Danmarks Tekniske Universitet

DTU

02312, 62531, 62532
INDLEDENDE PROGRAMMERING, UDVIKLINGSMETODER TIL IT-SYSTEMER OG VERSIONSSTYRING OG TESTMETODER

CDIO del 3

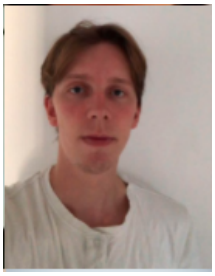
`https://github.com/ExoHumann/40_del3.git`

Authors

Krzysztof Kisiel s192884

Oliver Olsen s205443

Johannes Jensen s205413



Gustav Fauser s205446

Jonathan Hansen s205413

Alaa Hadood s205413

19. november 2020

1 Timeplan

Tid i timer Navn	Krav og analyse	Design	Implementering	Test	Programmering
Gustav Fauser		1			
Oliver Olsen		1			
Krzysztof Kisiel		1			
Johannes Jensen	4	3	1	0	2
Jonathan Høj		1			
Alaa Hadood		1			

Tabel 1: Tabel over antal timer brugt på hver del af projektet

2 Summary

Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Donec odio elit, dictum in, hendrerit sit amet, egestas sed, leo. Praesent feugiat sapien aliquet odio. Integer vitae justo. Aliquam vestibulum fringilla lorem. Sed neque lectus, consectetuer at, consectetuer sed, eleifend ac, lectus. Nulla facilisi. Pellentesque eget lectus. Proin eu metus. Sed porttitor. In hac habitasse platea dictumst. Suspendisse eu lectus. Ut mi mi, lacinia sit amet, placerat et, mollis vitae, dui. Sed ante tellus, tristique ut, iaculis eu, malesuada ac, dui. Mauris nibh leo, facilisis non, adipiscing quis, ultrices a, dui.

Indhold

3 Introduktion

IOOuterActive (os) har fået til at opgave at designe og implementere et Monopoly Junior spil. Der skal implementeres de væsentligste elementer fra brætspillet, for at spillet kan spilles. Spillet skal kunne spilles mellem 2-4 spillere.

4 Krav

For at spillet lever op til kundens forventninger opstilles der en række krav som spillet skal leve op til. Kravene er opstillet ud fra kundens vision(**nyborg_cdio_nodate**)

1. Få et beløb når du passerer start feltet.
2. Tag chancekort læg det nederst i bunken.
3. Implementering af forskellige chancekort (beslutter vi sammen).
4. Fængsel betal for at komme ud eller brug "Get out of jail" kortet.
5. Parkering / Besøg i fængsel.
6. Der skal være mulighed for at købe et felt hvis man lander på det.
7. Hvis en spiller ejer et felt, skal andre spillere betale husleje, når det lander på dette.
8. Hvis spilleren ejer flere af samme farve felt koster det mere.
9. Man skal kunne definere sin brik(farve/type/pattern/fill).
10. Man skal kunne se hvem der har købt ethvert felt.
11. Når en spiller lander på et felt, skal de fortsætte fra det felt på næste tur.
12. Spillet skal spilles af 2 til 4 spillere.

5 Analyse

Der skal udarbejdes følgende artifacts (udover kravliste):

1. Use case diagram
2. Eksempler på use case beskrivelser - vælg mindst én, der beskrives fully dressed
3. Domænemodel
4. Et eksempel på systemsekvensdiagram
5. Et eksempel på sekvensdiagram
6. (Et) Designklassediagram

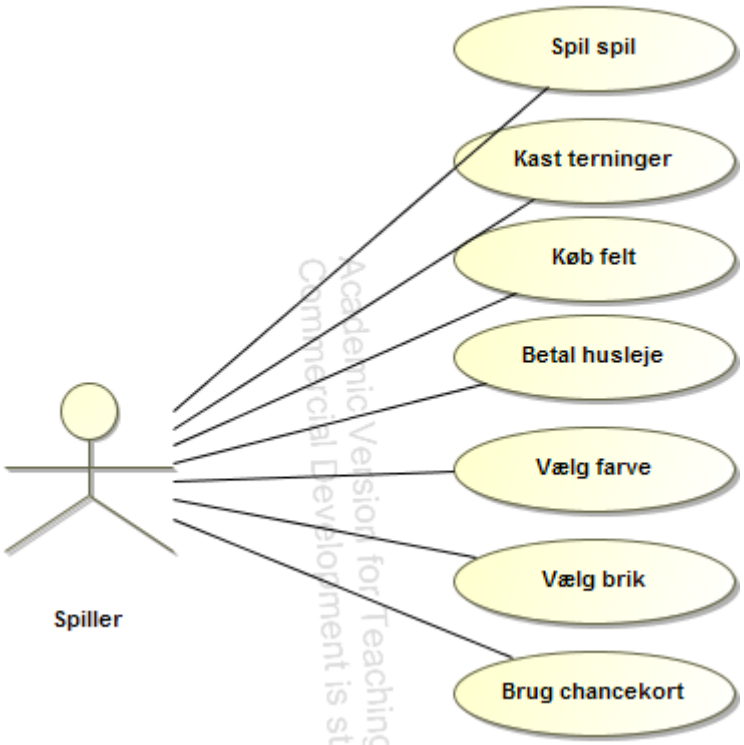
Der er vigtigt, at der er tydelig sammenhæng mellem beskrivelser og diagrammer. Det skal således være muligt at trace fra kravliste til analysedokumentation, designdokumentation og implementering. Beskriv ligeledes jeres brug af GRASP-mønstre.

5.1 Use cases

I opgaven er spillet, udarbejdet ud fra en række Use cases. Use cases beskriver det spillet skal bruges til at en bruger/spiller.

- Spil spil
- Kast terning
- Vælg farve/brik
- Køb felt
- Brug chancekort
- Betal husleje

De ovenstående use cases er illustreret i Figur 1



Figur 1: Use case diagram

Herunder i tabel 2 er use casen Spil spil blevet beskrevet fully dressed

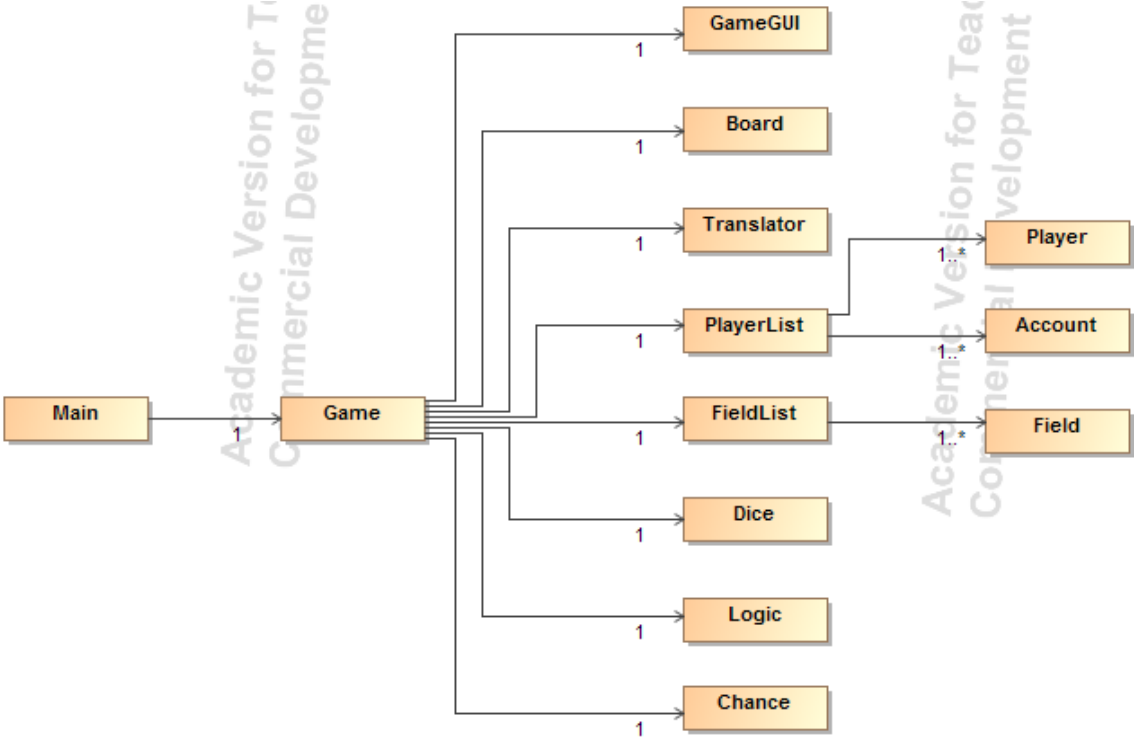
Use case	Spil spil
Omfang	UC1
Mål	Spilleren påbegynder spillet, for et antal spillere.
Primær aktør	Spilleren
Interessenter	De andre medspillere
Startbetingelser	Spilleren vil gerne spille spillet
Succes kriterie	Spillet starter med det valgte antal spillere
Hoved succes Scenarie	1. 2-4 personer vil gerne spille spillet. 2. Spillet startes op på computeren 3. Der vælges et antal spillere fra 2-4 4. Der vælges navn for spillerne • Spiller 2-4 vælger navn *gentag indtil alle spiller har valgt navn.* 5. Der vælges farve for spillerne • Spiller 2-4 vælger farve *gentag indtil alle spiller har valgt en farve.* 6. Der vælges en brik til alle spillerne • Spiller 2-4 vælger brik *gentag indtil alle spiller har valgt en brik.* 7. Spillet starter med de valgte navne, farver og brikker.

Udvidelser	• Der indtaster mindre eller flere spillere end 2-4 1. Spillet giver en fejlbesked 2. Spilleren ændrer antal spillere 3. forsætter fra 3. i hoveddelen • Der indtastet en forkert farve for en af spillerne 1. Spillet giver en fejlbesked 2. Spillet lukker ned • Der vælges en forkert brik, eller en brik der er optaget 1. Spillet giver en fejlbesked 2. Spilleren vælger en anden brik 3. fortsætter fra 7. i hoveddelen
Specielle krav	Ingen
Teknologi- og dataformats-variationer	Ingen
Hyppighed	Hver gang spillet starter fra ny
Diverse	Ingen

Tabel 2: Fully dressed beskrivelse af use case UC1

5.2 Domænemodel

Der er blevet udarbejdet en domænemodel (figur 2) over spillet, hvor sammenhæng mellem klasserne fremgår.



Figur 2: Domænemodel over Spil

5.3 Grasp-mønstrer

GRASP-mønstre er generelle design mønstre til at give ansvar til enkelte klasser i et program/spil.

Skaber Skaber mønsteret tildeler ansvaret for hvilken klasser der opretter nye objekter. I spillet har klassen "Game"ansvaret for at oprette nye objekter, som man kan se på figur 2

Lav kobling Lav kobling omhandler at klasser har lav afhængighed af andre klasser, samt ændringer i en klasse ikke påvirker andre klasser. Det betyder også at klasserne er lettere at genbruge i andre projekter. Lav kobling anvendes på alle klasserne i spillet.

Høj sammenhørighed Høj sammenhørighed omhandler at hver klasse har sit eget veldefineret og fokuserede ansvarsområde. Mønsteret er anvendt i mange af spillets klasser f.eks. i "Dice-klassen, som kun indeholder attributter og metoder der omhandler terninger.

5.4 Klasser

Udfra domænemodellen er der blevet defineret en række klasser, som spillet består af. Klasserne er blevet defineret ud fra MVC-princippet (Model view controller).

5.4.1 Game

I "Game" klassen, bliver alt vores kode taget i brug. Game opretter alle de andre klasser og anvender metoder fra dem til at kører spillet, den bliver med andre ord brugt som en "Controller"til hele spillet. Klassen opretter f.eks. den grafiske grænsflade, alle spillerne, og felterne.

5.4.2 Logic

"Logic" klassen er ansvarlig for spillereglerne, så som at spillet afsluttes når en af spillerne ikke har flere penge tilbage og at spillerne på skift får deres tur. "Logic" klassen er også ansvarlig for at sende spillerne i fængsel, trække chancekort, køb af felter, flytte spillerne osv.

5.4.3 Account

I "Account" klassen, kontrolleres spillernes pengebeholdninger. Pengebeholdningen skal ændres hver gang spillerne lander på et nyt felt og der enten skal trækkes penge fra, eller tillægges penge til pengebeholdningen. De andre klasser skal kunne tilgå spillernes pengebeholdninger. Bla. er "Game" klassen afhængig af pengebeholdningen, for at se om en spiller har tabt (ved at miste alle sine penge).

5.4.4 Player

I "Player" klassen skal der holdes styr på spillernes informationer. Deres navn, farve, position, om de er i fængsel, hvor mange felter de ejer og hvilken tur spillerne er i gang med.

5.4.5 PlayerList

I "PlayerList" klassen, skal der holdes styr på antallet af spillere, med deres tilhørende information. Denne klasse gør brug af de informationer, som "Player" klassen holder styr på.

5.4.6 Dice

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

5.4.7 Board

Boardklassen opretter et spillebrættet med informationer fra GUI og fieldlist.

5.4.8 FieldList

I "FieldList" klassen bliver alle felterne oprettet til spillet. Felterne er af typen Field, og bliver opbevaret i et array.

5.4.9 Field

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

5.4.10 GameGUI

GameGUI klassen skal have ansvar for at opdatere den grafiske brugerflade til spillet. Klassen indeholder derfor metoder der opdatere GUI, med information fra selve logikken i spillet. GameGUI klassen skal ikke holde styr på regler, point osv. Men blot have ansvar for den grafiske repræsentation af spillet, for spillerne. Gui klassen holder styr på felter og spiller og samt de biler der bliver brugt til at bevæge spilleren. GameGUI bruger metoder fra "Matador GUI" version 3.1.7. (Budtz m.fl. 2020)

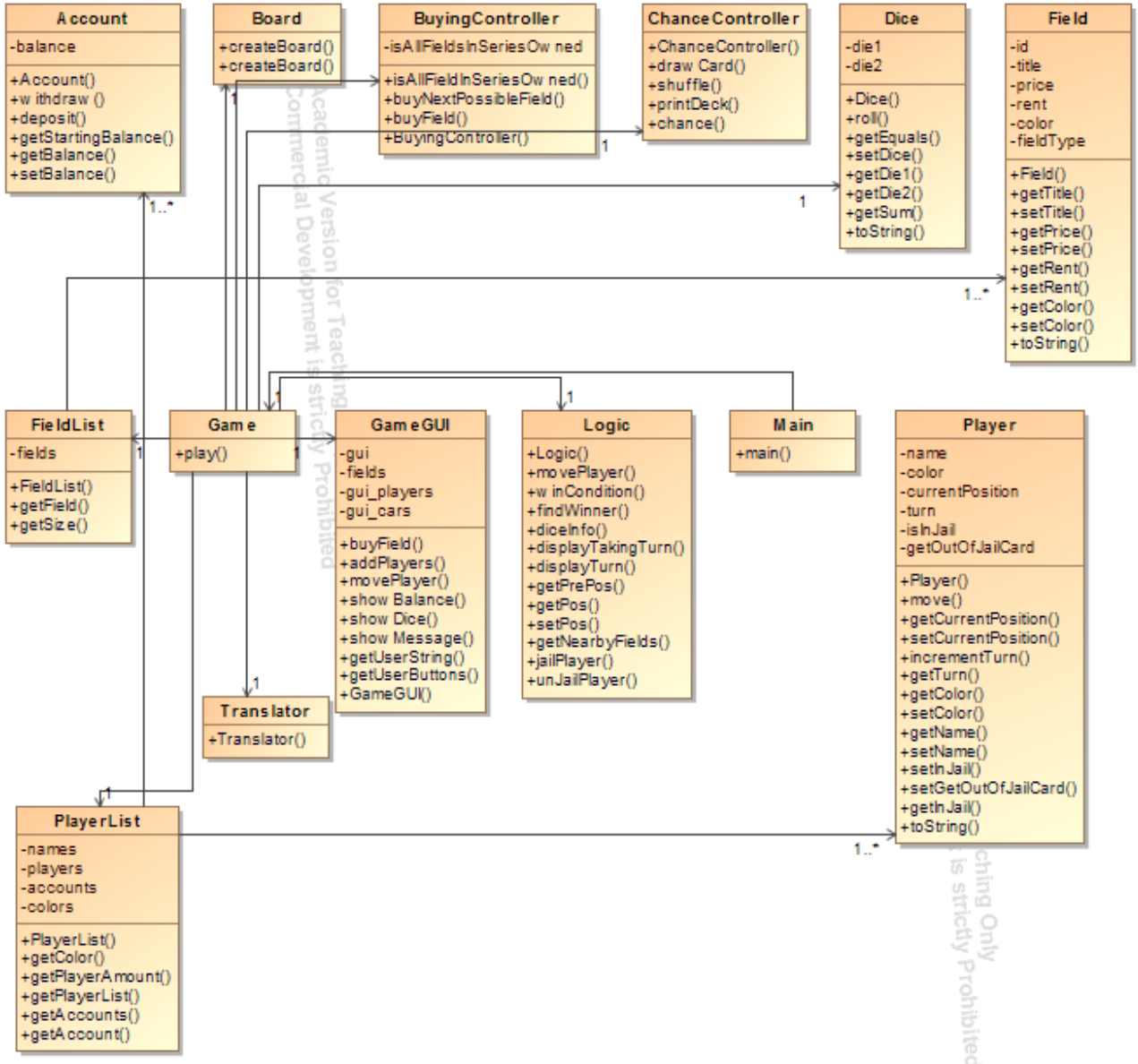
5.4.11 Translator

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

6 Design

6.1 Designklassediagram

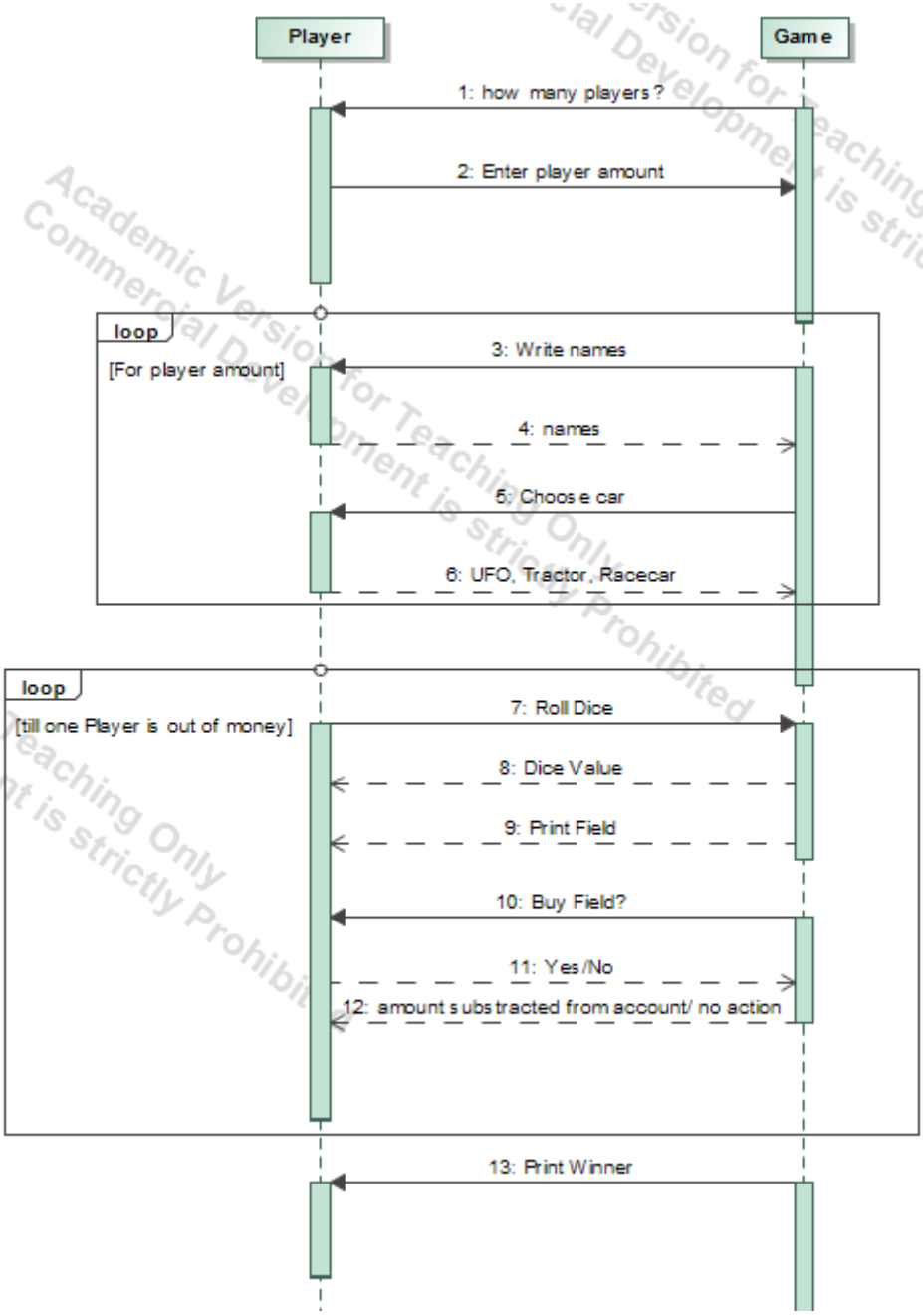
Designklassediagrammet (figur 3) viser hvilke metoder de enkelte klasser skal indeholde. Metoderne er blevet defineret ud fra deres ansvarsområder og beskrivelser i analysen.



Figur 3: Designklassediagram over spil

6.2 Systemsekvensdiagram

Systemsekvensdiagram (figur 4) illustrere sekvensen af metodekald for at spille spillet. Sekvensdiagrammet viser dog kun et enkelt system (hoveddelen) af spillet. For at udføre nogle af de metoder som bliver kaldt på "Game"bliver der brugt yderligere metoder fra andre klasser f.eks. "Logic"



Figur 4: Systemsekvensdiagram over kerne "loopet"

6.3 Sekvensdiagram

Sekvensdiagrammet viser hvordan hele programmet hænger sammen, med metodekald på tværs af klasser.

7 Implementering

Spillet er blevet implementere i Java, der er et objektorienteret programmeringssprog. Klasserne og metoderne er blevet defineret og implementeret udfra domænemodellen (figur 2 side 9), designklassediagrammet (figur 3 side 12), og sekvensdiagrammet (figur).

7.1 Game

I "Game"klassen er der hvor spillet kører (while loop). Loopet kører indtil der er blevet fundet en vinder udfra "Logic"klassens vinde betingelse.

```
for (int i = 0; i < playerAmount; i++) {
    while(!logic.winCondition(pl)) {
```

Figur 5: "While loop"indtil en vinder er fundet

Det er også at Chancekort bliver trukket til spillerne fra chancekort dækket(klassen).

```
if (logic.landedOnChance) {
chance.chance(pl, fl, playerTurn, logic, gameGui);
}   if (logic.drawAnother) {
    chance.chance(pl, fl, playerTurn, logic , gameGui);
    }
```

Figur 6: Hvis spilleren er landet på et chancefelt træk da et chancekort

7.2 Logic

Logic indeholder alt logikken i spillet dvs. de metoder som bliver brugt i "Game"klassen til f.eks. at flytte spillerne, købe felter, eller sætte spilleren i fængsel. Derudover holder klassen også styr på hvad der skal til for at vinde spillet og for at finde vinderen af spillet. Klassen bruges også til at skrive bestemt information ud i konsollen.

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

Figur 7: Kode der beskriver hvordan man vinder

7.3 Field

"Field"klassen er en abstrakt klasse, dvs. at den ikke er implementeret. Klassen er abstrakt da der i spillet bliver anvendt mange forskellige slags felter, der mere eller mindre har de samme egenskaber, dog med enkelte variationer. Der er derfor mulighed for at oprette felt klasser der passer speficikt til den type der skal laves. "Field"klassen indeholder en række "standard"metoder som f.eks. en konstruktør (figur 8) og en toString() metode (figur 9).

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

Figur 8: Field klassens konstruktør

```
@Override
public String toString() {
    return "Field{" +
            "title='" + title + '\'' +
            ", price=" + price +
            ", description='" + description + '\'' +
            ", color=" + color +
            ", fieldType='" + fieldType + '\'' +
            '}';
}
```

Figur 9: Field klassens toString metode

8 Dokumentation

8.1 Arv (Oliver)

8.2 Abstract (Jonathan)

Her kan man dokumentere vores Field klasse som er abstract.

8.3 Dokumentation for overholdt GRASP

Til at udvikle programmet har vi anvendt GRASP mønstre. I afsnit 4 står der beskrevet at der bliver anvendt Skaber-mønster, lav-kobling og høj sammenhørighed. Skaber mønsteret er som tidligere beskrevet, hvilken klasse der har ansvar for at lave nye klasser. I spillet er "Skaber"klassen Game, da den skaber alle andre klasser i spillet, direkte og indirekte gennem andre klasser(som den har skabt). Lav kobling er klasser der afhænger meget lidt af andre klasser. Dette er f.eks. gældende for "Dice-klassen som kan anvendes uden nogen andre klasser. Det samme gør sig gældende for "Account-klassen, "Translator-klassen, "Player-klassen osv. Høj sammenhørighed er klasser der har et veldefineret og fokuseret ansvarsområde. Dette gør sig gældende i klasser som "Player", "Dice"og "Translator"hver enkelt af de klasser har deres eget veldefineret ansvarsområde i spillet, og tager sig kun af de opgaver.

8.4 Override (Gustav)

Fortæl hvad det hedder hvis alle fieldklasserne har en landOnField metode der gør noget forskelligt.

9 Konfiguration

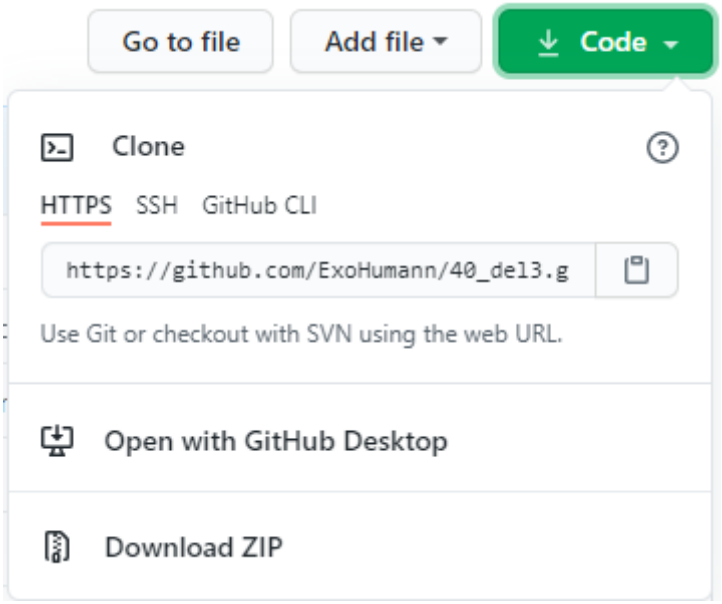
9.1 Versionsstyring

Under udviklingen af spillet er der blevet anvendt versionsstyring i form a git og et "repository"på Github. Repository'et kan findes her: `https://github.com/ExoHumann/40_del3` På github kan den seneste version af spillet findes.

Der er derudover også blevet anvendt versionsstyring til dokumentationen(dette dokument) her er der også blevet anvendt git og Github. Repository'et kan findes her: `https://github.com/joha413/CDIO3-rapport` Det er her at den seneste version af dokumentationen kan findes.

9.2 Hent og byg spillet

Når man gerne vil hente og bygge spillet med Intellij IDEA(*IntelliJ IDEA* 2020) Åbner man Intellij og trykker: File →New →Project from Version Control. Derefter kopiere man git repository url ind fra github (figur 10). Derefter trykker man Clone



Figur 10: Github repository klon værktøjer

Når projektet er åbnet i Intellij trykker man på den grønne hammer(eller ctrl+f9) for at bygge projektet.

10 Test

For at kunne dokumentere at spillet opfylder kravene fra (afsnit 4), er der blevet lavet en række unit test i JUnit. testene viser om de metoder/klasser opfylder de krav som er blevet defineret at de skal kunne, for at leve op til kundens ønsker.

11 Projektplanlægning

12 Konklusion

Figurer

Tabeller

Litteratur

Budtz, Christian m.fl. (nov. 2020). *diplomit-dtu/Matador_ GUI*. original-date: 2015-02-17T09:51:38Z. URL: `https://github.com/diplomit-dtu/Matador_GUI` (bes. 16.11.2020).

IntelliJ IDEA (2020). *IntelliJ IDEA: The Java IDE for Professional Developers by JetBrains*. en. URL: `https://www.jetbrains.com/idea/` (bes. 17.11.2020).

13 Bilag