CDIO-3

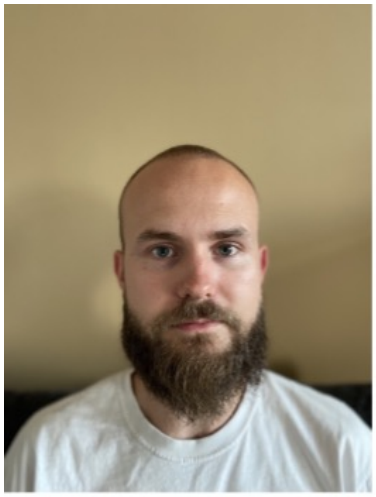
Fag: 02312 / 62531 / 62532
Gruppe 4

Afleveringsfrist: November 2021

Andreas Aagaard,
S215830

Alexander Elsing,
s216248

Alex Batten,
s211436



Marcus Jacobsen,
S215799

Nikolaj Beier,
S215814

Resumé

Til denne CDIO opgave, laver vi endnu et spil. Denne gang er det Monopoly Junior, som skal laves. De funktionelle krav er udarbejdet ud fra de udleverede regler til spillet og et billede af spillepladen. Vi lykkedes med at lave et fungerende Monopoly Junior spil, som lever op til størstedelen af de funktionelle krav såsom køb af ejendomme, betaling af husleje og indsamling af penge ved start feltet. Vi har accepteret, at vi ikke kunne få alle med, da det også var tilladt til denne opgave. De mest fundamentale krav er også testet med unit tests og dokumenteret i vores test cases.

Vi benyttede use case-, sekvens- og klasse-diagrammer i analyse- og designfasen i UML notation som del af en agil arbejdstilgang, for at simulere et ægte arbejdsmiljø. Fører man tankegangen videre, kan de tidligere CDIO opgaver kan også ses som første iterationer af dette endelige produkt.

1 Timeregnskab

Dato	Aktivitet	Medlemmer	Start	Slut	Timer
Uge 1					
01/11-2021	Møde om overblik over mål for ugen	Alle	15:30	17:00	1:30
02/11-2021	Start på analysedel (Diagrammer, krav og use cases påbegyndes	Alle	9:00	12:30	3:30
03/11-2021	Fortsat arbejde på analysedel	Alle bortset fra Marcus	12:30	16:00	3:30
04/11-2021	Analysedel færdiggøres	Alle	11:00	17:00	6:00
05/11-2021	Møde for at få overblik, og tilddele opgaver til weekenden	Alle	13:30	14:00	0:30
Uge 2					
08/11-2021	Møde om overblik over mål for ugen	Alle bortset fra Alexander	15:30	17:00	1:30
09/11-2021	Kodedelen af basisspillet startes, opgaver uddeles	Alle bortset fra Alexander	10:00	12:30	2:30
10/11-2021	Fortsat kodning af basisspillet	Alle Bortset fra Marcus & Alexander	12:30	15:00	2:30
11/11-2021	Basisspil færdiggøres	Alle bortset fra Alexander	10:00	16:00	6:00
12/11-2021	Møde for at få overblik, og tilddele opgaver til weekenden	Alle bortset fra Alexander	13:30	14:00	0:30
Uge 3					
15/11-2021	Møde om overblik over mål for ugen	Alle	15:30	17:00	1:30
16/11-2021	Ekstraopgaver til spillet påbegyndes, opgaver uddeligeres	Alle	10:00	12:30	2:30
17/11-2021	Fortsat kodning af uddeligerede ekstraopgaverne	Alle bortset fra Marcus	12:30	15:00	2:30
18/11-2021	Fortsat kodning af uddeligerede ekstraopgaverne	Alle	10:00	17:00	7:00
19/11-2021	Møde om overblik over mål for ugen	Alle	13:30	14:00	0:30
Uge 4					
23/11-2021	Rapportskrivning og fortsat kodedel	Alle	10:00	12:30	2:30
24/11-2021	Rapportskrivning og kodedel færdiggøres	Alle	12:30	15:00	2:30
25/11-2021	Rapportskrivning	Alle	10:00	15:00	5:00
26/11-2021	Projektet tjekkes efter fejl	Alle	13:30	14:00	0:30
I alt					52:30

Figure 1: Timeregnskabstabel

Indholdsfortegnelse

1 Timeregnskab **3**

2 Indledning **4**

3 Projektplanlægning **4**
3.1 Maven 4

4 Overvejelser **4**

5 Analyse **4**
5.1 Use Case 4
5.2 Use Case Diagram 6
5.3 Krav 6
5.4 Kravprioritet 7
5.5 Domæne 7
5.6 Systemsekvensdiagram 8

6 Design **9**
6.1 Sekvensdiagram 9
6.2 Klassediagram 9

7 Implementering **10**
7.1 Java 10
7.2 Maven 14

8 Test **14**
8.1 Test Cases 14
8.2 Exploratory Testing 16
8.3 JUnit 16

9 Konfiguration **16**

10 Versionsstyring **16**

11 Planlægning af tid **18**

12 Konklusion **18**

13 Bilag **20**
13.1 Spørgsmål 20
13.2 Vision 20
13.3 Gruppekontrakt 21

14 Kilder **22**

2 Indledning

Vores firma har denne gang modtaget en opgave om at bygge Monopoly Junior, der indeholder alle de regler, vi kan nå at implementere. Vi skal derfor tage stilling til hvilke krav er af højest prioritet, og fokusere på disse først. Spillet benytter, ligesom de forrigere CDIO opgaver, en udleveret GUI. Den hjælper brugeren med at holde styr på scoren og se hvad der foregår i spillet. Vi benytter også nogle af klasserne fra forrige projekt. Rapporten her dokumenterer vores analyse- design- og implementeringsfase, hvor vi finpudser arbejdsmetoden fra tidligere CDIO opgaver. Desuden indeholder den dokumentation for en række tests, som bekræfter at spillet lever op til kravene.

3 Projektplanlægning

3.1 Maven

Vi har til dette projekt valgt at bruge Maven som vores framework til de eksterne biblioteker vi bruger, som i dette tilfælde er GUI'en. Maven giver os også en mere simpel og effektiv måde at kompilere koden til UTF-8, hvilket er den verison af Java vi har valgt at der skal bruges.

4 Overvejelser

I starten af projektet gjorde vores gruppe sig en masse overvejelser, om hvordan vi skulle gå til opgaven, samt hvilke mål og forventninger vi havde til projektet. Den første ting vi blev enige om, var at vi kunne bruge mange af de ting vi skrev i CDIO-2, til at udforme spillet i CDIO-3. Vi kunne dog hurtigt se at det ikke var muligt at genbruge metoderne i deres aktuelle tilstand, men ting som spillerklassen mm. var oplagt at bruge som de var. Da vi i vores tidligere CDIO, havde relativt få klasser, samt få men lange metoder, planlagde vi derfor at bruge mange flere klasser. Tilsvarende blev vi også enige om at metoderne skulle være mindre, og at der samtidigt skulle være flere. På den måde kunne vi gøre koden til spillet meget mere overskueligt.

En anden stor prioritet vi havde til koden, var at vi skulle begynde at bruge arvede klasser. Dette gjorde det muligt at korte meget ned på koden, da det gjorde at vi kunne genbruge koden i klasserne uden at skulle skrive det igen.

5 Analyse

5.1 Use Case

I vores analyse af den udleverede opgave, har vi identificeret en række use cases. De vigtigste use cases er: kast terninger og ryk brik - dem gruppérer vi sammen og kalder *Tag tur*. Herudover vurderer vi, at spilleren får lov til at vælge antal spillere når spilleren starter spillet. I UML ville vi skrive at en use case *Start Spil* inkluderer *vælge antal spillere*.

Herudover er der en række use cases afhængigt af hvilket felt, man lander på. Disse use cases *extender* use casen *Tag tur*. Vi kunne også kalde dem sub-use cases. De er alle i listen forneden.

Til sidst skal systemet slutte spillet, når summen af en spillers bankbeholdning er 0 eller under. Spilleren med den højeste mængde penge er vinderen. Dette er en use case, som skiller sig ud fra de andre, da det er selve systemet, der udfører det. I dette tilfælde vil systemet være en sekundær aktør.

Brief

Vi opstiller use cases i *brief* format.

UC1: Start spil.
UC2: Vælg antal spillere.
UC3: Tag tur.
UC4: Køb felt.
UC5: Betal husleje.

UC6: Tag chancen.
UC7: Gå i fængsel.
UC8: Besøg fængsel.
UC9: Passerer start.
UC10: Slut spil.

Fully Dressed

Nu udvider vi **UC3** til *fully dressed* format.

UC3: Tag tur

Rækkevidde (Scope) Matadorspil

Niveau : Brugermål

Primær aktør : Spiller

Interessanter

- Spiller : Spilleren er interesseret i at slå med to terninger, der er fair. Spilleren ønsker også at spillet giver og tager de rigtige mængder penge afhængig af hvor de lander.

- Kunde : Kunden er interesseret i et produkt der kan køres på maskinerne i databarerne samt et sæt funktionelle krav (Se kravspecifikation)

Forudsætninger

- Programmet er åbnet.

- Spilleren har startet spillet og valgt antal spillere. **UC1** & **UC2**

Success Garanti

- Spilleren spiller sin tur færdig og det bliver næste spillers tur.

Main Scenario

1. Spilleren slår med terningerne.
2. Spilleren rykker brikken frem svarende til terningernes værdi.
3. Spilleren agerer afhængigt af hvor den lander og om den har passeret start **UC9**.
   Hvis spilleren lander på et normalt felt, som er ejet af en anden, betales huslejen **UC5**. Hvis feltet ikke er ejet, har spilleren muligheden for at købe det **UC4**. Andre muligheder er gå i fængsel-felt **UC7**, besøg fængsel-felt **UC8**, chance-felt **UC6**

Specielle Krav

- 2-4 spillere spiller spillet. - Spillet kommer med tekst feedback til spillerne, så de ved hvor i spillet de er.

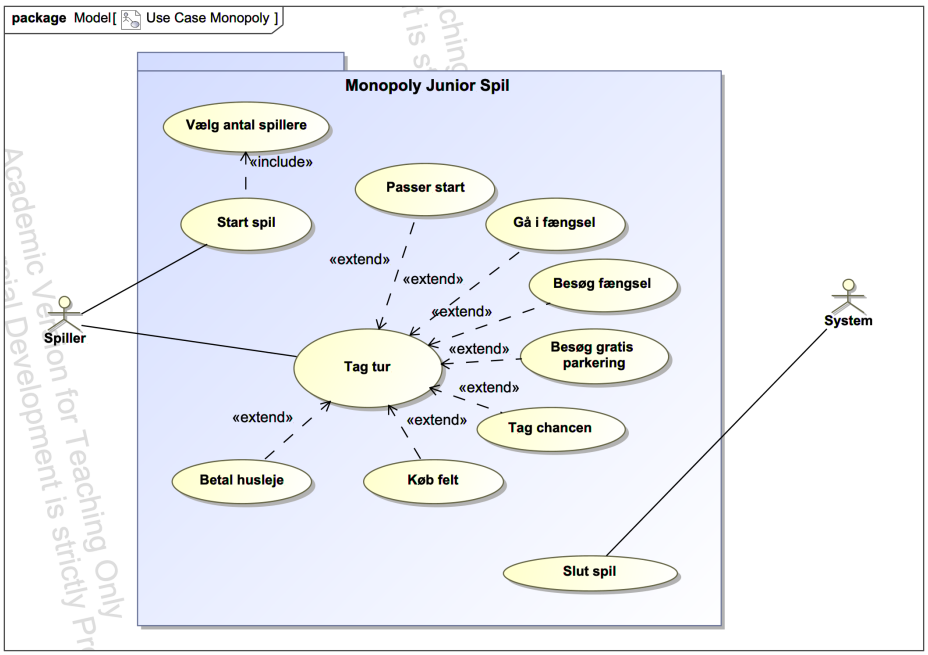
Teknologi og Data Variationer N/A

Hyppighed

- Spillerne skiftes på tur.

Andet (Herunder åbne problemer) N/A

5.2 Use Case Diagram



Figur 2: Use case diagram

5.3 Krav

Funktionalitet
De funktionelle krav til spillet findes i det udleverede Monopoly Junior regelsættet.[2]

- 2-4 spillere skal kunne spille spillet
- For 2 spillere får hver spiller 20M, for 3 spillere 18M, for 4 spillere 16M
- Spillerne skiftes til at slå terningerne
- Yngste spiller starter
- Spillerne har en pengebeholdning.
- Spillere skal lande på et felt og gå videre derfra
- Der skal være en spilleplade med 24 felter [3]
- Når en spiller ikke kan betale, slutter spillet og en vinder kåres. Vinderen er spilleren med flest penge.
- Spillere modtager penge hver gang de passerer start
- Spillere skal købe ikke-ejede felter, som de lander på
- Spillere skal betale husleje på ejede felter, som de lander på. Huslejen svarer til feltets pris
- Huslejen fordobles, når en spiller ejer begge felter i samme farve.
- Når en spiller lander på chancekort, trækkes et chancekort og chancekortets effekt går ud over spilleren.
- Når en spiller lander på *gå i fængsel*, skal spilleren i fængsel. Man passerer ikke start.
- Når en spiller er i fængsel skal den betale 1M på dens næste tur. Hvis den har et *Kom ud af fængsel*-chancekort, skal dette benyttes i stedet.
- Når en spiller lander på *gratis parkering*, sker der ikke noget.

Usability

- Github commit for hver times kodning eller ved hvert fuldførte delopgave.
- Spillet skal være på 1 sprog.

Reliability

- Tests skal med i projektmappen.

Performance

- Skal kunne køres uden bemærkelsesværdig forsinkelser (vi bruger kravet fra sidste på 333 ms)

Supportabillity

- Genbruger terningen fra sidste projekt.
- Genbruger spiller-klassen fra sidste projekt.
- Skal kodes i java

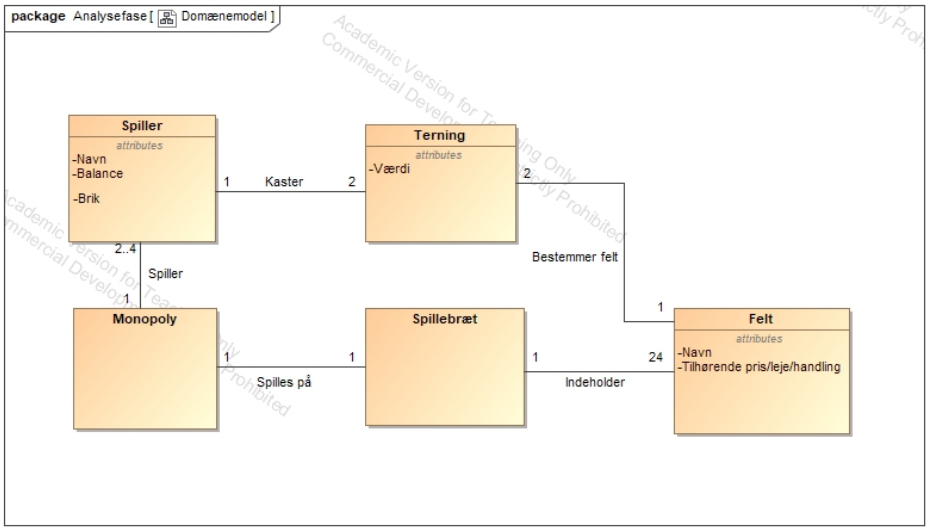
5.4 Kravprioritet

Til opgaven er vi blevet bedt om at prioritere reglerne og det er ikke et krav at implementere dem alle. For at hjælpe os med overblikket over vores prioriteter, bruger vi værktøjet MoSCoW. Vi rangerer derfor alle de funktionelle krav herunder. Akronymet står for:
Mo - Must Have S - Should Have Co - Could Have W - Won't Have

Must Have	1.	2-4 spillere skal kunne spille spillet
	2.	Spillerne skiftes til at slå terningerne
	3.	Spillerne har en pengebeholdning
	4.	Spillere skal lande på et felt og gå videre derfra
	5.	Spillere skal betale husleje på ejede felter, som de lander på. Huslejen svarer til feltets pris
	6.	Spillere skal købe ikke-ejede felter, som de lander på
	7.	Når en spiller ikke kan betale noget, slutter spillet og en vinder kåres. Vinderen er spilleren med flest penge.
Should Have	8.	Spillere modtager penge hver gang de passerer start
	9.	Når en spiller lander på *gå i fængsel*, skal spilleren i fængsel
	10.	Når en spiller lander på *gratis parkering*, sker der ikke noget.
	11.	For 2 spillere får hver spiller 20M, for 3 spillere 18M, for 4 spillere 16M
Could Have	12.	Når en spiller lander på *chancekort*, trækkes et chancekort og chancekortets effekt går ud over spilleren.
	13.	Når en spiller er i fængsel skal den betale 1M på dens næste tur. Hvis den har et *Kom ud af fængsel*-chancekort, skal dette benyttes i stedet.
Won't Have	14.	Huslejen fordobles, når en spiller ejer begge felter i samme farve
	15.	Yngste spiller starter

5.5 Domæne

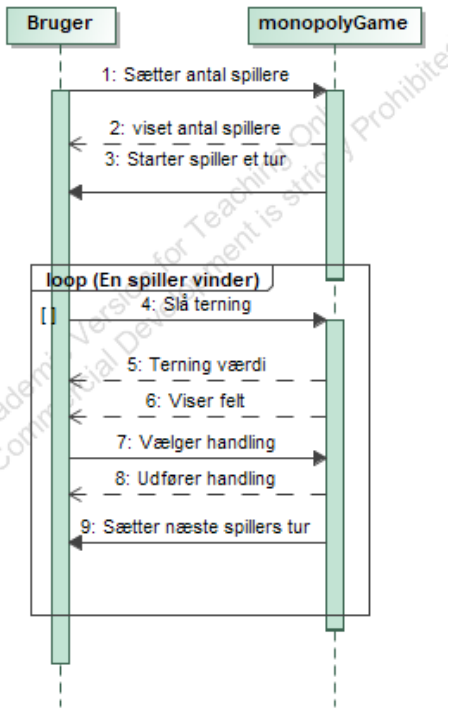
I vores domæne model identificeret de klasser som vi forventer at skulle anvende i vores diagram. Vi har valgt at inkludere felt, terning og spillebræt, da vi gerne vil designe spillet så disse objekter er synlige for spillerne. Dermed har vi valgt ikke at danne en specifik "brik" klasse, da vi mener at der ikke er nok fylde til at danne en hel klasse omkring brikkerne, derfor vil vi kan anvende brikken som en attribut under spiller, for at få information om hvor spilleren er placeret, og derefter håndteres visning af spillernes placeringer i spillebrættet.



Figur 3: Analyse-klasse diagram skitse

5.6 Systemsekvensdiagram

I systemsekvensdiagrammet er der en aktør (bruger), og et system (MonopolyGame). Spillet, starter med at en aktør vælger antallet af spillere. Systemet viser antalet af spillere, og sætter start saldoen. Derefter starter spillet spiller 1 tur. Under turen slår spilleren terningen, der rykker en brik i spillet. Systemet viser terningens værdi, og felt. Alt efter hvilket felt aktøren lander på, udfører aktøren en handling. spillet viser konsekvensen af spillerens handling, og sætter den næste spillers tur. Dette forsætter indtil at en spiller har mindre end 0m tilbage.

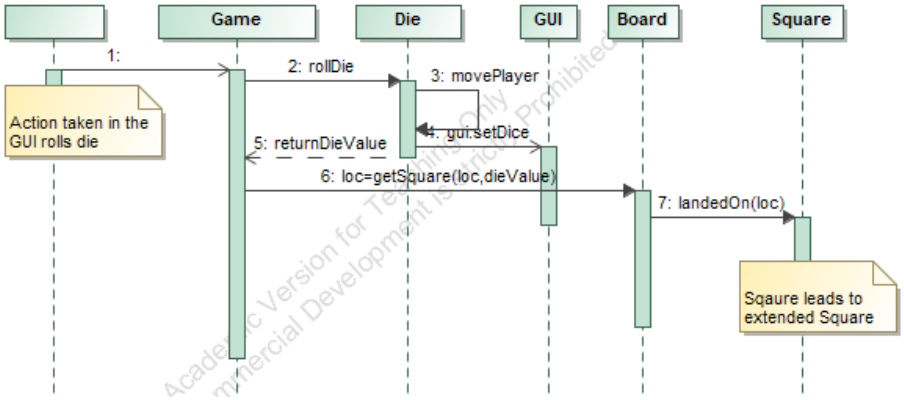


Figur 4: System sekvens diagram MagicDraw

6 Design

6.1 Sekvensdiagram

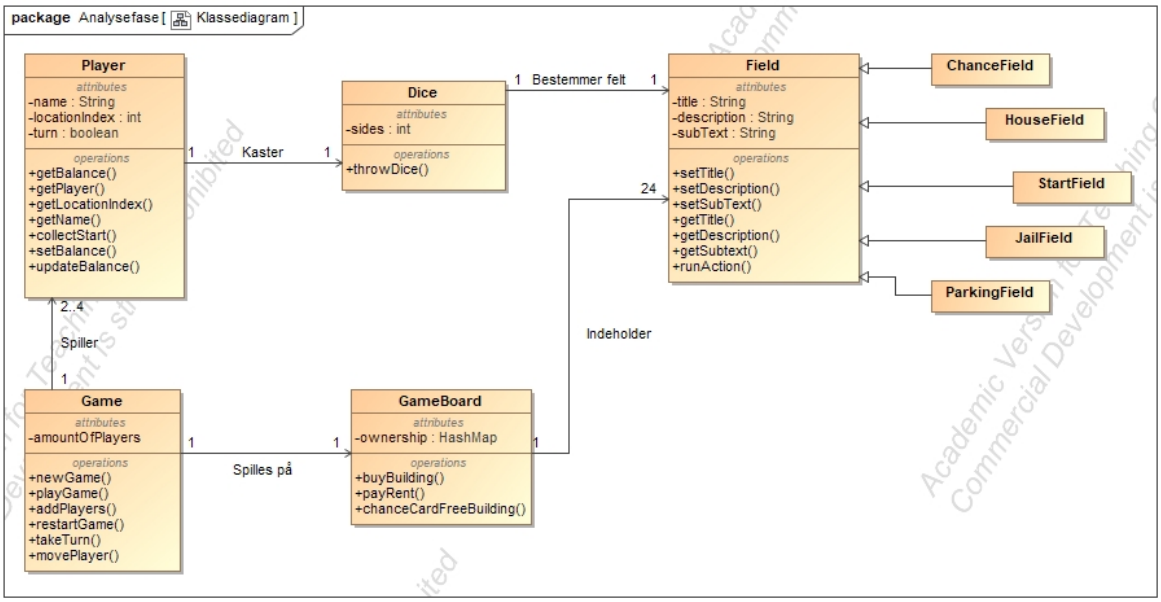
I sekvens diagrammet er vist en en spillers tur mellem rollDie og landedOn. spilleren starter med at rollDie igennem GUI'en dette kalder movePlayer og viser Gui.setDie i Guien. Systemet bruger loc.getSquareValue tli at rykke brikken, som derefter finder hvilken square der er landedOn og udfører den pågældende handling



Figur 5: Sekvensdiagram MagicDraw

6.2 Klassediagram

I vores klassediagram har vi anvendt de fundne klasser i analysefasen gennem domænemodellen og dermed fundet hvilke attributter og metoder som klasserne burde indeholde for at opnå vores Use case. Vi har valgt at opdele vores felter i deres forskellige korresponderende felttyper, da vi gerne vil have muligheden for forskellige interaktioner med brugeren alt efter hvilket felt de lander på. Det gør vi ved at sub-klasserne af felter har en super-klasse som de får nedarvet information fra. Med det kan vi modulere hver enkelt sub-klasse med *@Override* til at skabe forskellige interaktioner alt efter hvilket felt man er landet på, og dermed opnå høj sammenhængskraft da ansvaret bliver videredelegeret til sub-klasserne.



Figur 6: Klassediagram MagicDraw

7 Implementering

Til forskel fra tidligere CDIO opgaverne, har vi valgt at skrive koden og kommentarer i koden på engelsk. Dette er så koden er læselig for flere mennesker og for at undgå ASCII problemer med æ, ø og å.

7.1 Java

Efter nogle overvejelser omkring klasserne, use cases, krav og GRASP-elemnterne begyndte vi at programmere spillet.

Vi genbrugte *Player*, *Account* og *Dice* klasserne fra CDIO-2.

For at forbedre designkoncepterne fra CDIO-2, oprettede vi flere metoder og klasser. Heriblandt en klasse til spilbrættet (*Gameboard*) og felterne (*Fields*). Vi benytter arv og polymorphism til felterne. De er beskrevet mere forneden.

Når et *Gameboard*-objekt bliver instantieret, oprettes de 24 felter, der er på vores Monopoly Junior plade. Metoder ses på figurer 7 & 8

```
public GameBoard() { this.fields = this.createFields(); }
```

Figur 7: GameBoard konstruktør

```
private Field[] createFields() {
    return new Field[]{
            new StartField(),
            new HouseField( name: "Burgerbaren", description: "Burgerbaren", subText: "1", new Color( r: 65, g: 80, b: 30)),
            new HouseField( name: "Pizzeriaet", description: "Pizzeriaet", subText: "1", new Color( r: 65, g: 80, b: 30)),
            new ChanceField(),
            new HouseField( name: "Slikbutikken", description: "Slikbutikken", subText: "1", new Color( r: 173, g: 216, b: 230)
            new HouseField( name: "Iskiosken", description: "Iskiosken", subText: "1", new Color( r: 173, g: 216, b: 230)),
            new JailField( visit: true),
            new HouseField( name: "Museet", description: "Museet", subText: "2", Color.pink),
            new HouseField( name: "Biblioteket", description: "Biblioteket", subText: "2", Color.pink),
            new ChanceField(),
            new HouseField( name: "Skateparken", description: "Skateparken", subText: "2", Color.orange),
            new HouseField( name: "Swimmingpoolen", description: "Swimmingpoolen", subText: "2", Color.orange),
            new ParkingField(),
            new HouseField( name: "Spillehallen", description: "Spillehallen", subText: "3", Color.red),
            new HouseField( name: "Biografen", description: "Biografen", subText: "3", Color.red),
            new ChanceField(),
            new HouseField( name: "Legetøjsbutikken", description: "Legetøjsbutikken", subText: "3", Color.yellow),
            new HouseField( name: "Dyrehandlen", description: "Dyrehandlen", subText: "3", Color.yellow),
            new JailField(),
            new HouseField( name: "Bowlinghallen", description: "Bowlinghallen", subText: "4", Color.green),
            new HouseField( name: "Zoo", description: "Zoo", subText: "4", Color.green),
            new ChanceField(),
            new HouseField( name: "Vandlandet", description: "Vandlandet", subText: "5", Color.blue),
            new HouseField( name: "Strandpromenaden", description: "Strandpromenaden", subText: "5", Color.blue),
```

Figur 8: createFields metode

Dette array af *Fields* bliver brugt til at instantiere GUI'en i *Game* klassen. Dette gøres ved at kalde *getGuiFields* metoden, som looper igennem *Fields*-arrayet, og tager deres GUI-felt med *getGuiField* metoden. Dette nye *GUI_Fields*-array kan så bruges til at lave et GUI objekt i *Game*-klassen.

Nu hvor spilpladen var oprettet kunne vi begynde at kigge på flere funktionele krav. Det første var at slå med terningen og rykke brikken. Vi grupperede dette sammen i vores use cases og gør det samme i koden. Metoden kaldes *takeTurn* og ses i figur 15. Metoden kaster med terningerne, rykker spilleren og kalder *runAction* for det felt, spilleren er landet på. *movePlayer* metoden rykker spilleren på pladen og holder styr på en int værdi for hver spiller, der hedder *locationIndex*.

Når *locationIndex* overstiger 24 har spilleren taget en tur om pladen. Int værdien nulstilles og spilleren får 2M for at passere start.

runAction er en abstrakt metode, som ændres afhængigt af hvilket felt, spilleren er landet på. "Besøg Fængsel", "Gratis Parkering", og "Start" gør ikke noget. Husfelterne giver mulighed for at købe, og hvis de er købt skal spilleren betale huslejen til ejeren.

```
/**
 * Player takes a turn.
 */
private void takeTurn(Player player){
    this.gui.getUserButtonPressed( msg: "kast", ...buttons: "kast");

    int dice = this.dice.kastTerning();
    this.gui.setDie(dice);
    movePlayer(player,dice);
    this.gameboard.getFields()[player.getLocationIndex()].runAction(player, game: this);
}
```

Figur 9: takeTurn metode

For at holde styr på vores købte bygninger anvender vi et HashMap, som bliver styret i *GameBoard*. Et HashMap gør det muligt for os at holde al data om ejede bygninger og hvilke spillere der ejer hvad samlet, ved at et HashMap anvender *Keys*, som hver har sin egen korresponderende *Value*. Det vil sige at vi eksempelvis kan kalde om vores HashMap har en *Key*, som indeholder værdien for felt 6. Hvis vores HashMap har en *Key* på felt 6, betyder det at feltet er opkøbt. Dermed har den *Key* en *Value*, som er feltets ejer. Med dette, kan vi anvende HashMappet til vores metoder *isBought*, *BuyBuilding* og *PayRent*.

```
public boolean isBought(HouseField houseField){
    return ownership.containsKey(houseField);
}
```

Figur 10: isBought Metode

Ved *isBought*, tjekker vi om vores HashMap indeholder det specifikke felt, som vi er landet på. Hvis værdien returnerer sand, betyder det at HashMappet indeholder feltet og dermed er feltet købt, hvorimod hvis den returnerer falsk er feltet ikke købt. Metoden bliver anvendt således at når man lander på et HouseField, bliver det først tjekket om feltet er købt eller om det er til salg. Når der er blive fundet ud af om feltet er til salg eller ej, anvender vi *BuyBuilding* metoden, hvis feltet er til salg, og *PayRent* hvis det allerede er opkøbt og der dermed skal betales husleje.

```
if (gameBoard.isBought( houseField: this)) {
    gameBoard.PayRent( field: this, player, game);
    System.out.println("Paid rent");
} else {
    gameBoard.BuyBuilding( field: this, player, game);
}
```

Figur 11: runAction i HouseField

Vores *BuyBuilding* metode, virker ved at et if-statement tjekker om hvorvidt spilleren har nok penge til at købe feltet uden at gå i 0. Hvis dette er sandt trækker vi pengene fra spilleren og indsætter feltet som en *Key* i vores HashMap samt spilleren som *Value* til korrosponderende *Key*, ved at anvende HashMappets *.put(Key, Value)* metode

```
public void BuyBuilding(HouseField field, Player buyer, Game game){
    currentFieldValue = Integer.parseInt(field.getSubText());
    if(currentFieldValue < buyer.getSaldo()) {
        game.getGui().getUserButtonPressed( msg: "This building isn't bought. You have enough money to buy it!",  ...buttons: "Buy");
        ownership.put(field, buyer);
        buyer.saldoOpdatering(-(currentFieldValue));
    }else {
        game.getGui().getUserButtonPressed( msg: "This building isn't bought. You don't have enough money to buy it",  ...buttons: "Continue");
    }
}
```

Figur 12: BuyBuilding Metode

Hvis feltet allerede er købt, anvender vi derimod *PayRent* metoden. Metoden virker ved at vi gennem HashMappet finder hvilken spiller der ejer det felt som man er landet på. Derfra lægger vi feltets værdi til ejeren af feltet og trækker værdien fra spilleren som er landet på det ejedede felt.

```
public void PayRent(HouseField field, Player rentPayer, Game game){
        Player owner = ownership.get(field);
        if(owner!=rentPayer) {
            currentFieldValue = Integer.parseInt(field.getSubText());
            String str = "This building is owned. You have to pay " + currentFieldValue + "M to "+ owner.getNavn();
            game.getGui().getUserButtonPressed(str,  ...buttons: "Okay");
            owner.saldoOpdatering(currentFieldValue);
            rentPayer.saldoOpdatering(-(currentFieldValue));
        }
}
```

Figur 13: PayRent Metode

Under chance kort var der femten forskelige chance kort, som lå under otte forskelige mulige handlinger. Vi vil gå igennem de vigtigste her. Det første chance kort rykkede spilleren direkte til start og giver to m til spileren.

```
game.getGui().displayChanceCard( txt: "Move to start and receive 2M");
game.setPlayer(player,  indexLocation: 0);
player.collectStart();
ChanceCardNumber++;
```

Figur 14: Move to start

En større mængde af chancekortene var opbygget på måden ryk til en af flere felter hvis dette felt ikke er købt for man det gratis, ellers skal man betale husleje. Her brugte vi chanceFreeBuilding, der tjekkede om feltet var krævet eller ej. Hvis feltet var krævet brugte man noget ligende PayRent som vist længere oppe, ellers fik man bygningen gratis.

```
public void chanceFreeBuilding(HouseField field, Player buyer, Game game){
    currentFieldValue = Integer.parseInt(field.getSubText());
    if(isBought(field)) {
        game.getGui().getUserButtonPressed( msg: "The free building is already claimed, you have to pay rent",  ...buttons: "Pay rent");
        Player owner = ownership.get(field);
        if(owner!=buyer) {
            currentFieldValue = Integer.parseInt(field.getSubText());
            owner.saldoOpdatering(currentFieldValue);
            buyer.saldoOpdatering(-(currentFieldValue));
        }
    }else {
        ownership.put(field, buyer);
        game.getGui().getUserButtonPressed( msg: "This free building isn't claimed. It's yours for free!",  ...buttons: "Claim");
```

Figur 15: ChanceFreeBuilding

Chance kort koden er for alt for uoverskueligt, men grundet tidsrestriktioner var der ikke tid til at lave en klasse hvori at alt koden ville være lavet til metoder. Det er en klart et punkt der kan laves bedre til en fremtidig opgave.

Der var også endnu et muligt chance kort hvori at alle spilere skulle give en m til spilleren der trak det men det blev fravalgt grundet tidsrestriktioner

Oversætter

Som noget ekstra har vi i vores program valgt at lave en oversætter klasse, som tillader os nemt at kunne oversætte hele spillet til et andet sprog. Dette har vi gjort ved hjælp af to metoder, skrivFil som laver en tekst fil med programmets strenge og Læsfil som tager tekst filen som input og laver det til en ArrayList kaldet for liste. Ved hjælp af skrivFil metoden har vi skabt en engelsk version af alle strengende samt en dansk version, hvorved vi kan erstatte de strenge vi har i alle andre klasser ved hjælp af Læsfil arrayet. Måden det ville fungere på er ved at skrive liste.get(stringnummer) til hver de tilhørende strenge i de klasser og indstille hvilken tekst fil der skal indlæses. Hvis nu man ville lave det sådan at brugeren nemt kunne skifte mellem sprog kunne man evt. vælge at lave en knap i GUI'en med en simpel boolean.

Arv

Som det ses i vores design-klassediagram er de forskellige feltklasser tæt forbundet og faktisk deler de så mange attributter og metoder, at vi kan sætte dem til at nedarve fra en fælles feltklasse. Dermed er vores *Field* klasse også en creator for sub-klasserne, da den indeholder de værdier som sub-klasserne arver. Den har den nødvendige initial iserende data for at sub-klasserne kan anvendes i systemet.

Arv, eller nedarvning er når en sub-klasse udvider en super-klasse. Der kan godt eksistere flere sub-klasser til en super-klasse, men i Java kan klasser ikke have flere super-klasser.

I vores projekt er der en række felt-klasser, som alle har nogle fælles træk. De er del af spillepladen (GUI'en), man kan lande på dem, og de har et navn, farve og beskrivelse. Derfor kan man sige de nedarver fra en fælles feltklasse. De udvider også feltklassen, da husfelterne for eksempel har en husleje og kan have en ejer. I og med at vi deler vores felter op i forskellige klasser opnår vi høj sammenhængskraft og lav kobling, da vi uddelegere ansvaret for at håndtere *landOnField* til de enkelte klasser og dermed opnår vi også mere modularitet i systemet. Videreudvikling af projektet kunne eventuelt være at håndtere flere typer felter, hvilket nemt kunne indsættes ved at vi har denne lave kobling og høje sammenhængskraft i vores felt-klasser, da ansvaret for *landOnField* bliver kørt individuelt, og der afhænges ikke af information fra andre sub-klasser, men kun af information fra nedarvningen af super-klassen.

Abstract

Hvis en super-klasse aldrig instantieres og kun bruges til at gruppere sub-klasser sammen, kaldes det en abstract. I vores tilfælde er *Fields*-klassen en abstract.

Service klasse

Vi har anvendt *Account* klassen, som en service klasse for spiller. Det har vi gjort for at anvende Pure Fabrication, da *Player* klassen blev for stor efter vores mening, så vi tilføjede *Account* klassen for at fjerne noget af ansvaret for klassen og dermed opnå lav kobling og høj sammenhængskraft.

Polymorphism

Når en spiller rykker rundt på brættet og lander på forskellige felter er der forskellige konsekvenser afhængigt af hvilke felter, man lander på. En måde at løse dette på vil være at lave en stor switch case, som tjekker hvor spilleren er landet og agerer derudfra. Dette er dog ikke god programmeringsskik, da det hurtigt kan blive meget uoverskueligt. I stedet bruges polymorphism. Ordet betyder, "At have mange forme". I Java kan det betyde at et objekt betragtes anderledes afhængig af hvilken situation det bliver kaldt i. Det kunne for eksempel være en metode *landOnField*, der gør noget forskelligt afhængigt af hvilket felt, der er landet på. Ved hjælp af nedarvning har felttyperne forskellige *landOnField* metoder. Dette kaldes sub-type polymorphism.

7.2 Maven

Til projektet bruges GUI'en fra de tidligere projekter. Ligesom til sidste projekt, bruger vi igen Maven til at køre vores GUI som en dependency.

Vi ønskede ydermere at bygge en JAR-fil, så spillet kunne køre på databarens computere. Vi besluttede med erfaring fra sidste projekt, at Maven ville være en god løsning til dette. Maven ville tillade os at byggeJAR filen til den ønskede Java version. Vi stødte dog på problemet med depencencies i Maven, og kunne ikke bygge en JAR fil, der havde alle dependencies. Derfor er man nød til at køre programmet igennem en IDE, som IntelliJ.

8 Test

Løbende har vi testet spillet på to forskellige måder. En White Box metode og en Black Box metode.

8.1 Test Cases

På næste side har vi opstillet en række test cases til at bekræfte, at vi lever op til kravene i kapitel 5.3.

TEST CASE ID	TEST SCENARIE	TEST CASE	TEST PROCEDURE	RELATEREDE KRAV	FORVENTET RESULTAT	STATUS
TC1	Start spil	Antallet af spillere	Kør spil og vælg 2-4 antal spillere. Tjek hvor mange spillere er med.	2-4 spillere skal kunne spille spillet	2-4 spillere initialeres	Bestået
TC2	Start spil	Beholdningen	Start spil Vælg 2-4 antal spillere Tjek spillernes beholdning	For 2 spillere får hver spiller 20M, for 3 spillere 18M, for 4 spillere 16M	Ved valg af 2 spillere: 20M. 3: 18M. 2: 16M	Bestået
TC3	Spilleren passerer start	Beholdningen	Start spil Ryk en spiller 25 felter Tjek spillerens beholdning	Spillere modtager penge hver gang de passerer start	Spillerens beholdning inkrementerer med 2M.	Bestået
TC4	Landet på husfelt	Køb felt	Spiller lander på et uejet felt Spiller køber feltet hvis den har beholdningen til det	Spillere skal købe ikke-ejede felter, som de lander på	Spillerens mister prisen af feltet fra sin beholdning. Spilleren er herefter ejer	Bestået
TC5	Landet på husfelt	Betal leje	Spiller lander på et ejet felt Spiller mister lejen fra beholdning Ejeren modtager lejen	Spillere skal betale husleje på ejede felter, som de lander på. Huslejen svarer til feltets pris	Spilleren mister lejen fra sin beholdning Ejeren modtager lejen i sin beholdning	Bestået
TC6	Spillet slutter	Slut spil	En spillers beholdning går under 0 Spilleren med højest beholdning kåres som vinder	Når en spiller ikke kan betale, slutter spillet og en vinder kåres. Vinderen er spilleren med flest penge.	Spillet slutter Korrekt vinder kåres	Bestået
TC7	Spiller i fængsel	Kom ud af fængsel kort	Spilleren er landet i fængsel Spilleren mister sit kom ud af fængsel kort og kan rykke som normalt på næste tur	Når en spiller er i fængsel skal den betale 1M på dens næste tur. Hvis den har et Kom ud af fængsel-chancekort, skal dette benyttes i stedet.	Spilleren rykker normalt på næste tur. Spilleren mister sit kom ud af fængsel kort	Ikke testet
TC8	Spiller i fængsel	Betal ud af fængsel	Spilleren er landet i fængsel Spilleren mister 1M og kan rykke som normalt på næste tur	Når en spiller er i fængsel skal den betale 1M på dens næste tur. Hvis den har et Kom ud af fængsel-chancekort, skal dette benyttes i stedet.	Spilleren rykker normalt på næste tur. Spilleren mister 1M fra beholdningen	Ikke testet
TC9	Brugertest	Usability / Sprog	En person gennemgår produktet og undersøger om sproget præsenteret for dem er det samme. Bruger laver en streg på papir for hver gang sproget er anderledes	Spillet skal være på 1 sprog.	Ingen forskelle, samme sprog (engelsk)	Bestået

Figur 16: Test Cases

Testcase ID	TC9
Resume	Tester usability i form af om sproget er det samme igennem hele programmet.
Krav	Spillet skal være på 1 sprog.
Precondition	Programmet er åbnet og kører, tester er informeret omkring hvad de skal teste. Har et stykke papir og en blyant.
Postconditions	Tester har noteret på papiret antal fejl
Test fremgangsmåde	En person gennemgår produktet og undersøger om sproget præsenteret for dem er det samme. Bruger laver en streg på papir for hver gang sproget er anderledes, som antages som en fejl.
Test data	0
Forventet resultat	Ingen forskelle, samme sprog (engelsk)
Resultat	Ingen forskelle
Status	Bestået
Testet af	Test forsøg: Emma Frandsen Test leder: Alex Batten
Dato	26-11-2021
Testmiljø	IntelliJ IDEA 2021.2.1 (Ultimate Edition) Build #IU-212.5080.55, built on August 24,2021 --- Windows 11 Home 21H2 Build 22000.318

Figur 17: Formel Test Case TC9

8.2 Exploratory Testing

Den hyppigste form for test, vi har foretaget var en Black Box test, hvor vi har kørt spillet og set om det lever op til de funktionelle krav. Dette kaldes også exploratory testing. Selve testen kræver ikke nogen indsigt i koden og er derfor god til at finde bugs løbende. I praksis har vi foretaget sådan en test for cirka hver metode tilføjet.

Denne form for test har givet os en code coverage på: class: 100%, method: 90%, line: 80%. Dette er meget tilfredsstillende og med et par forsøg i slutningen af projektet har vi konkluderet at næsten 90% af programmet derfor er fri for bugs.

De resterende 10% af koden, som ikke bliver berørt er kode, som vi godt kan konkludere virker. Det er for eksempel et andet antal spillere, end det der er valgt. Det er også når spilleren har nogle valg, som *restart game*. Udover det findes der nogle getters og setters, som ikke bruges.

8.3 JUnit

Hver gang vi var i mål med et væsentligt funktionelt krav har vi skrevet en test case og testet med JUnit test. Dette er en form for White Box testing, hvor vi kontrollerer spillets gang og ser om det agerer som forventet. Vi har valgt at bruge denne type test til vores formelle test cases. For eksempel testes **TC3** med en JUnit test, hvor spillet startes og en spiller rykkes 25 felter frem. Så tjekkes spillerens beholdning. Eksempel ses i figur 18

Med vores JUnit tests opnåede vi en coverage på: class: 91%, method: 73%, line: 31%. Dette er meget tilfredsstillende og godt indefor opgavebeskrivelsens krav. De linjer kode vi ikke dækkede med JUnit testing er primært *ChanceField* klassen, da det var en af de sidste klasser vi tilføjede og derfor ikke formelt fik testet. Vi accepterer dog dette faktum da det ikke var et af vores højest prioriterede krav.

```
/**
 TC3
 */
@Test
void movePlayerStart() {
    Game game = new Game();
    game.addPlayers( n: 2,game.getGui());
    game.startBalance( startMoney: 16);
    int expectedFirstValue = 16;
    assertEquals(expectedFirstValue,game.getPlayer( index: 1).getSaldo());
    game.movePlayer(game.getPlayer( index: 1), dice: 25);
    int expectedSecondValue = expectedFirstValue+2;
    assertEquals(expectedSecondValue,game.getPlayer( index: 1).getSaldo());
}
```

Figur 18: JUnit test af TC3

9 Konfiguration

Vores projekt er bygget i maven hvilket jo var et krav til denne opgave. Maven bruges til at definere projektets dependencies, herunder GUI biblioteket og junit pakken. Udover det har vi også valgt maven til at tvangs kompilere til Java 8, hvilket er den ældste version af Java vi har kunne finde på databarernes computere.

10 Versionsstyring

Git

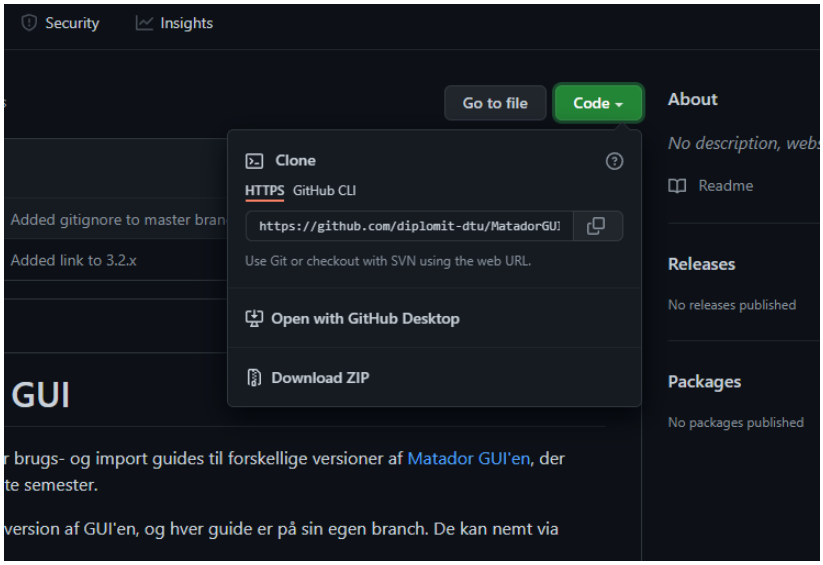
Vores projekt er versioneret med git, hvilket er et værktøj som har tilladt os at arbejde hver især på forskellige dele af vores projekt samtidigt. f.eks. har vi for flere individuelle funktioner af programmet haft en separat branch, hvor vi bekymringsfrit har kunne kode uden at påvirke vores allerede fungerende dele.

Vores git repository har vi haft hostet på GitHub så vores projekt altid kan hentes over internettet hvis noget skulle gå tabt. Afsnittet under dette forklarer forskellige måder at kunne hente git repositoriet fra GitHub, samt import af repositoriet fra en lokal maskine.

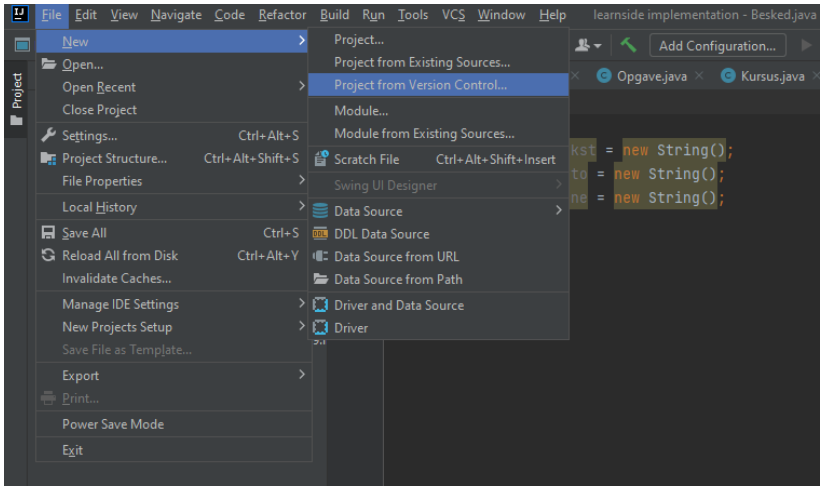
Vejledning til import af git repository:

Metode 1: Importere repository fra GitHub

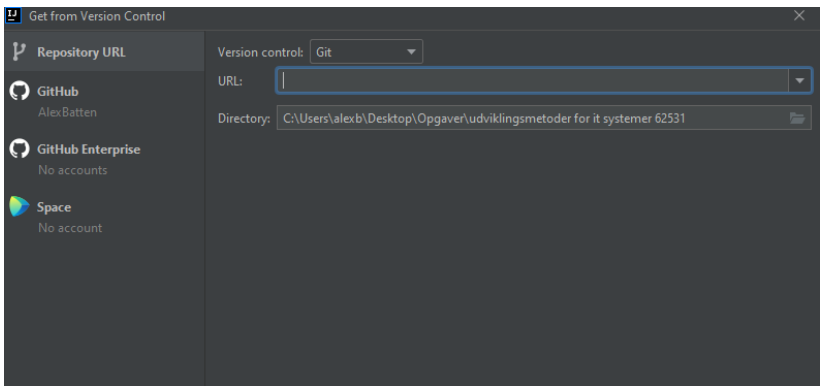
1. Find git repository url fra GitHub. Du kan finde et "Clone" url under "Code" på det projekt du gerne vil importere.



2. Åben nu intellij, gå under file, new, også den der hedder Project from Version Control:



3. Her fra vil du kunne indsætte git url'en, samt vælge stien til hvor projektet skal gemmes på din maskine.



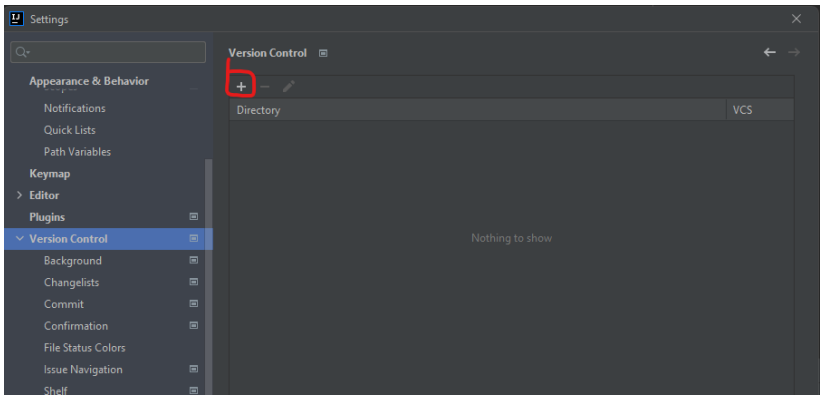
4. Til sidst trykker du på clone nede i højre hjørne.

Metode 2: Importere lokalt repository

Alternativt hvis du har et intellij projekt med et lokalt git repository på din maskine du gerne vil importere ind i intellij er dette også ret simpelt,

1. Gå under file, open, herfra vælger du rodmappen til dit projekt 2. Her burde intellij give en besked omkring configuration af VCS da den kan se .git mappen, hvis ikke:

2a. Gå under settings -> Version Control, specificer .git mappen i projektet ved at trykke på plus.



2b. Gå under VCS -> Enable Version Control Integration. Derefter burde repoet være åbnet i projektet.

11 Planlægning af tid

Vi har i dette CDIO projekt aftalt i starten af hver uge, hvilke dage vi skulle arbejde på projektet, alt efter hvornår gruppemedlemmerne havde tid til at arbejde på det. Samtidigt har vi holdt "fredagsmøder" hvor vi alle kom med input om hvad vi har lavet i ugen, fået overblik over hvor langt vi var kommet, samt et overblik over hvor meget vi skulle have lavet i ugen efter.

Vores arbejde var opdelt i 3 dele, som er analyse, kode og rapportskrivning. Ligeledes delte vi også 4-ugers perioden vi har haft til at lave projektet op i 3 dele, hvor analysedelen udgjorde den første uge, kodedelen udgjorde de to midterste uger og den sidste uge, rapportskrivning.

12 Konklusion

Til dette projekt har vores firma fået til opgave at lave et Monopoly-junior spil, og færdiggøre så mange regler som vi kunne nå, men samtidigt have et spil som man kan spille. Vi har derfor kigget på hvilke regler der var et krav for at spillet virker, og derefter bygget på det. Vi arbejdede med kravene i den rækkefølge vi synes var vigtigst, og har næsten lavet alle reglerne, og bygget et færdigt spil. Til dette spil har vi genbrugt mange af de gamle elementer fra sidste spil, dog modificeret mange af dem så de passer ind på hvad målet med dette spil er.

Vi startede med at analysere vores opgavebeskrivelse og arbejdede med hvordan spillet skulle laves. Ved hjælp af alt det forarbejde vi lavede, kunne vi identificere hvilke klasser der skulle bruges, hvilket samtidigt gav os mulighed for at begynde på at kode selve spillet.

Vi har igen, ligesom sidst brugt Maven som framework, og har også denne gang lavet en runnable JAR-fil. Maven har gjort det muligt at rykke GUI'en ind i JAR-filen, hvilket har gjort at man ikke skal have GUI'en hentet ned på egen computer men at den kan bruges gennem JAR-filen. Til spillet har vi også gjort brug af JUnit testing, hvilket har sikret os at alle tingene i koden virker som det skal. Som test har vi også kørt programmet igennem selv for at sikre os at tingene vi ikke har testet via JUnit tests virker, og at de funktionelle krav var opfyldt.

Vi har til sidst gået projektet igennem, og blevet enige om at vi har nået det mål vi satte os med at komme så mange regler i spillet igennem som det var muligt at nå. Vi har derfor konstateret at spillet virker og at det var klar til at blive afleveret.

13 Bilag

13.1 Spørgsmål

Spørgsmål 1: Under analyse og design dokumentation står der "Et eksempel på system sekvensdiagram" og "Et eksempel på sekvensdiagram". Vil det sige de ikke behøver at være baseret på projektet / være fuldførende? Eller skal de forstås som bare at tage et med ud af flere?

Svar: de diagrammer i vedhæfter skal være baseret på projektet, men i kan godt nøjes med at lave diagrammer for en mindre del af projektet

Spørgsmål 2: Er der et fængsel, og hvis der er hvordan kommer man så ud af fængslet?

Svar: fængsel == Restrooms i reglerne

Spørgsmål 3: Skal der railroads med, da de ikke er på brættet men ikke i reglerne?

Svar: Brug denne spildeplade som udgangspunkt (det er de engelske regler)

https://drive.google.com/file/d/15oSUaFK5NtryM21fVlUwhGYGYiCACye7/view0

13.2 Vision

I skal udvikle et Monopoly Junior spil. Vurder hvad der er det vigtigste for at spillet kan spilles! Implementer de væsentligste elementer for at spillet kan spilles. I må gerne udelade regler - prioritér! Nu har vi terninger og spillere på plads, men felterne mangler stadig en del arbejde. I dette tredje spil ønsker vi derfor at forrige del bliver udbygget med forskellige typer af felter, samt en decideret spilleplade. Spillerne skal altså kunne lande på et felt og så fortsætte derfra på næste slag. Man går i ring på brættet. Der skal nu være 2-4 spillere.

13.3 Gruppekontrakt

Gruppekontrakt

Projekt og titel	**CDIO Projekt - Matadorspil**
Gruppemedlemmer	Alex Batten, Marcus Jacobsen, Alexander Elsing, Andreas Aagaard, Nikolaj Beier
Linje	Softwareteknologi 02312 / 62531 / 62532

Gruppens aftaler

1. Ambitionsniveau for – *samarbejdet* – *kvaliteten* – *tidsforbruget* – *andet*	1: Vi arbejder altid sammen om opgaverne 2: Altid højest mulige kvalitet 3: Indtil vi er færdige 4:
2. Møder – *Mødeplan/dagsorden*	- Morgenbøn - Morgenmøde på cirka 10 min hvor vi lægger en plan over dagens mål
3. Studie- og arbejdstider – *Hvor meget skal vi arbejde om ugen* – *Lektier fra gang til gang* – *Hvor ofte mødes vi*	1: Arbejde i skolen, og derhjemme 2: Samarbejde og samtale via Discord 3: Forskelligt, men opfordring til fysisk møde (Primære dage ville være tirsdag (morgen), onsdag samt torsdag over discord hvis der er behov)
5. Kommunikation uden for skolen – *Telefonnumre* – *Facebookgruppe* – *Andet*	1: Facebook gruppe: De faglige fem 2: Discord Samtale Telefonnumre: Alex Batten: 29924081 Andreas Aagaard: 42516860 Alexander Elsing: 60137143 Marcus Jacobsen: 31729624 Nikolaj Beier: 29840232
6. Opbevaring af dokumenter – *F.eks. Googledocs, Dropbox e.a.*	1: Google Docs / Drev bruges til opbevaring af vores dokumenter
7. Fremgangsmåde hvis noget går galt – *hvis aftaler ikke overholdes* – *hvis én person er syg i længere tid* – *hvis nogen bliver uvenner* – *andet*	For hver 10 minutter du kommer for sent betales der 10 kroner til gruppe pengekassen, desuden hvis du kommer for sent flere gange i træk skal du også lægge bøden fra sidste gang oven i. Hvis du kommer til tiden resettes din bødeliste. Pengene skal bruges efter sidste eksamensdag hvor vi holder fest. Gyldig fraevar / forsinkelse diskuteres i gruppen. 1: Samtale om problemet (Ved sygdom, andre aftaler etc.)

Gruppekontrakt

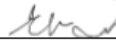
	2: Ikke noget med at blive uvenner. Tal sammen om problemet og løs det i fællesskab. På den måde undgår man at forsinke opgaven
8. Andre aftaler for samarbejdet – *herunder evt. regler for inddragelse af vejleder.*	1: Almen arbejde og høflighed er påkrævet

Underskrifter:

Dato: 14-09-21 Navn: Alex Batten

Dato: 14-09-21 Navn: Alexander Kiel Elsing

Dato: 14-09-21 Navn: Andreas Aagaard s215830 

Dato: 14/09-21 Navn:  Marcus Jacobsen

Dato: 14/09-21 Navn: Nikolaj Beier

14 Kilder

References

[1] Craig Larman, *Applying UML and Patterns*, Pearson Education, Limited, 3. Udgave, 2004, ISBN: 9780131489066.

[2] Monopoly Junior regelsæt:
docs.google.com/document/d/13fSwzPAN2rliVBn$_e BgwEVITwAXYIZVR3lZ3oPzo2_0$

[3] Monopoly Junior felter:
https://drive.google.com/file/d/15oSUaFK5NtryM21fVlUwhGYGYiCACye7/view