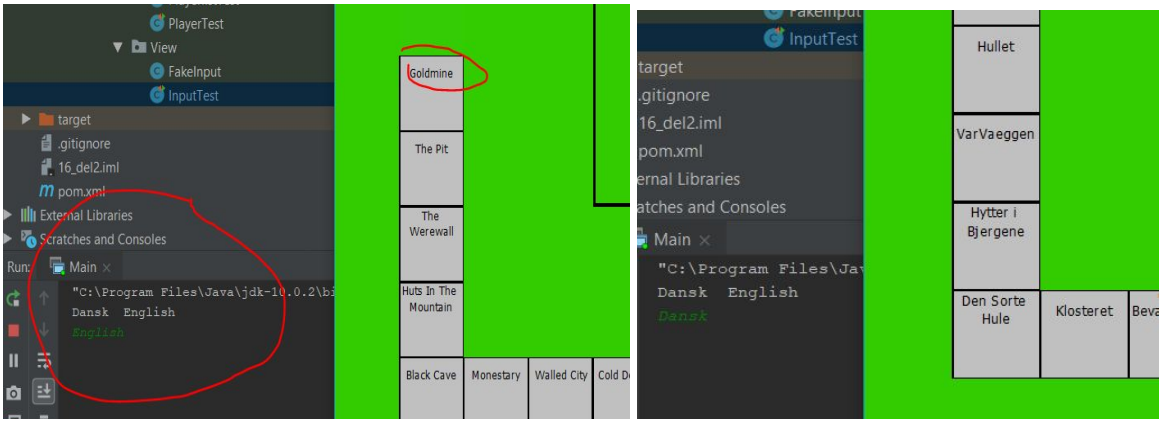
Spilleren har som start 1000 point på kontoen så den positive unit test kigger på det er sandt en account starter med 1000 som efterfølges af en negativ test der sikrer, at den ikke bare starter på 0. Det er testet, og testen virker.

```
@Test
void changeBalance() {
    //positiv test
    account.changeBalance(1000);
    assertEquals(2000,account.getBalance());
    account.changeBalance(-3000);
    assertEquals(0,account.getBalance());
}
```

Nu ved vi getBalance metoden virker, så der kan nu testes for om balancen kan gå under 0.
Der tilføjes med changeBalance metoden 1000 point til kontoen der nu har 2000 som forventet og herefter fjernes der 3000 point fra kontoen og der checkes om kontoen er præcis 0.
Det er testet, og kontoen bliver 0, altså testen virker.

Et andet krav er at det skal være nemt at oversætte spillet til andre sprog.

Sproget i spillet er opsat så teksten bliver indlæst fra en text fil med dens overmappe som sprognavn.
En manuel test af spillet viser at dette er tilfældet da ved spillets start bliver brugeren spurgt om sprog præference.



```
class InputTest {
Input input = new FakeInput();

   private String filePath1 =
"src\\main\\textfiles\\Dansk\\Fields.txt";
   private String filePath2 =
"src\\main\\textfiles\\English\\Fields.txt";
   @Test
   void mReader() {
       // positiv danish test

assertEquals("Taarnet:250:false",input.mReader(filePath1).get(0));

assertEquals("Krateret:-100:false",input.mReader(filePath1).get(1)
);
       assertEquals("Palads
Porten:100:false",input.mReader(filePath1).get(2));
       assertEquals("Den Kolde
Oerken:-20:false",input.mReader(filePath1).get(3));
       assertEquals("Den Bevaeggede
By:180:false",input.mReader(filePath1).get(4));

assertEquals("Klosteret:0:false",input.mReader(filePath1).get(5));
       assertEquals("Den Sorte
Hule:-70:false",input.mReader(filePath1).get(6));
       assertEquals("Hytter i
Bjergene:60:false",input.mReader(filePath1).get(7));

assertEquals("VarVaeggen:-80:true",input.mReader(filePath1).get(8)
);

assertEquals("Hullet:-50:false",input.mReader(filePath1).get(9));

assertEquals("Guldminen:650:false",input.mReader(filePath1).get(10
));

       //positiv english test

assertEquals("Tower:250:false",input.mReader(filePath2).get(0));

assertEquals("Crater:-100:false",input.mReader(filePath2).get(1));
       assertEquals("Palace
Gates:100:false",input.mReader(filePath2).get(2));
       assertEquals("Cold
Desert:-20:false",input.mReader(filePath2).get(3));
       assertEquals("Walled
City:180:false",input.mReader(filePath2).get(4));

assertEquals("Monestary:0:false",input.mReader(filePath2).get(5));
       assertEquals("Black
Cave:-70:false",input.mReader(filePath2).get(6));
       assertEquals("Huts In The
Mountain:60:false",input.mReader(filePath2).get(7));
       assertEquals("The
Werewall:-80:true",input.mReader(filePath2).get(8));
       assertEquals("The
Pit:-50:false",input.mReader(filePath2).get(9));

assertEquals("Goldmine:650:false",input.mReader(filePath2).get(10)
);

   }
```

En automatisk test af tekst indlæsningen viser igen at at de korrekte linjer i filerne indlæses og vil blive indsat på deres korrekte pladser

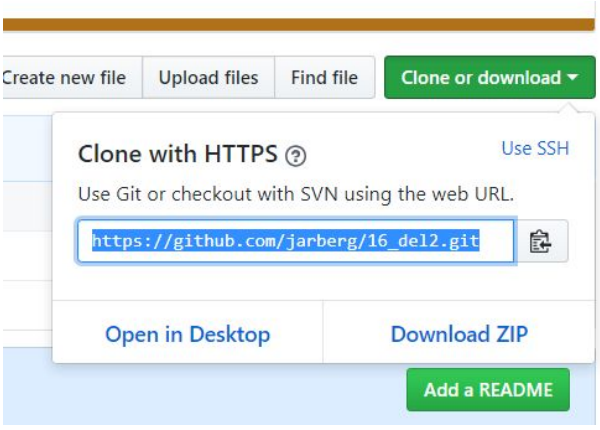
`` @Test void shouldWorkWithFakeDice(){ DiceCup testCup = new DiceCupWithFakeDice(); testCup.roll(); assertEquals(2, testCup.getSum()); } ```	En automatisk test der anvender teststubbe for at sikre sig at raflebægeret virker for sig selv. Teststubbene er 'DiceCupWithFakeDice' som extender DiceCup og indeholder nogle instanser af FakeDie klassen der extender Die men hvor 'getValue' altid returnerer 1.
`` @Test void shouldReturnSumFromTwoToTwelve() { DiceCup testCup = new DiceCup(); int sum = testCup.getSum(); assertTrue(sum < 13 && sum > 1); } ```	Denne test sikrer sig at summen af terningerne når raflebægeret bliver lavet er et tal mellem 2 og 12.
`` @Test void shouldReturnSumFromTwoToTwelveOnRoll() { DiceCup testCup = new DiceCup(); testCup.roll(); int sum = testCup.getSum(); assertTrue(sum < 13 && sum > 1); } ```	Denne test sikrer sig at summing af terningerne når raflebægeret slås med giver 2 og 12
`` @Test void onThirtySixThousandRolls_ShouldBeRandomWithinTwoHundred() { DiceCup testCup = new DiceCup(); int[] resultCounters = new int[13]; for (int i = 0; i <36_000 ; i++) { testCup.roll(); int sum = testCup.getSum(); resultCounters[sum]++; } ```	Denne automatiske test sikrer sig at tilfældigheden af raflebægeret stemmer overens med virkelighedens sandsynligheder. Der er fx. en 1/36 chance for at slå to 1'ere. Vi giver testen en fejlmargin på 200, så hvis der i et kørsel af testen bliver slået fx 950 2'ere ud af de 36 000 slag, siger vi at det er tæt nok på den statistiske virkelighed. Man kan argumentere for at det ikke giver mening at bruge tid på at teste om Java's egen API passer, men det er en god test at vise en kunde.
`` //Using 2 dice probability. fx 2/36 chance of 3. assertTrue(resultCounters[2]>1000 - 200); assertTrue(resultCounters[2]<1000 + 200); assertTrue(resultCounters[3]>2000 - 200); assertTrue(resultCounters[3]<2000 + 200); assertTrue(resultCounters[4]>3000 - 200); assertTrue(resultCounters[4]<3000 + 200); assertTrue(resultCounters[5]>4000 - 200); assertTrue(resultCounters[5]<4000 + 200); assertTrue(resultCounters[6]>5000 - 200); assertTrue(resultCounters[6]<5000 + 200); assertTrue(resultCounters[7]>6000 - 200); assertTrue(resultCounters[7]<6000 + 200); assertTrue(resultCounters[8]>5000 - 200); assertTrue(resultCounters[8]<5000 + 200); assertTrue(resultCounters[9]>4000 - 200); assertTrue(resultCounters[9]<4000 + 200); assertTrue(resultCounters[10]>3000 - 200); assertTrue(resultCounters[10]<3000 + 200); assertTrue(resultCounters[11]>2000 - 200); assertTrue(resultCounters[11]<2000 + 200); assertTrue(resultCounters[12]>1000 - 200); assertTrue(resultCounters[12]<1000 + 200); ```	

Konfiguration

I skal derfor udfærdige en beskrivelse af minimumskrav og en vejledning i hvordan kildekoden compiles, installeres og afvikles.

Som minimumskrav til styresystemet er Windows
Som minimumskrav til redigering af spillet kræves intellij
Som minimumskrav til programsprog kræves java version 10 eller over
Som minimumskrav til unit tests og GUI kræves Maven

For at hente projektet til kundens lokale computer skal de logge ind på deres github profil hvor kildekoden vil være delt med dem fra repositoriet https://github.com/jarberg/16_del2.



alternativt kan projektet hentes manuelt ved at downloade en zipfil fra samme repo.

i Intellij
Vælg fil tabben, hold musen over new og der efter "project from version control" og vælg git hvis brugeren er logget ind på sin gitprofil i intellij vil de have muligheden for at indsætte et link til et repo selv eller vælge via en dropdownliste det ønskede repo.
Herefter vælges den lokale folder som projektet skal opbevares i samme menu
tryk clone når url og filstig er indtastet.

det lokale intellij projekt er nu sat op med spillets git repo

Projektforløb

Under denne opgave har vi gjort et forsøg på at være mere struktureret ift. hvem der laver hvad og hvornår. Vi startede alle sammen ud med at læse opgavebeskrivelsen hjemmefra, og aftalte hvem der skulle lave hvad. Dette var primært i analysen. Vi fik også opstillet en "to-do-liste" over de ting vi manglede i opgaven. Det fungerede egentlig meget godt med analysen, men så gik vores gruppe arbejde lidt i stå. Men vi kom dog tilbage til at være

engageret, og vi fik alle siddet og startet på at kode og kigge analysen igennem sammen. Vores kommunikation kan godt blive bedre især når vi når til at kode og teste. Vores tidsforbrug på opgaven varierer (lidt meget) fra hvert enkelt medlem i gruppen. Alle har på et tidspunkt hjulpet til med opgaven dog skal vi kommunikere bedre næste gang, uddele flere opgaver til hinanden og lave en fast to-do-liste, der er lavet fra start.

Konklusion

Ud fra kundens krav og beskrivelse har vi opfyldt kravene. Vi har først opstillet kravene via. funktionelle -og ikke funktionelle krav, som har hjulpet os med at definere kravene samt danne os et overblik over kravene. Derefter har vi analyseret vores use cases, der har skabt overblik over, hvad spillet skal minimum indeholde. I designfasen har vi ført analysen videre dog har vi opstillet klasser, attributter og metoder, som fungerede som en slags guideline. Vores test har også haft en væsentlige rolle, da det er her vores metoder og spil er blevet testet.
Alt i alt, så fungerer vores program som ønsket, og det viser vores tests. Opstillingen af vores analyse har gjort, at vi har haft en rød tråd og guideline i vores arbejde. Sidst, så har denne opgave været tæt på en succesfuld oplevelse.

Bilag

Fuldt timeregnskab

Rasmus	Analyse	Design	Kode	Dokumentation	Versionsstyring
22/10	1	1		0.5	
23/10	1	1			1
06/11		1	4	0.5	
07/11		2	1		
09/11				1	
I alt	2	5	5	2	1

Jens Daniel	Analyse	Design	Kode	Dokumentation	Versionsstyring
22/10	1/4				
23/10	1			1	
03/11				½	
05/11	½	3½		½	0.1
06/11		1	2	½	2
07/11			5	½	1
08/11		1	8	3	½
09/11				6	
I alt	1.75	5.5	15	12	3.6

Michael	Analyse	Design	Kode	Dokumentation	Versionsstyring
22/10	0.30			0.25	
23/10	1				
07/11		1	2	½	1
08/11		1	7	4	2

09/11			1	2	½
I alt	1.3	2	10	6.75	3.5

Helle	Analyse	Design	Kode	Dokumentation	Versionsstyring
22/10	0.5				
23/10	1				
d. 28/10	1 time				
d. 4/11	1 time				
d. 6/11		0.5	1		
d. 7/11	0,2				
d. 8/11	4.7	0.5	1	3.5	
d. 9/11				3,5	
I alt	4.7	0.5	1	3.5	

Tamanna	Analyse	Design	Kode	Dokumentation	Versionsstyring
23/10	1				
2/11	0,5				
4/11	1				