Projektopgave efterår 2013 - jan 2014

02312-14 Indledende programmering og 02313 Udviklingsmetoder til IT-Systemer

Projektnavn: del2 Gruppe nr: 19 Afleveringsfrist: mandag den 11/11 2013 Kl. 5:00

Denne rapport er afleveret via Campusnet (der skrives ikke under)

Denne rapport indeholder 57 sider incl. denne side

Studie nr, Efternavn, Fornavne

s110795, Mortensen, Thomas Martin
Kontakt person (Projektleder)

s113577, Johansen, Chris Dons

s123897, Ahlgreen, Thomas Kamper

Timeregnskab

Dato	Deltager	Design	Implementering	Test	Dok.	Andet	I alt
18/10-13	Thomas Mortensen	2					2
18/10-13	Chris Johansen	2					2
18/10-13	Thomas Ahlgreen	2					2
25/10-13	Thomas Mortensen		2,5				2,5
25/10-13	Chris Johansen		2,5				2,5
25/10-13	Thomas Ahlgren		2,5				2,5
1/11-13	Thomas Mortensen		2,5				2,5
1/11-13	Chris Johansen		2,5				2,5
1/11-13	Thomas Ahlgren		2,5				2,5
8/11-13	Thomas Mortensen			1	6		7
8/11-13	Chris Johansen			1	6		7
8/11-13	Thomas Ahlgren			1	6		7
diverse	Thomas Mortensen				4		
diverse	Chris Johansen					2	
diverse	Thomas Ahlgren						
							48

Indhold

1 Indledning

Spilfirmaet IOOuterActive har givet os endnu en opgave.

Kundens vision er en udvidelse af vores allerede udviklede programmer fra [**del1**] og [**del2**]. Denne gang ønsker kunden at lave flere forskellige typer felter. Disse felter skal implementeres på en "rigtig"spilleplade, hvor man kan gå i ring. Samtidig ønskes der mulighed for 2-6 spillere. Spillerens konto skal sættes med et fast beløb fra start, og spillet skal slutte, når en spiller er gået bankerot.

Projektet forventes at have elementer fra både **FURPS+** og **GRASP**. Herudover er der lavet specifikke krav til hvilke artefakter, der skal ingå i krav, analyse, kode, test, og designdokumentation.

OBS: alle gruppemedlemmer er lige ansvarlige for alle dele af vores dokumentation

2 Kravspecificering og Use cases

I dette afsnit vil vi beskrive vores kravspecificering og vores Use cases, som vi har udarbejdet til denne rapport.

2.1 Kravspecificering

I dette afsnit har vi taget udgangspunkt i kundens vision. Vi har læst den igennen, og stillet spørgsmålstegn, ved de ting, der kunne være tvivl om. Kunden er dog blevet til kravspecificering i forløbet, og derfor var der få tvivlsspørgsmål.

1. Hvad skal der ske, hvis en spiller ikke kan betale det han skal?
2. Skal man betale, hvis man lander på et felt, der er ejet af en bankerot spiller?
3. Skal man have mulighed for at købe eet felt, der tidliger var ejet af en spiller, der nu er gået bankerot?

Vores kontakt i firmaet, som er vores projektleder, har besvaret disse punkter på følgende måde

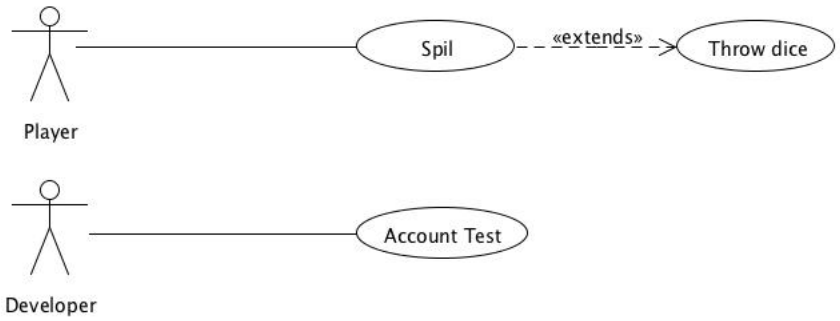
1. Spilleren der ikke kan betale, må betale det han har, og går herefter bankerot.
2. Feltet er ikke længere ejet af nogen, og derfor skal der heller ikke betales.
3. Da feltet ikke længere ejes, skal man kunne købe det.

2.2 Use Cases

Thomas A: Indsæt tekst fra fully dresses og brief + diagram

Vi vil i dette afsnit beskæftige os med 2 use cases, som er beskrevet i nedenstående diagram. Til vores spil har vi lavet en fully dressed use case, og til vores test har vi blot lavet en brief use case.

2.2.1 Use Case Diagram



Figur 1: Use Case Diagram

2.2.2 Fully Dressed Use Case

Vi har i dette afsnit beskrevet vores use case med en fully dressed use case for spil.

Use Case

Spil

Scope

Terningspil

Level

User Goal

Primary actor

Spiller

Stakeholders and Interests

Spiller: vil vinde spillet

Preconditions

Eclipse er installeret på maskinen.
Programmet er startet.

Main Success Scenarie:

1. Spiller starter spillet
2. Spiller indtaster Navn
3. Spiller Ruller med terningerne
4. Spiller lander på et felt med positiv konsekvens
5. Spiller modtager point – Spiller gentager punkt 3-5 indtil 3000 eller mere er opnået
6. System checker vinder
7. System viser vinder
8. System lukker ned

Extensions Alternative scenarier:

*a - til hvert et tidspunkt

1. Spiller lukker spillet ved at taste "q"

4a. Spiller lander på et felt med negativ konsvens

1. Spiller får trukket point
2. Spiller ryger under 0 point – punkt 6-8 i main success aktiveres

4b. Spiller lander på et felt uden konsekvens

1. Spiller for trukket point
2. a Spiller ryger under 0 point – punkt 6-8 i main success aktiveres
3. b Spiller rammer felt 10. uden at komme under 0, hviket giver ekstra tur. – punkt 3 i main success aktiveres

Specielle Krav:

- Skal kunne køre på en Windows maskine med Java EE på DTU's computere
- Skal kunne spilles af en almindelig bruger

2.2.3 Brief Use Case Account test

Vi har valgt at lave en brief use case over vores test.

Account Test En udvikler tester klassen med forskellige grænse og middelværdier, der er foruddefineret i klassen. Resultaterne printes ud i konsollen, og udvikleren ser om resultaterne fra testen, stemmer overens med de forventede resultater.

3 FURPS+

Thomas M: forklar furps, og hvordan vi har brugt det.

I **UP** bruger man **FURPS+**, der er udviklet af Hewlett-Packard til at kategoriserer kravene til ens system. "+"i **FURPS+** kom i følge [Wik13] til senere, efter HP ønskede at dække flere kategorier med denne model. Vi har beskrevet kort, hvad der generelt hører under de forskellige punkter i **FURPS+**, og efterfølgende listet de funde krav i den aktuelle opgave op.

3.1 Functional

Hvilke funktioner skal systemt have. Hvad skal det kunne. Sikkerhed.

Vi har taget udgangspunkt i kundens [IOO13] opgave. Ud fra deres vision, bilag og andre punkter i rapporten har vi fundet frem til følgende punkter:

- En udbyggelse af forrige system med forskellige felttyper.
- En spilleplade, hvor spillerne skal kunne lande på et felt og fortsætte på næste slag. Samtidig skal man kunne gå i ring på brættet.
- Man skal kunne vælge mellem 2-6 spillere.
- Et *Territory* skal kunne købes af en spiller, når han lander på feltet. Feltet skalhave en fast leje. Jo højere feltnummer jo højere pris og leje.
- Et *Refuge* skal give en bonus på enten 5000 eller 500 afhængigt af feltnummeret, når en spiller lander på denne type felt.
- Når man lander på et *Labor Camp* felt, skal man betale 100 gange værdien af et nyt terningeslag. Dette beløb skal i øvrigt ganges med antallet af*Labor Camps* med den samme ejer.
- Der er to felter af typen *Tax*. Det ene felt betaler man et fast beløb af $Kr.2000,-$, når man lander på feltet. Det andet skal man selv vælge om man vil betale et fast beløb af $Kr.4000,-$ eller om man vil betale 10% af hele sin formue. (Hvilket betyder, både af hvad der står på kontoen og af felter man ejer).
- Lander man på et felt af typen *Fleet*, bestemmes beløbets størrelse ud fra hvor mange *Fleet* felter, der har samme ejer.

Sikkerhed er et punkt, vi ikke kan finde noget om, så vi må gå ud fra firmaet selv sørger for dette.

3.2 Usability

Menneskelige faktorer, skal man bruge hjælp. Dokumentation af systemet i brug.

Vi går ud fra, at kravene til **Usability** er de samme som i foregående opgaver. Det er kun dokumentation, der er nævnt i denne opgave. Kravene fra sidst, var at det skulle kunne bruges af en normal person med en 9. klasses eksamen. Der må ikke være brug for hjælp. Dog må man godt forklare hvad spillet går ud på.

Til dokumentation forventes der **Designsekvensdiagrammer**, der viser den dynamiske tilgang til programmet.

3.3 Reliability

Hyppigheden af fejl, kan det nemt gendannes og forudsigelighed.

For at formindske hyppigheden af fejl, har vi løbende brugt User Test, for at fange semantiske fejl. Herudover har IOOuterActive stillet en skabelon til**Junit**

tests, der gerne skulle fange fejl i vores `Field` klasser. Derudover er der ikke nogle målbare krav til **Reliability**.

3.4 Performance

response times, throughput, accuracy, availability, resource

3.5 Supportability

daptability, maintainability, internationalization, configurability.

Her kommer de underfaktorer, som +'et repræsenterer

3.6 Implementation

resource limitations, languages and tools, hardware,

3.7 Interface

constraints imposed by interfacing with external systems.

3.8 Operations

system management in its operational setting

3.9 Packaging

How to deliver the system? Physical box or file... How many installations are there?

3.10 Legal

licensing and so forth.

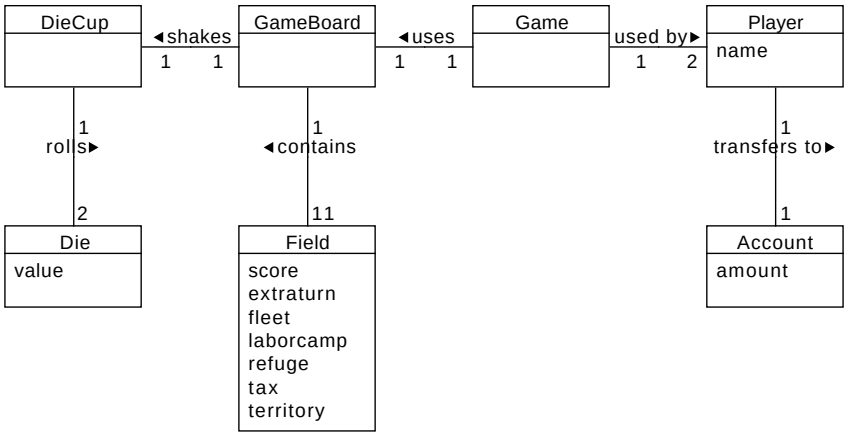
4 Domænemodel

Thomas M: indsæt diagram og forklar

Da vi har kunnet bruge en stor del af vores domænemodel fra sidst, har vi valgt ikke at lave en navneordsanalyse, da vi ville bruge for lang tid på at lave den, i forhold til udbyttet af denne. Dette er årsagen til, at vi starter med domænemodellen.

Vores domæne model er lavet før begyndelse af implementering. Det er udført forinden for at få et overblik over hvilke elementer, vi kunne have behov for i vores program.

Vi har i 2 vist vores hoved domæner. *Game*, som er selve spillet. *Gameboard*, der holder styr på indholdet i spillet. *Field*, som indeholder en score for feltet, og om det giver ekstra tur. *Player*, der bruger spillet. *Account* holder styr på point. *DieCup*, som skal rulle vores terninger. Til sidst har vi *Die* som indeholder terningeværdier, vi skal bruge til at udregne, hvor spilleren lander.



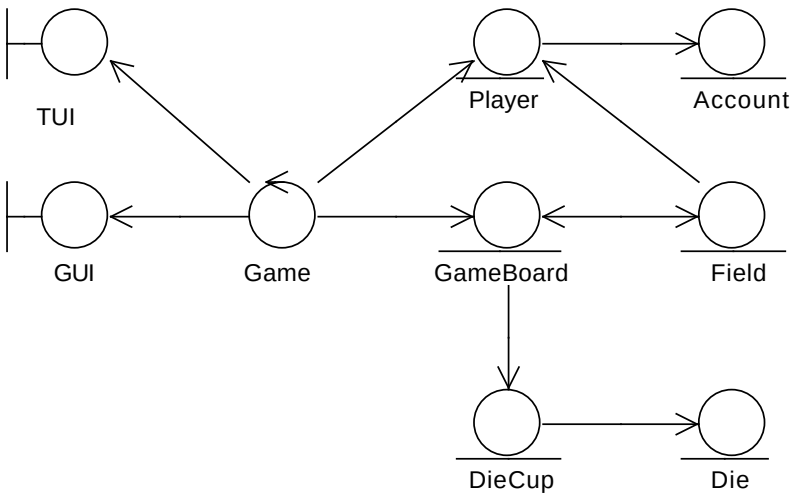
Figur 2: Domæne Model

5 BCE model

Thomas M: indsæt model og forklar

I denne CDIO opgave, har vi igen benyttet os af en BCE model til at skabe oveblik over vores kendskab i koden.
Vi har denne gang ændret
Det vil sige at vi i denne model kun har tilføjet vores nye entitetsklasser. Det handler om entiteterne *Account*, *Field*, og *GameBoard*. Vi har også lavet en ny boundary vi kalder *Graphic*, som bearbejder beskeder til vores importerede *GUI*.
Som det fremgår af diagrammet, styrer controlleren *Game* stadig spillet. Det vil sige at vores controller tildeler ansvaret ud til de nære klasser.
Account er en pengebeholdning der styres af *Player* klassen. Vores controller har derfor kun brug for direkte kendskab til *Player*.

Field holder styr på felterne i spillet, og vores *Gameboard* holder styr på felterne. Vores *Game* controller har dermed kendskab til felterne i spillet gennem vores *Gameboard*.



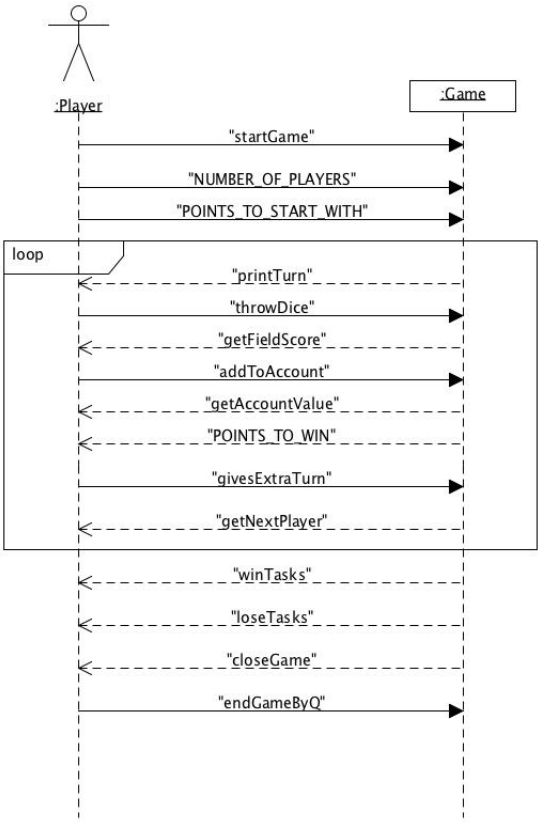
Figur 3: BCE Model

Ud fra disse informationer kan controlleren så sende besked til vores boundary's *TUI* og *Graphic*, som viser brugeren konsekvenserne i spillet grafisk og i tekst. Dette er hvad 3 beskriver

6 Systemsekvens Diagram

Thomas A

Vores System sekvens diagram tager udgangspunkt i vores fully dressed use case: spil. Det første der sker, er at selve spillet startes. Herefter finder vi ud af hvor mange spillere, der er. Spillernes startbalance bliver nu sat. Herefter vil spillet kører i en løkke, indtil en af spillerne har tabt eller vundet. Løkken printer turen på spilleren. Terningen kastes. Spilleren får konsekvensen af det felt han lander på. Beløbet bliver overført til spillerens konto. Spilleren får nu vist hvad balancen er på kontoen. Der bliver kontrolleret, hvor vidt spilleren har opnået point nok til at vinde eller har mistet alle sine penge (Her mangler en metode til at kontrollere balancen på spillerens konto). Hvis det er tilfældet hopper vi ud af løkken. Hvis det er et bestemt felt på pladen, skal spillet give spilleren en ekstratur (Pilen vender den forkerte vej i diagrammet). Dette vil give ham en tur mere i løkken. Hvis ingen af disse ting forekommer, giver spillet turen videre. Hvis en spiller har vundet, skriver konsollen, hvilken spiller det er og hans point. Hvis en spiller har tabt, skriver konsollen, hvilken spiller der så har vundet og hans point. Efter det lukker spillet. Spilleren har hele tiden mulighed



Figur 4: Systemsekvens Diagram

for at stoppe spillet med tasten "q".
Systemsekvensdiagrammet er vigtigt at have med, da det vise hvordan systemet reagere på udefrakommende inputs. Det giver også indblik i hvordan systemet skal fungere uden at skulle skrive en masse kode.

7 Kode

Chris: skriv implementering af kode.

7.1 Struktur og pakker

Ligesom i CDIO del1 projektet, er programmet skrevet med fokus på at overholde BCE-modellen så meget som overhovedet muligt. Derfor er programmet også i dette projekt opdelt i pakker med navne svarende til BCE modellen, dvs.

TUI og Graphic i "Boundary", Game og Main i "Controller" og Account, Die, DieCup, Field, GameBoard og Player i "Entity". Med andre ord er klasserne opdelt i pakker efter hvilken type de er.

Undtagelsen her er de forskellige test-klasser, som er lagt ned i en pakke, "TestTools", for sig selv. Dette er gjort ud fra en ide om at det vil være udviklere der benytter dem, og at de typisk nemmest vil kunne benyttes ved at omdøbe og indsætte dem direkte på andre klassers pladser. Således er der ikke nogen direkte henvisninger til dem i koden til det egentlige program, som almindelige brugere vil opleve.

Herunder en gennemgang af de forskellige klasser og deres funktion.

7.2 TUI

Denne klasse håndterer alt hvad der kommer ind og ud af konsollen. For en grundigere beskrivelse af denne klasses funktionalitet henvises til det tilsvarende afsnit i CDIO del1 rapporten (7.0.4, side 12). Nogle metoder har andre navne og udskriver selvfølelig noget andet tekst, men den overordnede virkemåde er fuldstændigt identisk.

7.3 Graphic

Denne klasse styrer alt hvad der skal ændres på den grafiske brugerflade. Der kan argumenteres for, at denne klasse burde navngives "GUI" for at overholde mønsteret fra tekst brugerfladen, men dette ville besværliggøre udviklingen, fordi det bibliotek der bliver stillet til rådighed til brugerfladen også bruger denne navngivning. Således vælger vi blot at kalde klassen "Graphic", med henvisning til at den styrer den grafiske del af programmet.
Nogle metoder kalder blot de tilsvarende metoder i biblioteket direkte (setDice, addPlayer, close), men vi vælger at kalde dem gennem Graphic for at have henvisningerne til GUI-biblioteket samlet det samme sted. Denne fremgangsmåde gør det langt nemmere at vedligeholde programmet i tilfælde af ændringer/opdateringer af det bibliotek der benyttes.
Graphic indeholder desuden en "moveCar"-metode, som blot kalder to metoder i GUI-biblioteket – en til at fjerne brugerens bil(er) fra spillepladen, og en til at sætte bilen på et nyt felt.

Den interessante del af Graphic-klassen er imidlertid "setupFields"-metoden. Her udføres en række kald til hjælpemetoden "createField", der igen kalder metoder i GUI-biblioteket til at ændre titel, undertitel og beskrivelse af et felt. De felter, som benyttes til dette spil (2 til 12) opsættes med en liste af kald, som indeholder informationer om navn og score for hvert felt. Der kan argumenteres for, at opsætningen af felter på GUI'en kunne genbruge navne fra TUI'en og

felt scoreværdier fra GameBoard, men vi vælger undlade at gøre dette, simpelthen for at holde koblingen så lav som muligt, og for at gøre programmet mere fleksibelt ift. ændringer. F.eks. kunne man måske forestille sig at en kunde kun ville betale for at få oversat enten TUI eller GUI i programmet, hvilket ikke ville være muligt hvis de hænger sammen, eller måske mere sandsynligt, at kunden var interesseret i at modificere programmet til kun at have en GUI, og derved fjerne TUI-klasse. Uanset hvad vil det i sådanne tilfælde være en fordel, at de forskellige klasser ikke er koblet for meget sammen, og det vurderer vi er vigtigere, end at spare et par linjers kode ved at kunne nøjes med at skrive felternes navne et enkelt sted.

7.4 Account

Denne klasse repræsenterer en konto, med en mængde penge/score. Klassen er ganske simpel, i den forstand at den kun indeholder en enkelt attribut med tilhørende get og set metoder, samt en metode der kan tilføje til den eksisterende score.

Det specielle ved Account-klassen er, at den sikrer at en Account-balance ikke kan blive negativ, og giver en tilbagemelding på om en transaktion er gennemført eller ej.
Klassen indeholder både en konstruktør som tager en "oprettelses-balance" (initialAccountValue), samt en konstruktør som ikke tager nogen argumenter, og så blot opretter kontoen med en værdi på 0.

7.5 Die

Denne klasse er identisk med den Die-klasse der blev benyttet i CDIO del1. For en beskrivelse af klassen henvises til det tilsvarende afsnit i CDIO del1 rapporten (7.0.6, side 13).

7.6 DieCup:

Denne klasse læner sig kraftigt op ad den DieCup-klasse der blev benyttet i CDIO del1, der er blot fjernet noget overflødig funktionalitet i den udgave der er benyttet i dette projekt. For en beskrivelse af funktionaliteten i denne klasse henvises derfor til det tilsvarende afsnit i CDIO del1 rapporten (7.0.7, side 13).

7.7 Field

Denne klasse er skrevet til at bære de relevante informationer om et felt på spillepladen. Navn og beskrivelse er lagt i brugerfladeklasserne, for at gøre det nemmere at oversætte, så det eneste Field-klassen skal indeholde, er den effekt feltet skal have på en spiller – dvs. hvilken score feltet giver, og om feltet skal

give en ekstra tur.

Hvad et felt skal gøre, kan ikke ændre sig i løbet af et spil, så der er ingen set-metoder til attributterne – de sættes med konstruktøren, og ændrer sig ikke efterfølgende. Dvs. klassen har en konstruktør som tager et heltal for hvilken score feltet giver, samt en boolsk værdi for om feltet giver en ekstra tur. De fleste felter giver ikke en ekstra tur, så klassen indeholder også en konstruktør som kun tager et heltal for score, og så blot sætter værdien for ekstra tur til falsk. På den måde kan der spares lidt kode ved oprettelsen af alle felterne.

7.8 GameBoard

Denne klasse repræsenterer en spilleplade i spillet. Den indeholder blot et array med alle de felter som skal bruges, samt en get-metode til at få fat i et felt på listen. Felterne oprettes og tildeles deres værdier i konstruktøren, og ændres ikke efterfølgende.

Der kan argumenteres for, at listen med felter fint kunne ligge i Game-controlleren, men for at få højst mulig sporbarhed ift. et "rigtigt" spil, hvor der naturligvis vil være en spilleplade, vælger vi at lave en klasse til at indeholde felterne. Som en bonus fjerner denne løsning også noget kompleksitet fra controlleren i forhold til oprettelse af felterne.

7.9 Player

Denne klasse indeholder data om en spiller. I dette spil er der kun behov for at lagre en score og et navn, men ved at benytte en struktur med en decideret klasse til at indeholde spillerdata, er programmet forberedt til at gemme andre informationer om en spiller – f.eks. spillerens position på spillepladen, eller hvad der blev slået sidst osv..

Scoren for en spiller gemmes i dette program i en klasse til formålet – Account. Når et nyt objekt af Player-klassen oprettes, laves der således også blot en ny Account, som så holder styr på spillerens score. Se evt. beskrivelse af Account-klassen tidligere i denne rapport.

7.10 Main

Main-klassen benyttes kun til at oprette og starte Game-kontrolleren, og indeholder således hverken logik, metoder eller data.

7.11 Game

Dette er klassen som indeholder selve logikken i spillet, og her vil derfor være en del ting der er interessante at nævne omkring beslutninger ift. implemente-

ringen.

Først og fremmest er det værd at bemærke, at spillerne i programmet lagres i et array af spillere. Det betyder dels at det bliver nemmere at lave en generisk løsning på logikken for en spillers tur, men dels også at det er nemt at udvide spillet til flere spillere. Faktisk er antallet af spillere angivet som en konstant i begyndelsen af klassen, og hele programmet er skrevet med tanke på, at det skal kunne virke med mere end 2 spillere – på den måde er programmet godt forberedt til udvidelse.

Et eksempel på dette er oprettelsen af nye player-objekter, som bliver kørt i Game-konstruktøren. Objekterne oprettes i en løkke, som stoppes når antallet

```
// Make all player-objekts in loop
for (i = 0; i < NUMBER_OF_PLAYERS; i++) {
    players[i] = new Player(POINTS_TO_START_WITH);
}
```

Figur 5: Spillere oprettes i en løkke

af oprettede spillere nå op på det antal, som er angivet i konstanten i starten af programmet.

Foruden player-objekterne oprettes i konstruktøren også diverse andre objekter af klasser, som bruges i kontrolleren – dieCup, GameBoard samt en scanner. Desuden åbnes GUI'en fra konstruktøren, og felterne på GUI'en indstilles til noget der er passende for spillet.

Den øvrige logik ligger i "startGame"-metoden, som også kaldes direkte fra main. TUI'en kaldes for at udskrive spillets regler, og herefter bedes alle spillere indtaste deres navn. Igen er der tale om en fleksibel løsning, som vil virke med flere end 2 spillere. Igen foregår det selvfølgelig i en løkke, og igen kører løkken

```
// Ask for all player names and save them in the player objects. Also
// adds the players to the GUI.
for (i = 0; i < NUMBER_OF_PLAYERS; i++) {
    TUI.printNameRequest(i);
    players[i].setName(TUI.getUserInput(scanner));
    Graphic.addPlayer(players[i].getName(), players[i].getAccount()
            .getAccountValue());
}
```

Figur 6: Indlæs navne i en løkke

bare indtil der er kørt lige så mange gange som der er angivet, at der skulle være spillere. Inde i løkken udskrives så først en lille tekst, som beder brugens indtaste et navn, derefter indsamles det indtastede fra konsollen, hvorefter det gemmes i det player-objekt som passer til spilleren, og til sidste opdateres GUI'en med den nye information.

Herefter starter selve logikken for spillernes "ture" i spillet. Turene køres i en endeløs løkke ("while(true)"), som så kan afbrydes ved forskellige scenarier.

Denne løsning er valgt, dels fordi der er mange forskellige ting der kan afbryde spillet (en spiller trykker ”q”, en spiller vinder, en spiller taber), dels fordi vi er interesseret i at kunne afbryde spillet midt i en tur. F.eks. giver det ikke meget mening at der køres en hel tur færdig med score osv., hvis en spiller har trykket ”q” for at afslutte spillet, og det giver heller ikke meget mening at udskrive endnu en status i spillet, og tjekke for om en af spillerne har vundet, hvis der allerede er en spiller som har tabt osv..

Selve tur-logikken starter med at udskrive navnet på den spiller, som skal slå. Herefter ventes på input fra konsollen. Når der er givet et input, tjekkes om inputtet er ”q” – er det det, afsluttes spillet. Ellers fortsættes, og der slås med terningerne. Ud fra værdien af terningerne, findes så det felt som spilleren er landet på, og scoren for det pågældende felt hentes. Denne score tilføjes så spillerens player-objekt, og der tjekkes om transaktionen er gennemført. Er den ikke det, må det skyldes at balancen på spillerens konto er blevet negativ, så spillet afsluttes og den aktuelle spiller erklæres som taber af spillet. Bliver transaktionen gennemført fortættes spillet, og der udskrives en status. Herefter tjekkes om spilleren har opnået det ønskede antal point for at vinde – har han det, afsluttes spillet og spilleren erklæres som vinder. Har han ikke det, fortsættes spillet, og der tjekkes om det felt, spilleren er landet på, giver en ekstra tur. Gør det det, køres en ny tur med den samme spiller, gør det ikke det, skiftes til den næste spiller i rækkefølgen, før der køres en ny tur.

For hver af alle disse handlinger (afslut spillet, erklær en spiller som taber, erklær en spiller som vinder, udskriv status, skift til næste spiller i rækkefølgen), findes en hjælpemetode, som udfører de relevante operationer for en handling.

Når spillet afsluttes udføres en oprydning af de benyttede komponenter. GUI’en lukkes, scanneren, som benyttes til at hente input fra konsollen, lukkes,

```
private void cleanUp() {
    Graphic.close();
    scanner.close();
    System.exit(0);
}
```

Figur 7: Metode til oprydning

og til sidst afsluttes programmet.

Når en spiller erklæres som taber, udskrives navnet og scoren for taberen, og der ventes på konsolinput, for at sikre at spillerne når at se beskeden, inden spillet lukker. Når der gives et input, lukkes spillet med cleanUp metoden ovenfor. Præcist det samme gør sig gældende når en spiller vinder, bortset selvfølgelig fra at teksten som udskrives er anderledes.

Når der udskrives status, foregår det ligeledes med en hjælpemetode. Herfra opdateres både GUI og TUI. Der udskrives en status til TUI’en terningernes

```
private void loseTasks(int activePlayer) {
    TUI.printLoser(players[activePlayer].getName(), players[activePlayer]
            .getAccount().getAccountValue());
    TUI.getUserInput(scanner);
    cleanUp();
}
```

Figur 8: Metode for tabt spil

```
private void statusTasks(int activePlayer) {
    TUI.printStatus(players, dieCup.getSum());
    Graphic.setDice(dieCup.getValueDie1(), dieCup.getValueDie2());
    Graphic.updatePlayers(players);
    Graphic.moveCar(players[activePlayer].getName(), dieCup.getSum());
}
```

Figur 9: Metode til opdatering af TUI og GUI

værdi, samt score for alle spillere. Derudover indstilles terningerne på GUI'en, ligesom spillerens score opdateres og den aktuelle spillers placering på spillepladen ændres.

På samme måde er der også en metode til at skifte til den næste bruger i rækkefølgen. Igen er det her værd at bemærke, at metoden også vil virke med

```
private int getNextPlayer(int input) {
    if (input + 1 >= NUMBER_OF_PLAYERS) {
        return 0;
    }

    return input + 1;
}
```

Figur 10: Metode til næste spiller

mere end 2 spillere. Metoden tager nummeret på den nuværende spiller som input, og lægger en til – med mindre den spiller der i så fald skulle returneres ville have et højere nummer, end den konstant der er angivet for antallet af spillere – i så fald returneres blot 0 (dvs. der startes forfra i rækkefølgen).

8 Test

Vi blev bedt om at lave en J-unit test af vores felttypers `landOnField` metoder. Disse vil være beskrevet i dette afsnit.

j-unit test, snydeterninger og bruger test Thomas M

8.1 Test af Refuge

Sammen med opgaven fik vi et bilag, der beskrev hvordan testen af `Refuge` skulle laves.

Til formålet oprettes en testklasse `RefugeTest`. I denne klasse tog vi indholdet fra bilagene, og modificerede det lidt. For at få koden til at passe til vores

program ændrede vi nogle småting. Da vi har benyttet os af **BCE** notationen til vores pakker, måtte vi importere vores `Player` til at oprette objekter igennem `intity` pakken. Dvs. hvis kontoen sættes til 1, bliver transaktionen gennemført

```
Set to 1 = true
Set to 0 = true
Set to -1 = false
Set to 1000000000 = true
Set to -1000000000 = false
Set to 500 = true
Set to -500 = false

Set to 1000, add -1000 = true
Set to 1000, add -1001 = false
Set to 1001, add -1000 = true
Set to 1000, add 500 = true
Set to 1000, add -500 = true
```

Figur 11: Resultatet af testkørsel

(metoden returnerer "true"). Hvis kontoen sættes til 0, bliver transaktionen ligeledes gennemført osv.

Det interessante her er især grænseværdierne, dvs. netop 1, 0 og -1. Testen viser, at klassen opfører sig som vi ville forvente – en positiv værdi giver selvfølgelig true, men mere interessant, 0 giver også true. Ligeledes som forventet, returneres false, hvis balancen sættes til -1.

Når grænseværdierne virker som forventet, vil alle værdier typisk gøre det, men for en sikkerheds skyld testes også lige med nogle meget store værdier, og med nogle helt almindelige værdier. Disse test giver også det forventede resultat.

Herefter udføres en test hvor balancen først sættes til noget kendt, og derefter opdateres med "addToAccount". Ved den første test sættes balancen til 1000, hvorefter tilføjes -1000 – således må det forventes at den resulterende balance bliver 0. Dermed vil vi forvente at tranaktionen bliver fuldført, idet 0 ikke er negativt, og dermed er en godkendt værdi. Vi ser, at også denne test giver det forventede resultat.

Sæt til 1000 og tilføj -1001 må give -1, og dermed false – OK. Sæt til 1001 og tilføj -1000 må give 1, og dermed true – OK.

De mere almindelige værdier giver ligeledes det forventede resultat. Således må det være fair at sige, at det er sandsynligt at en transaktion kun bliver gennemført, hvis ikke den resulterer i en negativ balance.

8.2 Test og fejlfinding generelt

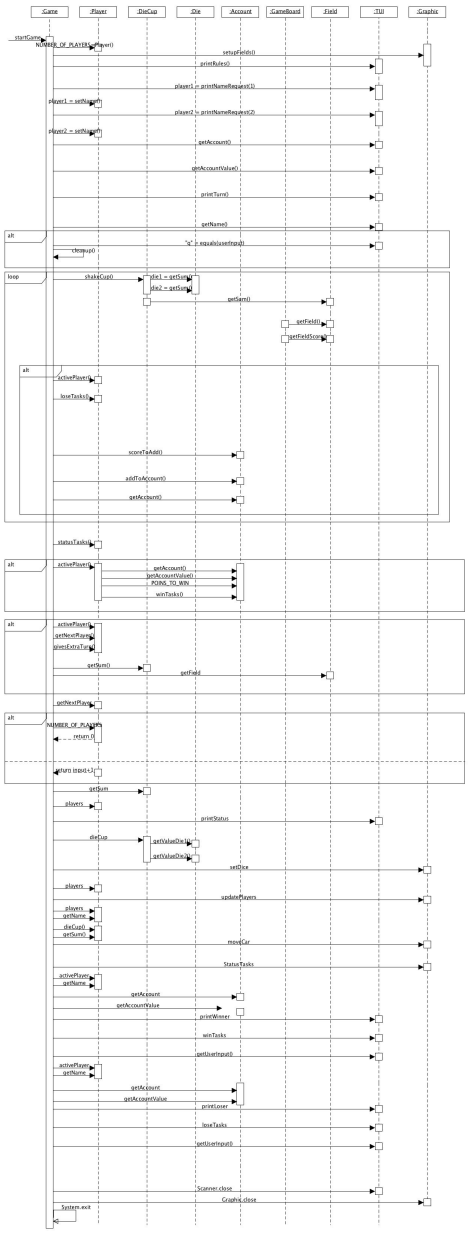
Foruden klassen til blackbox-test af Account-klassen, er der udviklet to testklasser, som kan bruges til fejlfinding og test i programmet. Tanken er, at en eller begge test-klasser kan erstatte de "rigtige" klasser, og på den måde give en

udvikler mulighed for at teste scenarier, som er svære og/eller tidskrævende at opnå ved almindeligt spil.

Der er dels tale om en GameBoard test-klasse, hvor værdierne for felterne alle er negative, således at sikringen af Account-klassen hurtigt kan testes i praksis. Derudover er der tale om en DieCup test-klasse, hvor en udvikler selv har mulighed for at sætte en fast værdi for hvad terningerne skal slå. Således kan der opnås at lande på f.eks. felt nr. 3 (som giver -200) mange gange i træk, og derved teste sikringen af Account-klassen, eller der kan landes på felt 12 (som giver +650) flere gange i træk, og derved teste de handlinger som udføres når en spiller vinder

9 Design sekvens-diagram

Chris: tegn diagram og forklar



Figur 12: Design Sekvens Diagram

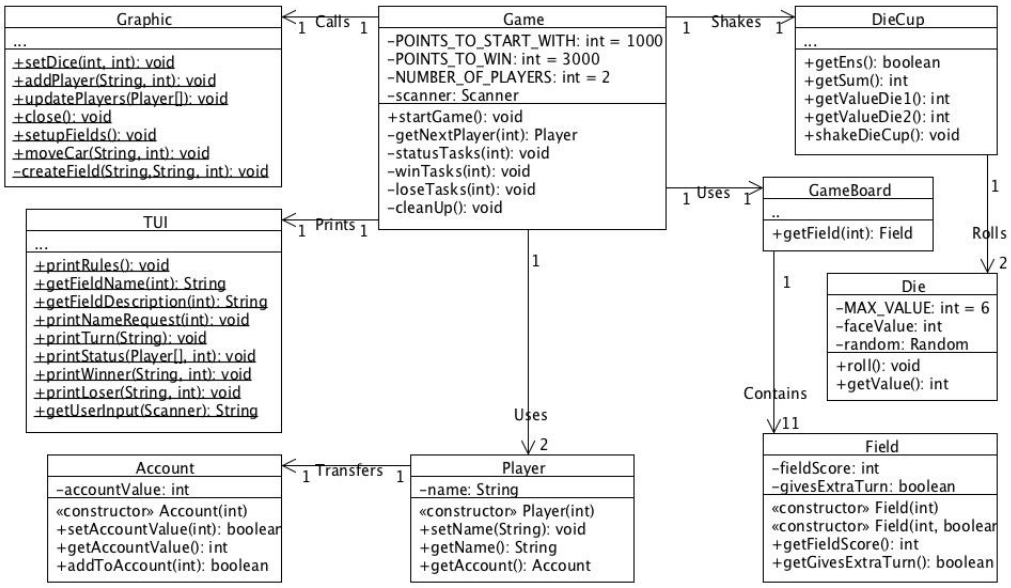
Vi har, som ved første spil, lavet et design sekvens-diagram, som dokumenterer, hvordan koden bliver udført sekventielt, når man spiller det nye feltspil. Hvis man sammenligner vores sekvensdiagram med vores kode afsnit, vil man

se at *Game* controlleren har det primære ansvar for at fordele opgaverne rundt til de ande dele af programmet. Her kan vi bare se det ud fra et tidsmæssigt perspektiv istedet. Hvordan de forskellige metodekald virker, kan man nærstudere i vores afsnit om selve koden.

Desværre er diagrammet fyldt med forkert syntaks og variabler, der er brugt som metodekald. Da dette er noget af det sidste vi har lavet i vores dokumentation, har tidsgrænsen bevirket, vi ikke har kunnet afhjælpe disse fejl inden for den ønskede tidsramme.
Vi vil dog anbefale, at man får dette rettet hurtigst muligt, hvis der er planer om eventuelle opdateringer til spillet, eller genbrug af dele af systemet.

10 Design-klassediagram

Thomas A: lav diagram og forklar



Figur 13: Design-Klassediagram

Vi har også valgt at dokumentere vores program med et klassediagram. Dette giver et godt overblik over programmets opbygning, og om vi har overholdt tankegangen for **GRASP**. Denne tankegang vil blive beskrevet i næste afsnit.

11 GRASP (General Responsibilty Assignment Software Patterns)

Thomas A: Forklar hvad vi har brugt

Vi vil i dette afsnit beskrive de forskellige GRASP patterns og give eksempler på, hvordan vi har brugt det til implementering af vores program [Lar04].

11.1 Controller

Beskrivelse af en Controller

Problem

Hvilket objet under UI laget kontrollerer system operationer

System operationer

System operationer støder vi første gang på under analysen af **SSD** *(System Sekvens Diagram)*. Dette er de vigtige hændelser i vores system. For eksempel, når en spiller i vores spil trykker enter for at rulle med terningerne. Her starter han en system hændelse, der giver et terningeslag.
En **Controller** er det første object under vores **UI** *User Interface*, der har ansvar for at modtage eller løse system operations beskeder.

Generel Løsning

Tildel ansvaret til en klasse, som benytter en af følgende to valgmuligheder.

1. Klassen repræsenterer hovedsystemet med en slags "rod object". En enhed, sofwaren kører inden i systemet, eller et decideret subsystem - Dette er variationer af en *Facade Controller*.

2. Klassen repræsenterer et use case scenarie hvor denne system handling ofte fremkommer. Denne vil tit være kaldet <UseCaseName>Handler, <UseCaseName>Coordinator eller <UseCaseName>Session.

Vores Løsning

Vi har lavet en **Controller** kaldet *GameController* til at styre vores sekvenser i spillet efter princippet med en *Facade Controller*.
Det giver meget god mening at have en klasse, der uddelegerer ansvar. Dette gør at *GameController* kun skal kontrollere og kordinere opgaver til andre objekter. Samtidig skal *GameController* ikke udføre meget arbejde selv. Dette opfylder den guideline, der findes i *Larman*

11.2 Creater

Beskrivelse af en Creator

Problem

Oprettelse af objekter, er en af de mest almindelige aktiviteter i et objektorienteret system.
Et generelt princip til oprettelses ansvar er meget brugbart. Hvis ansvarsfordelingen bliver fordelt godt, kan man opnå **lav kobling** *(Andet GRAS Pattern, der beskrives senere)*, større klarhed, indkapsling (den internne representation af et objekt, der er gemt fra kig udenfor objektets definition.) og genanvendelighed.

Generel Løsning

Giv klasse B ansvar for at oprette en instans af klassen A, hvis et af disse udtryk er sande (Helst flere af udtrykkenne).

- B indeholder, eller komposit agrigerer A
- B bruger A tæt.
- Bhar de initialiserende data til A, som bliver sendt til A, når denne er oprettet.
  Dermed er B **Expert** *(Andet GRAS Pattern, der beskrives senere)* for at oprette A.

Vores Løsning

Denne tankegang har vi brugt i forbindelse med vores domænemodel. I den kan vi udlede at vores *Game* bruger et *GameBoard*, som indeholder flere *Fields*. Vi kan dermed se at *Game* er en god kandidat til at bære ansvaret for at oprette *GameBoard*. *GameBoard* er også oplagt til at oprette *Field* objekter. Samme historie gentages i resten af modellen, så godt som muligt efter denne tankegang.

11.3 Expert

Beskrivelse af en Expert

Problem

Hvad er den gennerelle fremgangsmåde for at tildele ansvar til objekter.

Generel Løsning

Tildel ansvar til informations eksperten. Dette er klassen, der har de nødvendige informationer til at fuldfylde det ansvar.

Vores Løsning

Hvis vi kigger på eksempelvis vores *DieCup*, har den ansvaret for at kende *Die* terningeslag. Dermed er *Diecup* expert i at få terningeslag.

11.4 High Cohesion (Høj binding)

Beskrivelse af High Cohesion

Problem

Hvordan holder man objekter fokuserede, forståelige og håndterbare og som en sideeffekt benytter lav kobling *(Andet GRAS Pattern, der beskrives senere)*

Generel Løsning

Tildel ansvar, så bindingen forbliver høj. Brug dette til at vurdere alternativer. En klasse med lav binding laver mange urelaterede ting, eller har for mange opgaver.
Sådanne klasser ønskes ikke fordi de kan være:

- Svær at forstå
- Svær at genanvende
- Konstant udsat for forandringer

Klasser med lav binding har tit for meget ansvar, som kunne uddelegeres til andre objekter.

Vores Løsning

Et godt eksempel i vores program, er at vores *Game*klasse, holder styr på spillerne i *Player* klassen. Istedet for at *Game* også holder styr på spillerens pengebeholdning, har vi givet ansvaret til *Player*, som holder styr på *Account*.

11.5 Indirection

Beskrivelse af Indirection

Problem

Hvor skal man tildele et ansvarsområder, for at undgå direkte kobling mellem to eller flere ting?
Hvordan afkobler man objekter, så man opnår lav kobling, og genanvendelsesmuligheder forbliver høje.

Generel Løsning

Giv ansvaret til et mellemliggende objekt til at kommunikere mellem andre komponenter, så de ikke er direkte sammenkoblet.
Det mellemliggende objekt laver en *inderection* mellem de andre komponenter.

Vores Løsning

Vi har forsøgt at bruge *Inderection*, ved at lave en *Graphic* klasse, der holder styr på alt, der foregår i forhold til at kalde vores eksterne GUI. Dette gjorde vi for, at det ville være nemmere at lave spillet om, hvis der skulle ske modifikationer i GUI biblioteket. Dette skete rent faktisk, da vi var midt i projektet. Vi fik besked om at GUI'en, nu var blevet ændret. Selvom vi syntes det var lidt irriterende, kunne vi hurtigt rette det til, da vi kun skulle rette i den ene klasse.

11.6 Low Coupling (Lav kobling)

Beskrivelse af Low Coupling

Problem

Hvordan opnår man lav afhængighed. lav forandrings indvirkning og forhøjet genanvendelse.
Coubling er et mål for hvor stærkt et element er koblet til, har kendskab til eller er afhængig af andre elementer.
En klasse med høj kobling afhænger af mange klasser. Sådanne klasser kan være uønskede. De kan lide af følgende problemer.

- Tvunge lokale forandringer på grund af forandringer i sammenhængende klasser.
- Sværere at forstå i isolerede tilfælde.
- Sværere at genanvende, da det vil kræve tilhørende tilstedeværelse af afhængige klasser.

Generel Løsning

Tildel ansvar, så koblingen forbliver lav. Brug dette princip til at evaluere alternativer.

Vores Løsning

Dette er forsøgt implementeret i for eksempel vores *Game*, der istedet for at kalde til *Player* for bagefter at kalde *Account*, og dermed skabe høj kobling, har vi valgt at kalde *Account* igennem *Player* klassen. Dette sikrer en lav kobling, men samtidig også høj binding, da man sjældent kan bruge principperne alene.

11.7 Polymorphism

Beskrivelse af Polymorphism

Problem

Hvordan opretter man alternativer baseret på typer eller software, der kan kobles direkte på de allerede eksisterende komponenter. *Alternativer baseret på typer* - Hvis et program er designet med et if-else eller switch statement, og en ny variation opstår, kan det ofte betyde ændringer mange steder i koden. Denne fremgangsmåde gør det besværligt at udvide et program på en nem måde. Dette er fordi, ændringerne skal foretages flere forskellige steder, hvor denne betingelses logik er implementeret.

Generel Løsning

Når familiære alternativer eller opførsel varierer efter type (klasse), gives ansvaret for opførslen til typerne, hvori opførslen varierer. Dette gøres ved at bruge polymorfiske operatgioner.

Vores Løsning

Vi har ikke benyttet polymorphi i vores spil. Vi har dog tænkt på, at vores felter måske i fremtiden, vil komme til at være af forskellige typer. Dermed kan det blive aktuelt at benytte mønstret for polymorfi ved en eventuel opdatering af spillet.

11.8 Protected Variations

Beskrivelse af Protected Variations

Problem

Hvordan tildeler man ansvar til objekter, subsystemer eller systemer, så variationer og ustabilitet i disse elementer ikke får en uønsket effekt på andre elementer.

Generel Løsning

Identificer punkter med uønsket variation eller ustabilitet. Lav en stabil grænseflade om dem. Dette bruges tit i forbindelse med polymorfi.

Vores Løsning

I vores spil har det ikke været nødvendigt at bruge dette mønster.

11.9 Pure Fabrication

Beskrivelse af Pure Fabrication

Problem

Objekt orienterede designs er nogle gange karakteriserede ved at tage udgangspunkt i problemdomæner fra den virkelige verden, for at lette forståelsen. For eksempel *Gameboard* og *Field* klasserne. Nogle gange kan der der opstå situationer, hvor det giver problemer kun at tildele ansvar til domæne lags klasser. Det kan give dårlig binding, kobling eller lav genanvedelsesmulighed.

Generel Løsning

Giv et sæt ansvarsopgaver med høj sammenhæng til en kunstig eller belejelig klasse, der ikke er repræsenteret i domænelaget. Denne opfundne klasse skal supportere høj binding, lav kobling og være nem at genbruge.

Vores Løsning

Vi har opfundet klasser til at håndtere det grafiske i spillet med klassen *Graphic*, og den tekstbaserede del kørers i klassen *TUI*. Disse klasser opfylder ovenstående krav.

12 Kildeliste

Her vil vi oplyse om hvilke kilder, der er brugt til rapporten. Vi vil både oplyse om hvilke bøger, hjemmesider og software vi har brugt.

12.1 Bøger/rapporter

Referencer

[IOO13] IOOuterActive. *CDIO opgave 3 (Version 12)*. IOOuterActive, nov. 2013.

[Lar04] Craig Larman. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development (3rd Edition)*. Prentice Hall, 2004. ISBN: 0131489062.

[Wik13] Wikipedia. *FURPS — Wikipedia, The Free Encyclopedia*. [Online; accessed 16-November-2013]. 2013. URL: `\url{http://en.wikipedia.org/wiki/FURPS}`.

- Applying UML and patterns - An introduction to Object-Oriented Analysis and Design and Iterative Development - Craig Larman, Third Edition
- Java Software Solutions - Foundations of Program Design - Lewis and Loftus, Seventh Edition

- 19_del1 - Ahlgreen, Johansen og Mortensen

12.2 Hjemmesider

12.3 Programmer

- Eclipse - kepler
- UMLet
- Google Docs
- TexMaker

13 Bilag

13.1 Kode

Her vil hele vores kode til programmet være repræsenteret som bilag.

13.1.1 TUI - Boundary

```
1  package Boundary;
2
3  import java.util.Scanner;
4
5  import Entity.Player;
6
7  /**
8   * Class to handle input/output to/from the console.
9   *
10  * @author DTU 02312 Gruppe 19
11  *
12  */
13 public class TUI {
14   /**
15    * Prints game rules.
16    */
17   public static void printRules() {
18     System.out.println("");
19     System.out
20         .println("
                ------------------------------------------------------------------------
                ");
21     System.out
22         .println("| Rules:
                |");
23     System.out
24         .println("| You roll two dice. The sum determines
               which field you hit. |");
25     System.out
26         .println("| Each field have it's own value.
                                        |");
27     System.out
28         .println("| You get points on some, and loses
               points on others.       |");
29     System.out
30         .println("| The first player to hit 3000 points
               win.                    |");
```

```
31      System.out
32          .println("| Press \"Enter\" to roll, press \"q\"
                to exit.                    |");
33      System.out
34          .println("
                ------------------------------------------------------------------
                ");
35      System.out.println("");
36    }
37
38    /**
39     * Method to get the name of a field from the field
           number.
40     * The names are given according to the game rules, but
           could be translated.
41     *
42     * @param fieldNumber The number of the field to get a
           name for.
43     * @return The name of the field with the given number.
           Null if the field doesn't exist.
44     */
45    public static String getFieldName(int fieldNumber) {
46      switch (fieldNumber) {
47      case 2:
48        return "Tower";
49      case 3:
50        return "Crater";
51      case 4:
52        return "Palace gates";
53      case 5:
54        return "Cold Desert";
55      case 6:
56        return "Walled city";
57      case 7:
58        return "Monastery";
59      case 8:
60        return "Black cave";
61      case 9:
62        return "Huts in the mountain";
63      case 10:
64        return "The Wearwall";
65      case 11:
66        return "The pit";
67      case 12:
68        return "Goldmine";
69      }
```

```
70
71      return null;
72    }
73
74    /**
75     * Method to get the description of a field from the
          field number.
76     * The descriptions are given according to the game
          rules, but could be translated.
77     *
78     * @param fieldNumber The number of the field to get a
          description for.
79     * @return The description for the field with the given
           number. Null if the field doesn't exist.
80     */
81    public static String getFieldDescription(int
          fieldNumber) {
82      switch (fieldNumber) {
83      case 2:
84        return "You entered the tower. Gain 250 credits for
             climbing it.";
85      case 3:
86        return "You crashed into the road creating a crater
            . Pay 200 credits towards repair costs.";
87      case 4:
88        return "You went sightseeing at the palace gates
            but got robbed. Pay 100 credits.";
89      case 5:
90        return "You ran out of gas exploring the Cold
            Desert. Pay 20 credits for more gas to get home.
            ";
91      case 6:
92        return "You bet on a race in the Walled City and
            won. Gain 180 credits.";
93      case 7:
94        return "You climbed the mountain and gained wisdom.
             Do nothing.";
95      case 8:
96        return "You got lost in the black cave. Pay a park
            ranger 70 credits for helping you out.";
97      case 9:
98        return "You accidentally set fire to a hut on the
            mountain. Pay 60 credits for materials to
            rebuild.";
99      case 10:
```

```
100         return "You got caught naked and drunk strolling
                around in Wearwall. Pay the police a fine of 80
                credits.\nThey felt sorry for you. Recieve
                another turn.";
101       case 11:
102         return "You tried to steal a police car. Pay 90
                credits as bail.";
103       case 12:
104         return "You travelled to Alaska and struck it big
                in gold. Recieve 650 credits.";
105       }
106
107       return null;
108     }
109
110     /**
111      * Prints a short text, asking the specified player to
             type his name.
112      *
113      * @param playerNo The player number to print as part
             of the message.
114      */
115     public static void printNameRequest(int playerNumber) {
116       System.out
117           .println("Insert name for player " + (
                  playerNumber + 1) + ".");
118     }
119
120     /**
121      * Prints a short text, telling the player who's turn
             it is, and asking him to
122      * roll.
123      *
124      * @param name The name to print as part of the message
             .
125      */
126     public static void printTurn(String name) {
127       System.out.print("\nIt's " + name + "'s turn. Press
             enter to roll.");
128     }
129
130     /**
131      * Prints the current status of the game. Thats all
             players score and the
132      * sum of the dice.
133      *
```

```
134     * @param players An array of players to get the
            information from.
135     * @param sum The value that was hit with the dice.
136     */
137    public static void printStatus(Player[] players, int
           sum) {
138      System.out.println("You hit field number " + sum + ",
             "
139          + getFieldName(sum) + "\n" + getFieldDescription(
               sum));
140      System.out.println("The score is now:");
141
142      int i;
143      for (i = 0; i < players.length; i++) {
144        System.out.print(players[i].getName() + " = "
145            + players[i].getAccount().getAccountValue() + "
                 \t");
146      }
147
148      System.out.println();
149    }
150
151    /**
152     * Prints the name and score of the winning player.
153     *
154     * @param name The name of the player who should be
            declared the winner.
155     * @param score The score for the winning player.
156     */
157    public static void printWinner(String name, int score)
           {
158      System.out.println("Congratulations! " + name + " has
             won with "
159          + score + " points!\nPress Enter to exit.");
160    }
161
162    /**
163     * Prints the name and score of the losing player.
164     *
165     * @param name The name of the player who should be
            declared the loser.
166     * @param score The score for the losing player.
167     */
168    public static void printLoser(String name, int score) {
169      System.out.println("Sorry! " + name + " you have lost
             with " + score
```

```
170           + " points!\nPress Enter to exit.");
171     }
172
173     /**
174      * Reads a line from the console.
175      *
176      * @param scanner The scanner to read from
177      * @return Whatever the user inputs.
178      */
179     public static String getUserInput(Scanner scanner) {
180       return scanner.nextLine();
181     }
182 }
```

13.1.2 Graphic - Boundary

```
1  package Boundary;
2
3  import Entity.Player;
4  import boundaryToMatador.GUI;
5
6  /**
7   * Class to send commands to the GUI.
8   *
9   * @author DTU 02312 Gruppe 19
10  *
11  */
12 public class Graphic {
13   /**
14    * Method to set the value of the dice on the GUI.
15    *
16    * @param die1 Value of die1.
17    * @param die2 Value of die2.
18    */
19   public static void setDice(int die1, int die2) {
20     GUI.setDice(die1, die2);
21   }
22
23   /**
24    * Method to add a player.
25    *
26    * @param playerName The name of the player to add.
27    * @param playerScore The score of the player to add.
28    */
29   public static void addPlayer(String playerName, int
       playerScore) {
```

```
30      GUI.addPlayer(playerName, playerScore);
31    }
32
33    /**
34     * Method to update all players information on the GUI
          according to a given array of player objects.
35     *
36     * @param players The array of player objects to get
          the information from.
37     */
38    public static void updatePlayers(Player[] players) {
39      int i;
40
41      for(i = 0; i<players.length; i++) {
42        GUI.setBalance(players[i].getName(), players[i].
              getAccount().getAccountValue());
43      }
44    }
45
46    /**
47     * Close the GUI window.
48     */
49    public static void close() {
50      GUI.close();
51    }
52
53    /**
54     * Method to setup all the fields on the GUI, according
          to the rules of the game.
55     * Also clears all the unused fields.
56     */
57    public static void setupFields() {
58      createField("Tower", "+250", 2);
59      createField("Crater", "-200", 3);
60      createField("Palace gates", "-100", 4);
61      createField("Cold Desert", "-20", 5);
62      createField("Walled city", "+180", 6);
63      createField("Monastery", "0", 7);
64      createField("Black cave", "-70", 8);
65      createField("Huts in the mountain", "-60", 9);
66      createField("The Wearwall", "-80", 10);
67      createField("The pit", "-90", 11);
68      createField("Goldmine", "+650", 12);
69
70      //Remove unused fields from GUI
71      int i;
```

```
72      createField("", "", 1);
73      for(i=13; i<41; i++) {
74        createField("", "", i);
75      }
76    }
77
78    /**
79     * Method to move a car from any field to the field
          number given.
80     *
81     * @param playerName The name of the player who's car
          should be moved.
82     * @param fieldNumber The number of the field the car
          should be moved to.
83     */
84    public static void moveCar(String playerName, int
        fieldNumber) {
85      GUI.removeAllCars(playerName);
86      GUI.setCar(fieldNumber, playerName);
87    }
88
89    /**
90     * Helper method to setup all the parameters of a field
          on the GUI at the same time.
91     *
92     * @param title The title to set.
93     * @param subTitle The subtitle to set.
94     * @param fieldNumber The number of the field to change
          .
95     */
96    private static void createField(String title, String
        subTitle, int fieldNumber) {
97      GUI.setTitleText(fieldNumber, title);
98      GUI.setSubText(fieldNumber, subTitle);
99      GUI.setDescriptionText(fieldNumber, title);
100   }
101 }
```

13.1.3 Main - Controller

```
1  package Control;
2
3  public class Main {
4      public static void main(String[] args) {
5        //TestTools.AccountTesterController at = new
             TestTools.AccountTesterController();
```

```
6         //at.testAccount();
7
8         Game game = new Game();
9         game.startGame();
10      }
11  }
```

13.1.4 Game - Controller

```
1   package Control;
2
3   import java.util.Scanner;
4
5   import Boundary.Graphic;
6   import Boundary.TUI;
7   import Entity.DieCup;
8   import Entity.GameBoard;
9   import Entity.Player;
10
11  /**
12   * This is controller class in the dice game.
13   *
14   * @author DTU 02312 Gruppe 19
15   *
16   */
17  public class Game {
18    private final int POINTS_TO_START_WITH = 1000;
19    private final int POINTS_TO_WIN = 3000;
20    private final int NUMBER_OF_PLAYERS = 2;
21
22    private DieCup dieCup;
23    private Scanner scanner;
24    private GameBoard gameBoard;
25    // private GameBoardTest gameBoard;
26    private Player[] players;
27
28    /**
29     * Game constructor. Creates new instances of the
          required classes.
30     */
31    public Game() {
32      int i;
33
34      dieCup = new DieCup();
35      scanner = new Scanner(System.in);
36      gameBoard = new GameBoard();
```

```
37      // gameBoard = new GameBoardTest();
38      players = new Player[NUMBER_OF_PLAYERS];
39
40      // Make all player-objekts in loop
41      for (i = 0; i < NUMBER_OF_PLAYERS; i++) {
42        players[i] = new Player(POINTS_TO_START_WITH);
43      }
44
45      Graphic.setupFields();
46    }
47
48    /**
49     * Start the game.
50     */
51    public void startGame() {
52      int activePlayer, i, scoreToAdd;
53      String userInput;
54
55      TUI.printRules();
56
57      // Ask for all player names and save them in the
            player objects. Also
58      // adds the players to the GUI.
59      for (i = 0; i < NUMBER_OF_PLAYERS; i++) {
60        TUI.printNameRequest(i);
61        players[i].setName(TUI.getUserInput(scanner));
62        Graphic.addPlayer(players[i].getName(), players[i].
              getAccount()
63            .getAccountValue());
64      }
65
66      // Player 1 always starts. However this would work
            with Player 2, or
67      // even random.
68      activePlayer = 0;
69
70      // Start of the actual game. Infinite loop is broken
            only when someone
71      // wins or inputs "q"
72      while (true) {
73        // Write whos turn it is and wait for input
74        TUI.printTurn(players[activePlayer].getName());
75        userInput = TUI.getUserInput(scanner);
76
77        // Exit game if user inputs "q"
78        if ("q".equals(userInput)) {
```

```
79            cleanUp();
80          }
81
82          // Shake, and add the sum of the dice to the
                current players score
83          dieCup.shakeDieCup();
84
85          // Add points from field
86          scoreToAdd = gameBoard.getField(dieCup.getSum()).
                getFieldScore();
87          if(!players[activePlayer].getAccount().addToAccount
                (scoreToAdd)) {
88            loseTasks(activePlayer);
89          }
90
91          // Write status/score to both TUI and GUI
92          statusTasks(activePlayer);
93
94          // Check if player have won
95          if (players[activePlayer].getAccount().
                getAccountValue() >= POINTS_TO_WIN) {
96            winTasks(activePlayer);
97          }
98
99          // Switch turn to the next player, unless the
                current player gets an
100         // extra turn
101         if (!gameBoard.getField(dieCup.getSum()).
                getGivesExtraTurn()) {
102           activePlayer = getNextPlayer(activePlayer);
103         }
104       }
105     }
106
107     /**
108      * Simple method to get the number of the next player.
109      *
110      * @param input The number to toggle away from.
111      * @return 2 if 1 is given etc., but gives 1 if the
            value for number of players is reached.
112      */
113     private int getNextPlayer(int input) {
114       if (input + 1 >= NUMBER_OF_PLAYERS) {
115         return 0;
116       }
117
```

```
118       return input + 1;
119     }
120
121     /**
122      * Writes score and dice values to both GUI and TUI
123      */
124     private void statusTasks(int activePlayer) {
125       TUI.printStatus(players, dieCup.getSum());
126       Graphic.setDice(dieCup.getValueDie1(), dieCup.
              getValueDie2());
127       Graphic.updatePlayers(players);
128       Graphic.moveCar(players[activePlayer].getName(),
              dieCup.getSum());
129     }
130
131     /**
132      * Prints the name of the given player, along with a
            message telling that
133      * he has won the game., then waits for input, to make
            sure the message
134      * stays on the screen. Ends the program when any input
            is given.
135      *
136      * @param activePlayer The number of the player who
            should be declared the winner.
137      */
138     private void winTasks(int activePlayer) {
139       TUI.printWinner(players[activePlayer].getName(),
              players[activePlayer]
140           .getAccount().getAccountValue());
141       TUI.getUserInput(scanner);
142       cleanUp();
143     }
144
145     /**
146      * Prints the name of the given player, along with a
            message telling that
147      * he has lost the game., then waits for input, to make
            sure the message
148      * stays on the screen. Ends the program when any input
            is given.
149      *
150      * @param activePlayer The number of the player who
            should be declared the loser.
151      */
152     private void loseTasks(int activePlayer) {
```

```
153       TUI.printLoser(players[activePlayer].getName(),
              players[activePlayer]
154               .getAccount().getAccountValue());
155       TUI.getUserInput(scanner);
156       cleanUp();
157     }
158
159     /**
160      * Closes the program and cleans up properly.
161      */
162     private void cleanUp() {
163       Graphic.close();
164       scanner.close();
165       System.exit(0);
166     }
167   }
```

13.1.5 Player - Entity

```
1   package Entity;
2
3   /**
4    * Class to create a player. This class can be used for
         storing a name and account for a player.
5    *
6    * @author DTU 02312 Gruppe 19
7    *
8    */
9   public class Player {
10      private String name;
11      private Account account;
12
13      /**
14       * Constructor that initiates name to empty and
             account to an initial score.
15       */
16      public Player(int initialScore) {
17          name = "";
18          account = new Account(initialScore);
19      }
20
21      /**
22       * Saves the given name.
23       *
24       * @param input The name to save.
25       */
```

```
26      public void setName(String input) {
27          name = input;
28      }
29
30      /**
31       * Gets the name of the player
32       *
33       * @return The name of this player.
34       */
35      public String getName() {
36          return name;
37      }
38
39      /**
40       * Gets the account object.
41       *
42       * @return The players account.
43       */
44      public Account getAccount() {
45          return account;
46      }
47
48      /**
49       * Method that makes a text with the most important
             values in the class, and some description.
50       *
51       * @return A coherent string with values of name and
             account.
52       */
53      public String toString() {
54          return "Name = " + name + ", Account = " +
                account;
55      }
56  }
```

13.1.6 Account - Entity

```
1   package Entity;
2
3   /**
4    * Class to create an Account. This class can be used to
       store a number representing an account value.
5    *
6    * @author DTU 02312 Gruppe 19
7    *
8    */
```

```
 9 public class Account {
10   private int accountValue;
11
12   /**
13    * Constructor to create a new account. Takes no
          arguments, and sets the initial account value to 0.
14    */
15   public Account() {
16     accountValue = 0;
17   }
18
19   /**
20    * Constructor to create a new account. Takes an
          argument for initial account value.
21    *
22    * @param initialAccountValue The value to set the new
          account to.
23    */
24   public Account(int initialAccountValue) {
25     accountValue = initialAccountValue;
26   }
27
28   /**
29    * Method to set the account.
30    * Checks if the account will go below 0 before setting
          it.
31    *
32    * @param input The value to set the account to.
33    * @return True if the account was set correctly. False
          if the account was not set, because the input
          value was below 0.
34    */
35   public boolean setAccountValue(int input) {
36     if(input >= 0) {
37       accountValue = input;
38       return true;
39     }
40
41     return false;
42   }
43
44   /**
45    * Method to get the value of the account.
46    *
47    * @return The value of the account.
48    */
```

```
49    public int getAccountValue() {
50      return accountValue;
51    }
52
53    /**
54     * A method to add to the account, so the resulting
          value will be the existing value plus the input
          value.
55     * Checks if the account will go below 0 before adding
          to it.
56     *
57     * @param input The value to add.
58     * @return True if the account was set correctly. False
           if the account was not set, because the value
          would have gone below 0.
59     */
60    public boolean addToAccount(int input) {
61      if(accountValue + input >= 0) {
62        accountValue = accountValue + input;
63        return true;
64      }
65
66      return false;
67    }
68
69    /**
70     * A method to get the contents of the account as a
          string.
71     *
72     * @return The account value as a string.
73     */
74    public String toString() {
75      return Integer.toString(accountValue);
76    }
77  }
```

13.1.7 Gameboard - Entity

```
1  package Entity;
2
3  /**
4   * Class to create a game board. This class takes in a
       lot of fields and makes it a board.
5   *
6   * @author DTU 02312 Gruppe 19
7   *
```

```
 8   */
 9  public class GameBoard {
10    Field[] fields;
11
12    /**
13     * Constructor that makes an array of fields and sets
          it according to the rules of the game.
14     */
15    public GameBoard() {
16      fields = new Field[13];
17
18        //Create the fields
19      fields[2] = new Field(250);
20      fields[3] = new Field(-200);
21      fields[4] = new Field(-100);
22      fields[5] = new Field(-20);
23      fields[6] = new Field(180);
24      fields[7] = new Field(0);
25      fields[8] = new Field(-70);
26      fields[9] = new Field(-60);
27      fields[10] = new Field(-80, true);
28      fields[11] = new Field(-90);
29      fields[12] = new Field(650);
30    }
31
32    /**
33     * Takes the number of a field and gives the
          corresponding field-object.
34     *
35     * @param fieldNumber The number of the field to get.
36     * @return The field object corresponding to the number
          given.
37     */
38    public Field getField(int fieldNumber) {
39      return fields[fieldNumber];
40    }
41
42    /**
43     * A method to generate a nice string containing the
          value of all the fields.
44     *
45     * @return All the field values as a string.
46     */
47    public String toString() {
48      String output = "";
49      int i;
```

```
50
51      for(i=0; i<fields.length; i++) {
52        if(fields[i] != null) {
53          output = output + fields[i] + "\n";
54        }
55      }
56
57      return output;
58    }
59  }
```

13.1.8 Field - Entity

```
1   package Entity;
2
3   /**
4    * Class to create a field. This class can be used to
         contain the score of a field and a value for extra
         turn.
5    *
6    * @author DTU 02312 Gruppe 19
7    *
8    */
9   public class Field {
10    private int fieldScore;
11    private boolean givesExtraTurn;
12
13    /**
14     * Constructor for a field.
15     * Takes only the score of the field, and assumes the
           field does not give an extra turn.
16     *
17     * @param fieldScore The value that should be added to
           a player if they hit this field.
18     */
19    public Field(int fieldScore) {
20      this.fieldScore = fieldScore;
21      this.givesExtraTurn = false;
22    }
23
24    /**
25     * Constructor for a field.
26     * Takes the score of the field and a true/false value
           for wether the field gives an extra turn.
27     *
```

```
28      * @param fieldScore The value that should be added to
            a player if they hit this field.
29      * @param givesExtraTurn A value for wether the field
            should give an extra turn
30      */
31     public Field(int fieldScore, boolean givesExtraTurn) {
32       this.fieldScore = fieldScore;
33       this.givesExtraTurn = givesExtraTurn;
34     }
35
36     /**
37      * Gets the score from a field.
38      *
39      * @return The field score.
40      */
41     public int getFieldScore() {
42       return fieldScore;
43     }
44
45     /**
46      * Gets the true/false value for wether the field
            should give an extra turn.
47      *
48      * @return True for extra turn, otherwise false.
49      */
50     public boolean getGivesExtraTurn() {
51       return givesExtraTurn;
52     }
53
54     /**
55      * Prints a nice string with field score and the value
            for extra turn.
56      *
57      * @return The field score and extra turn value as a
            string.
58      */
59     public String toString() {
60       return "Field score: " + fieldScore + ", gives extra
             turn: " + givesExtraTurn;
61     }
62   }
```

13.1.9 DieCup - Entity

```
1  package Entity;
2
```

```
3  /**
4   * Class to create a diecup. This class will take in 2
        dice.
5   *
6   * @author DTU 02312 Gruppe 19
7   *
8   */
9  public class DieCup {
10
11      private Die die1, die2;
12
13      /**
14       * Constructor that set up two new dice.
15       */
16      public DieCup() {
17          die1 = new Die();
18          die2 = new Die();
19      }
20
21      /**
22       * Method to shake the diecup.
23       */
24      public void shakeDieCup() {
25          die1.roll();
26          die2.roll();
27      }
28
29      /**
30       * Adds value from die1 and die2.
31       *
32       * @return The sum of die1 and die2.
33       */
34      public int getSum() {
35          return die1.getValue() + die2.getValue();
36      }
37
38      /**
39       * Get faceValue from die1
40       *
41       * @return The current value of die1.
42       */
43      public int getValueDie1() {
44          return die1.getValue();
45      }
46
47      /**
```

```
48          * Get faceValue from die2
49          *
50          * @return The current value of die2.
51          */
52         public int getValueDie2() {
53             return die2.getValue();
54         }
55
56         /**
57          * Checks if the values of the dice are equal.
58          *
59          * @return True if facevalues of die1 and die2 are
                the same, otherwise False.
60          */
61         public boolean getEns() {
62             if (die1.getValue() == die2.getValue()) {
63                 return true;
64             }
65             return false;
66         }
67
68         /**
69          * Method that makes a text with the most important
                values in the class, and some description.
70          *
71          * @return A coherent string with values of Die1 and
                Die2
72          */
73         public String toString() {
74             return "Die1 = " + die1.getValue() + ", Die2 = "
                    + die2.getValue();
75         }
76     }
```

13.1.10 Die - Entity

```
1   package Entity;
2
3   import java.util.Random;
4
5   /**
6    * Class to create a die. This class can be used to
        generate random numbers from 1 to 6.
7    *
8    * @author DTU 02312 Gruppe 19
9    *
```

```
10  */
11 public class Die {
12
13     private final int MAX_VALUE = 6;
14     private int faceValue;
15     private Random random;
16
17     /**
18      * Constructor to set the facevalue of the die and
            instanciate the random generator.
19      */
20     public Die() {
21         random = new Random();
22         faceValue = 0;
23     }
24
25     /**
26      * Method to roll the die, and give it a new value.
27      */
28     public void roll() {
29         //Generator makes numbers from 0 to 5, so we add
               1 to get 1 to 6
30         faceValue = random.nextInt(MAX_VALUE) + 1;
31     }
32
33     /**
34      * Method so you can get the facevalue.
35      *
36      * @return The current facevalue of the die.
37      */
38     public int getValue() {
39         return faceValue;
40     }
41
42     /**
43      * Method that makes a string of facevalue to print.
44      *
45      * @return The current facevalue of the die as a
            string.
46      */
47     public String toString() {
48         return Integer.toString(faceValue);
49     }
50 }
```

13.1.11 AccountTesterController - TestTools

```java
package TestTools;

import Entity.Account;

/**
 * Class to test the Account class.
 *
 * @author DTU 02312 Gruppe 19
 *
 */
public class AccountTesterController {
  Account account;

  /**
   * Constructor to make a new account.
   */
  public AccountTesterController() {
    account = new Account();
  }

  /**
   * Method that tast the Account class and prints the
       results directly in the console.
   */
  public void testAccount() {
    //Set test - Limits
    System.out.println("Set to 1 = " + account.
        setAccountValue(1));
    System.out.println("Set to 0 = " + account.
        setAccountValue(0));
    System.out.println("Set to -1 = " + account.
        setAccountValue(-1));
    System.out.println("Set to 1000000000 = " + account.
        setAccountValue(1000000000));
    System.out.println("Set to -1000000000 = " + account.
        setAccountValue(-1000000000));
    //Set test - Average values
    System.out.println("Set to 500 = " + account.
        setAccountValue(500));
    System.out.println("Set to -500 = " + account.
        setAccountValue(-500));
    System.out.println();

    //Add test - Limits
    account.setAccountValue(1000);
```

```
38      System.out.println("Set to 1000, add -1000 = " +
            account.addToAccount(-1000));
39      account.setAccountValue(1000);
40      System.out.println("Set to 1000, add -1001 = " +
            account.addToAccount(-1001));
41      account.setAccountValue(1001);
42      System.out.println("Set to 1001, add -1000 = " +
            account.addToAccount(-1000));
43      //Add test - Average values
44      account.setAccountValue(1000);
45      System.out.println("Set to 1000, add 500 = " +
            account.addToAccount(500));
46      account.setAccountValue(1000);
47      System.out.println("Set to 1000, add -500 = " +
            account.addToAccount(-500));
48    }
49  }
```

13.1.12 DieCupTestEntity - TestTools

```
1   package TestTools;
2
3   /**
4    * Class to create a "cheating" diecup. This class will
         not actually use dice, but just give a hardcoded
         number every time.
5    *
6    * @author DTU 02312 Gruppe 19
7    *
8    */
9   public class DieCupTestEntity {
10    private final int VALUE_TO_GIVE1 = 1;
11    private final int VALUE_TO_GIVE2 = 2;
12
13    /**
14     * Constructor
15     */
16      public DieCupTestEntity() {
17
18      }
19
20      /**
21       * Method to shake the diecup.
22       * This won't do anything in this class, since the
             values are final. It just have to be there, to
             make
```

```
23       * it possible to substitute the "real" DieCup with
             this "false" one.
24       */
25      public void shakeDieCup() {
26        //Does nothing, since the values are fixed
27      }
28
29      /**
30       * Adds value from die1 and die2.
31       *
32       * @return The sum of die1 and die2.
33       */
34      public int getSum() {
35          return VALUE_TO_GIVE1 + VALUE_TO_GIVE2;
36      }
37
38      /**
39       * Get faceValue from die1
40       *
41       * @return The current value of die1.
42       */
43      public int getValueDie1() {
44          return VALUE_TO_GIVE1;
45      }
46
47      /**
48       * Get faceValue from die2
49       *
50       * @return The current value of die2.
51       */
52      public int getValueDie2() {
53          return VALUE_TO_GIVE2;
54      }
55
56      /**
57       * Checks if the values of the dice are equal.
58       *
59       * @return True if facevalues of die1 and die2 are
             the same, otherwise False.
60       */
61      public boolean getEns() {
62          if (VALUE_TO_GIVE1 == VALUE_TO_GIVE2) {
63              return true;
64          }
65          return false;
66      }
```

```
67
68      /**
69       * Method that makes a text with the most important
             values in the class, and some description.
70       *
71       * @return A coherent string with values of Die1 and
             Die2
72       */
73      public String toString() {
74          return "Die1 = " + VALUE_TO_GIVE1 + ", Die2 = " +
                VALUE_TO_GIVE2;
75      }
76  }
```

13.1.13 GameBoardTestEntity - TestTools

```
1   package TestTools;
2
3   import Entity.Field;
4
5   /**
6    * Class to create a "testing" game board. This will work
         exactly like the "real" gameboard,
7    * except for the field values which are customized to
         test the limits of the methods in account and game-
         classes.
8    *
9    * @author DTU 02312 Gruppe 19
10   *
11   */
12  public class GameBoardTestEntity {
13    Field[] fields;
14
15    /**
16     * Constructor that makes an array of fields and sets
           it to some good testing values.
17     */
18    public GameBoardTestEntity() {
19      fields = new Field[13];
20
21      // Create the fields
22      fields[2] = new Field(-3001);
23      fields[3] = new Field(-3000);
24      fields[4] = new Field(-2999);
25      fields[5] = new Field(-4000);
26      fields[6] = new Field(-5500);
```

```
27        fields[7] = new Field(-7000);
28        fields[8] = new Field(-700000);
29        fields[9] = new Field(-1000000);
30        fields[10] = new Field(-8000, true);
31        fields[11] = new Field(-9000);
32        fields[12] = new Field(-6500);
33      }
34
35      /**
36       * Takes the number of a field and gives the
             corresponding field-object.
37       *
38       * @param fieldNumber The number of the field to get.
39       * @return The field object corresponding to the number
             given.
40       */
41      public Field getField(int fieldNumber) {
42        return fields[fieldNumber];
43      }
44
45      /**
46       * A method to generate a nice string containing the
             value of all the fields.
47       *
48       * @return All the field values as a string.
49       */
50      public String toString() {
51        String output = "";
52        int i;
53
54        for (i = 0; i < fields.length; i++) {
55          if (fields[i] != null) {
56            output = output + fields[i] + "\n";
57          }
58        }
59
60        return output;
61      }
62    }
```