



KURSER:

02312 02313 02315

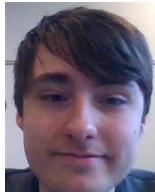
CDIO3 - Monopoly Junior



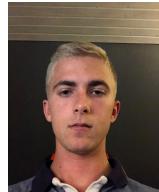
Jeppe Kaare Larsen
s170196



Anton N. Ranløv
s170429



Mathias Skøn Frimann
s163310



Nicklas Beyer Lydersen
s185105



Rasmus Søborg
s185119



Hans Krogh
s185110

30. november 2018

Ansvarsområder

| Diverse | Navn |
|-----------------|-------------------------|
| Indledning | Jeppe, Anton og Nicklas |
| Latex opsætning | Jeppe, Anton og Nicklas |
| Konklusion | Jeppe |
| Korrektur | Jeppe |

| Analyse | Navn |
|-----------------------|------------------|
| Krav og kravanalyse | Anton |
| Use Cases | Anton og Nicklas |
| Risiko Analyse | Mathias |
| Navneordsanalyse | Jeppe og Nicklas |
| Domænemodel | Hans og Nicklas |
| System Sekvensdiagram | Nicklas |

| Design | Navn |
|---------------------|-----------------|
| Arkitektur | Jeppe |
| Designklassediagram | Alle |
| Sekvensdiagram | Nicklas |
| GRASP-Mønstre | Jeppe og Rasmus |

| Implementering | Navn |
|--|-----------------------------------|
| Overvejelser | Jeppe, Anton og Mathias |
| Dokumentation | Rasmus |
| Nedarvning og abstrakt-klasse | Rasmus (Kode indsætning: Nicklas) |
| Gennemgang af koden | Jeppe, Anton og Mathias |
| Versionsstyring og konfigurationsstyring | Jeppe og Nicklas |
| Javadoc | Alle |

| Test | Navn |
|------------------------|----------------|
| Brugerinteraktionstest | Jeppe og Anton |
| JUnit | Hans |

Indhold

| | |
|--|-----------|
| Ansvarsområder | 1 |
| Indledning | 4 |
| Hovedafsnit | 5 |
| 1 Analyse | 5 |
| 1.1 MoSCoW Analyse | 5 |
| 1.2 Krav | 5 |
| 1.3 Use-case-diagram | 7 |
| 1.4 Brief Use Cases | 8 |
| (BUC1) RollDie: | 8 |
| (BUC2) PickUpChanceCard: | 8 |
| (BUC3) Buy/Sell/PayRent: | 8 |
| 1.5 Casual Use Cases | 8 |
| (CUC1) RollDie: | 8 |
| (CUC2) PickUpChanceCard: | 8 |
| (CUC3) Buy/Sell/PayRent: | 8 |
| 1.6 Fully Dressed Use Case | 9 |
| 1.7 Risiko Analyse | 10 |
| 1.8 Navneordsanalyse / Definering af klasser | 12 |
| 1.9 Domænemodel | 13 |
| 1.10 System Sekvensdiagram - lav nyt diagram | 14 |
| 2 Design | 15 |
| 2.1 Arkitektur | 15 |
| 2.2 Designklassediagram | 16 |
| 2.3 Sekvensdiagram | 17 |
| 2.4 GRASP-mønstre | 19 |
| 3 Implementering | 20 |
| 3.1 Overvejelser | 20 |
| Kravændringer | 20 |
| Brug af tidligere CDIO-kode | 20 |
| 3.2 Dokumentation | 20 |
| 3.3 Nedarvning og abstrakt-klasser | 21 |
| 3.4 Gennemgang af GameControllereren | 22 |
| GameController | 22 |
| 3.5 Versionsstyring | 23 |
| 3.6 Konfigurationsstyring | 24 |

| | |
|--|-----------|
| 4 Test | 25 |
| 4.1 Test-cases | 25 |
| 4.2 White box test | 26 |
| JUnit | 26 |
| Code coverage | 27 |
| 4.3 Black box test | 28 |
| Brugerinteraktionstest | 28 |
| 5 Konklusion | 29 |
| 5.1 Evaluering | 29 |
| Bilag | 30 |
| Bilag 1 - Anvendte værktøjer | 30 |

Indledning

Der er blevet stillet til opgave at udvikle et Monopoly Junior-spil med en fungerende GUI, hvor spilleren visuelt kan se en spilleplade, og hvor figuren befinner sig på spillepladen. Det skal være muligt for spillene at kunne se, hvad de ejer på spillepladen samt deres balance.

Kundens vision (fra opgavebeskrivelsen):

"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."

Hovedafsnit

1 Analyse

Der er gjort brug af forskellige analysemetoder for at optimere arbejdsprocessen og kvaliteten af programmet ”Monopoly Junior”. Der bliver i dette kapitel beskrevet de vigtigste artifikater gennem analysefasen. Hovedfokuspunkter har været kravspecifikation, use cases, domænemodel og system sekvensdiagram.

1.1 MoSCoW Analyse

Kundens krav deles op i fire underpunkter, for at få et overblik over hvilke krav der vil blive implementeret i denne version, samt hvilke krav der findes nødvendige for at programmet virker optimalt.

- Krav markeret med (M) er must krav. Dvs. at kunden har specificeret disse krav, og at de skal implementeres i programmet.
- Krav markeret med (S) er should krav. Dvs. at de specificerede krav burde implementeres, men de er ikke nødvendige for programmet.
- Krav markeret med (C) er could krav. Dvs. at de specificerede krav kunne blive implementeret i programmet, men de er kun idéer og er ikke en højt prioritet.
- Krav markeret med (W) er won’t krav. Dvs. at det er krav, som ikke bliver implementeret i denne version af programmet.

1.2 Krav

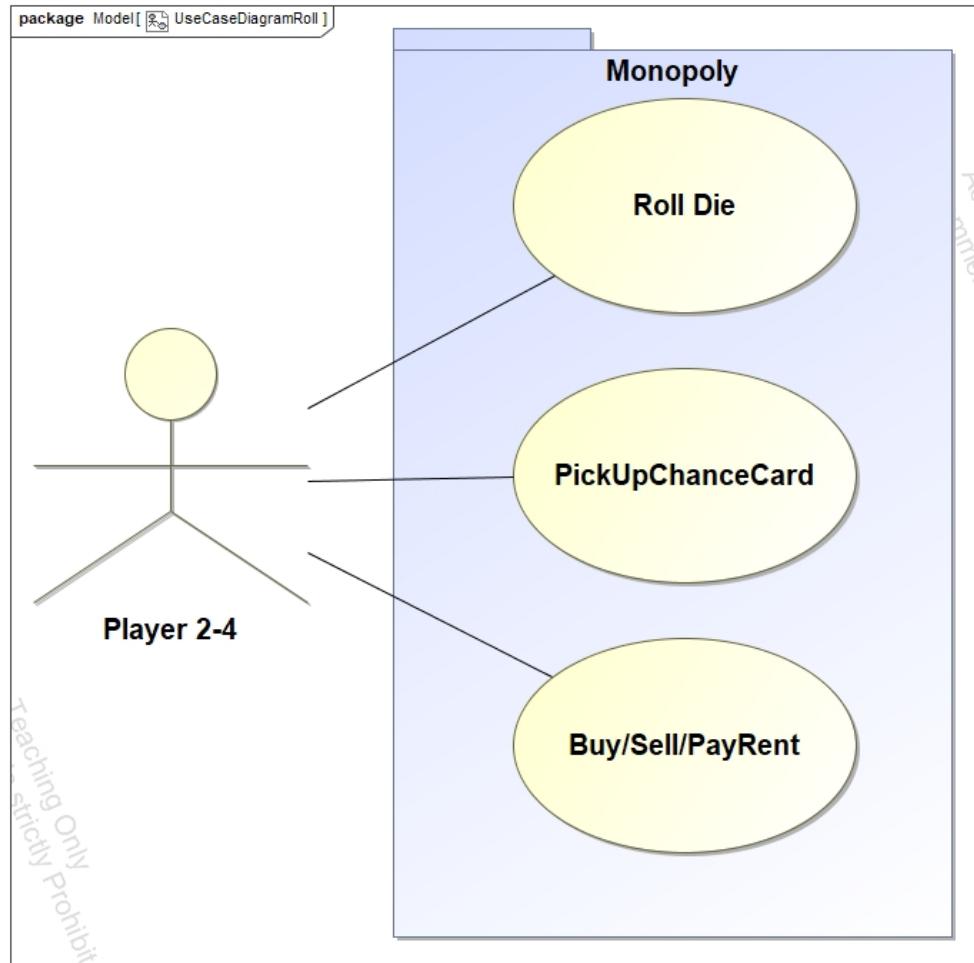
Her er kundens krav skrevet op, og der er lavet MoSCoW-analyse på hvert krav

- (M1) Der skal være en decideret spilleplade som kan blive set visuelt.
- (M2) Spillerne skal kunne lande på et felt og fortsætte derfra på næste slag. Man går i ring på brættet.
- (M3) Der skal nu kunne være 2-4 spillere.
- (M4) Der skal være passende konstruktører, get og set metoder og `toString` metoder.
- (M5) Der skal være en klasse der hedder `GameBoard`, der kan indeholde alle felterne i et array.
- (M6) En `toString` metode der udstriber alle felterne i arrayet.

- (M7) Der SKAL benyttes GUI denne gang.
- (M8) Der skal være 3 testcases med tilhørende testscripts og testrapporter og en JUnit test til centrale metoder.
- (M9) Der skal også være minimum en brugertest. Brugeren skal ikke kunne kode.
- (M10) Spillet skal opfylde følgende kriterier:
Regler, Chancekort og Spilleplade
- (S1) Forklar hvad arv er.
- (S2) Forklar hvad abstract betyder.
- (S3) Fortæl hvad det hedder hvis alle fieldklasserne har en landOnField metoder, der gør noget forskelligt.
- (S4) Dokumentation for test med screenshots.
- (S5) Dokumentation for overhold GRASP. testscripts og testrapporter og en JUnit test til centrale metoder.
- (C1) Anvend såvidt muligt CDIO1 og CDIO2 klasser til opgaven, for at minitere unødvendigt arbejde.

1.3 Use-case-diagram

Use case-analysen er valgt for at finde frem til aktørernes interaktion med de forskellige use cases. Dette projekt benytter sig af 3 use cases, som bliver skitseret i use case diagrammet figur 1.1.



FIGUR 1.1 - USE CASE DIAGRAM

1.4 Brief Use Cases

Der gøres brug af brief use cases til at beskrive vores 3 use cases fra figur 1.1. Dette er valgt for at få et hurtigt overblik over vores 3 use cases.

(BUC1) RollDie:

Spilleren slår med en virtuel terning og bliver rykket hen på det nye felt.

(BUC2) PickUpChanceCard:

Spilleren trækker et chancekort ud af de 24 og gør hvad der står.

(BUC3) Buy/Sell/PayRent:

Spilleren lander på et felt. Spilleren køber feltet, sælger feltet eller gør ingen ting eller betaler husleje.

1.5 Casual Use Cases

Herunder er de 3 use cases beskrevet mere detaljeret, for at få et endnu større overblik over, hvad der forgår i use case'ne.

(CUC1) RollDie:

Spilleren kaster med en virtuel terning, hvor spillet så genere en terningsum. Med terningesummen rykker spilleren så frem til et nyt felt.

(CUC2) PickUpChanceCard:

Spilleren lander på chancekort feltet, og skal nu trække et kort. Der er 24 forskellige chancekort som spilleren kan trække, som alle gør noget forskelligt, hvor spilleren så skal gøre hvad chancekortet beskriver.

Se chancekort reglerne for yderligere beskrivelse af de individuelle kort.

(CUC3) Buy/Sell/PayRent:

Spilleren lander på et felt som indeholder en bygning/forlystelse, som afgør:

Feltet er ikke ejet af nogle, spilleren kan købe hvis spilleren har råd. (Negativ)

Feltet er ejet af spilleren selv, spilleren kan vælge at sælge ellers sker der ingen ting. (Neutral)

Feltet er ejet af en anden spiller, spilleren betaler husleje til den anden spiller. (Negativ)

1.6 Fully Dressed Use Case

Her er en use case beskrevet så detaljeret som muligt. Dette giver et mere systematisk overblik over denne use case.

| (FDUC1) Monopoly Junior - RollDie | |
|--|---|
| Scope: | Monopoly Junior program. |
| Level: | User goal. |
| Primary Actor: | Spilleren. |
| Stakeholders and Interests: | Spiller: Spilleren vil gerne nemt kunne se hvad der sker når terningen bliver rullet. |
| Preconditions: | Spillet skal være startet og det skal være en spillers tur. |
| Success Guarantee: | Terningen bliver kastet og den givne spillerens brik bliver rykket antal felter svarende til antallet af øjne på terningen. |
| Main Success Scenario: | |
| 1. Spilleren kaster med en terning | |
| 2. Systemet genere et tilfældigt tal | |
| 3. Spilleren observere terningens øjne. | |
| 4. Spillerens brik bliver rykket antallet af øjne på terningen | |
| Extensions: | Spiller har tabt spillet og kan derfor ikke slå med terningen og rykke deres brik. |
| Special Requirements: | Fremvis grafisk metode. |
| Technology and Data Variations List: | Java Program |
| Frequency of Occurrence: | Hver gang spilleren har en tur |
| Miscellaneous | |

1.7 Risiko Analyse

Risikoanalysens formål er at vægte alt, hvad der kan gå galt og komme med både forebyggende og afbødende løsninger. Konsekvens er på en skala fra 1 - 5, hvor 1 er trivielt og 5 er alvorligt. Sandsynlighed er på en skala fra 1 - 5, hvor 1 er sjældent og 5 er uundgåeligt. Risikobedømmelsen er givet ud fra produktet af de to vægtede værdier. Tilsvarende værdien af risikobedømmelsen er der en kategori af aktion.

$$\text{Risk Rating} = \text{Likelihood} \times \text{Severity}$$

| | Catastrophic | 5 | 5 | 10 | 15 | 20 | 25 |
|------------|--------------|------------|---------------|------------|----------|----------|----|
| S | Significant | 4 | 4 | 8 | 12 | 16 | 20 |
| e | Moderate | 3 | 3 | 6 | 9 | 12 | 15 |
| v | Low | 2 | 2 | 4 | 6 | 8 | 10 |
| e | Negligible | 1 | 1 | 2 | 3 | 4 | 5 |
| r | | | 1 | 2 | 3 | 4 | 5 |
| i | Catastrophic | | STOP | | | | |
| t | Unacceptable | | URGENT ACTION | | | | |
| y | Undesirable | | ACTION | | | | |
| | Acceptable | | MONITOR | | | | |
| | Desirable | | NO ACTION | | | | |
| Likelihood | | | | | | | |
| | | Improbable | Remote | Occasional | Probable | Frequent | |

(Reference: (02313) Fildeling/Hold-A Ian/Lektion 4, slide 18)

FIGUR 1.7 - RISK RATING

Matrixet har formålet at definerer de forskellige aktionskategorier og deres tilsvarende indvirkning på projektet.

Risiko analyse for Monopoly Junior

| | Hvad kan gå galt | Konsekvens | Sandsynlighed | Riskotal |
|----|---|------------|---------------|----------|
| 1. | Miskommunikation | 1 | 2 | 2 |
| 2. | Projektmedlemmer bliver forhindret i at møde op | 2 | 2 | 4 |
| 3. | Større ændringer undervejs | 3 | 2 | 6 |
| 4. | 3. parts kode (Maven) | 3 | 3 | 9 |
| 5. | Versionsstyrings konflikter | 4 | 4 | 16 |

| | Forebyggende (pre) | Afbødende (post) |
|-------|--|--|
| 1.7.1 | Opret fælles forum for idédeling og planlægning. | Hold møde og debrief. |
| 1.7.2 | Sørg for at alle dokumenter distribueres mellem gruppemedlemmer efter hver arbejdsdag. | Aftal fokus område som det fraværende gruppemedlem skal arbejde med derhjemme. |
| 1.7.3 | Sørg for at programmets struktur er versatilt nok til at understøtte ændringerne. | Foretag tests for at tjekke alle input-outputs på ældre metoder stadig virker. |
| 1.7.4 | Sørg for at den nyeste version af 3. parts koden bliver brugt. | Søg hjælp fra evt. Kursuslærer eller hjælpelærer. |
| 1.7.5 | Split programmet op i relevante klasser og objekter så GitHub ved hvordan programsstrukturen er og kan indsætte ændringer uden konflikter. | Manuelt flette programmet. |

Alle forebyggende aktioner har været en del af det iterative arbejde i løbet af projektet.

1.8 Navneordsanalyse / Definering af klasser

Der gøres brug af en navneordsanalyse til at finde navneord til videreudvikling i forhold til domænemodellen og artifikter i designfasen. Der tages udgangspunkt i de tre casual use cases.

(CUC1): *Spilleren kaster med en virtuel terning, hvor spillet så genererer en terningsum. Med terningesummen rykker spilleren så frem til et nyt felt.*

(CUC2): *Spilleren lander på chancekort feltet og skal nu trække et kort. Der er 24 forskellige chancekort, som spilleren kan trække, som alle gør noget forskelligt, hvor spilleren så skal gøre hvad chancekortet beskriver.*

(CUC3): *Spilleren lander på et felt som indeholder en bygning/forlystelse, som afgør: Feltet er ikke ejet af nogle, spilleren kan købe hvis spilleren har råd. Feltet er ejet af spilleren selv, spilleren kan vælge at sælge ellers sker der ingen ting. Feltet er ejet af en anden spiller, spilleren betaler husleje til den anden spiller.*

Navneordende markeret med gul og andre relevante navneord fundet ved brainstorm, er blevet listet herunder.

Relevante navneord fra use-casene, som kan blive til klasser i domænemodellen:

- Spiller (Player)
- Terning (Die)
- Spil (Game)
- Felter (Field)
- Chancekort (ChanceCard)
- Bygning/Forlystelse (FieldAction)

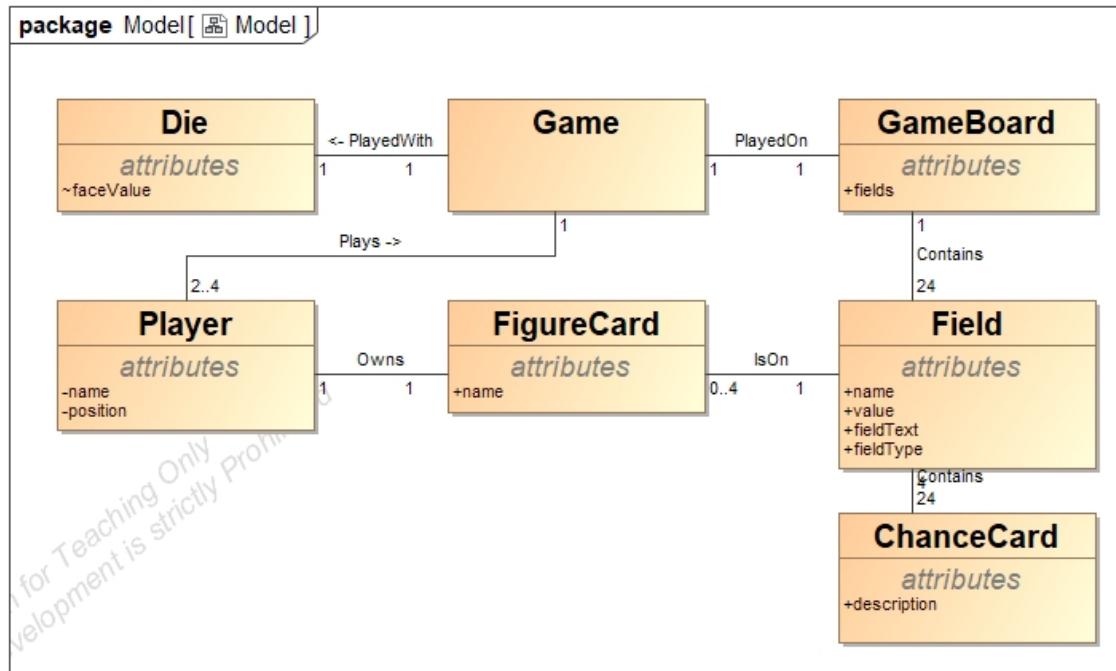
Andre relevante navneord, som kan blive brugt som klasser i domænemodellen:

- Figurkort (FigureCard)
- Spilleplade (GameBoard)

1.9 Domænemodel

Det er besluttet at arbejde ud fra principperne lav kobling og høj samhørighed. Dette kan blive set på domænemodellen ved at de forskellige klasser ikke er så afhængige af hinanden.

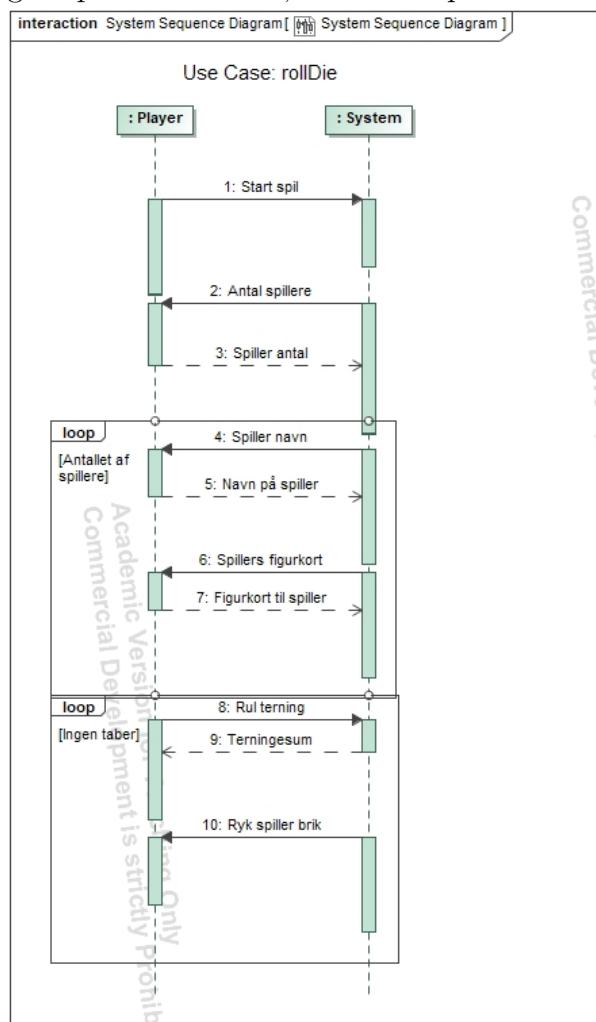
Dette gøres ved at objekter oprettes af andre objekter. Domænemodellen er skitseret nedenunder for at give et overblik over kodens sammenhæng til projektets samarbejdspartnere, som ikke nødvendigvis har noget forhold til kode.



Figur 1.2 DOMÆNEMODEL

1.10 System Sekvensdiagram - lav nyt diagram

Dette system-sekvensdiagram er på den fully dressed Use Case af rollDice. Diagrammet beskriver, hvordan brugerne og monopoly-systemet interagerer med hinanden. Når spillet bliver startet, skal spilleren vælge antallet af spillere og indsætte deres navn samt deres figurkort, som kører i et loop i forhold til valget af antallet af spillere. Herefter kaster den nuværende spiller på sin tur den virtuelle terning, hvor systemet returnerer terningesummen til spilleren, hvor systemet så rykker spillerens brik til det nye felt. Så længe ingen spillere har tabt, forsætter spillet.



FIGUR 1.3 SYSTEM SEKVENSDIAGRAM

2 Design

I designfasen gennemgås de vigtigste artifikater, som vil blive brugt i denne fase.

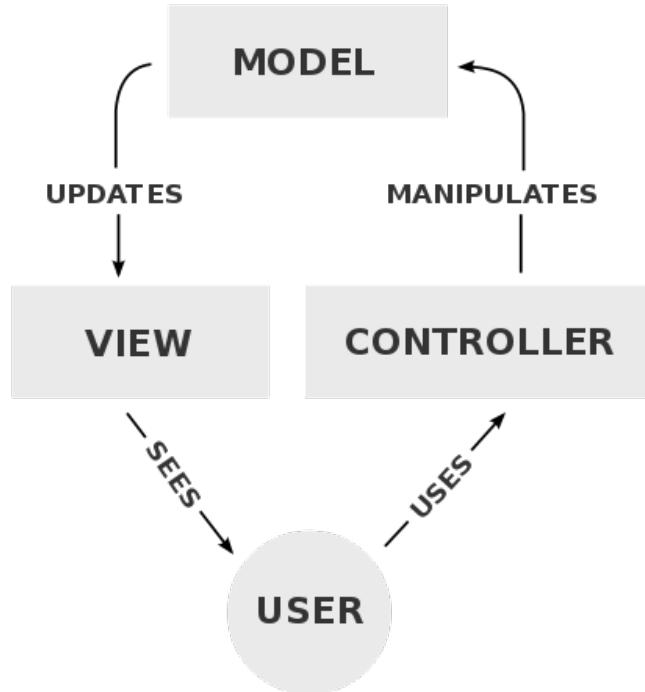
2.1 Arkitektur

Indeholder citater fra CDIO2.

Det er blevet besluttet at basere programmets arkitektur på konceptet ”Model View Controller”(MVC).

Fordelen med denne arkitektur er i dette tilfælde, at al input og output fra/til brugeren går gennem View-klassen, således er det ikke så omfattende at implementere forskellige former for userinterface (UI). Det kræver f.eks. en begrænset indsats at skifte konsol-interfacet ud med et grafisk interface, da der kun skal ændres i View-klassen.

Det er yderligere muligt at have forskellige View-klasser, dermed kan man have forskellige UI's. Man kan f.eks. have et UI til anvendelse på en smartphone og et andet til anvendelse på PC.



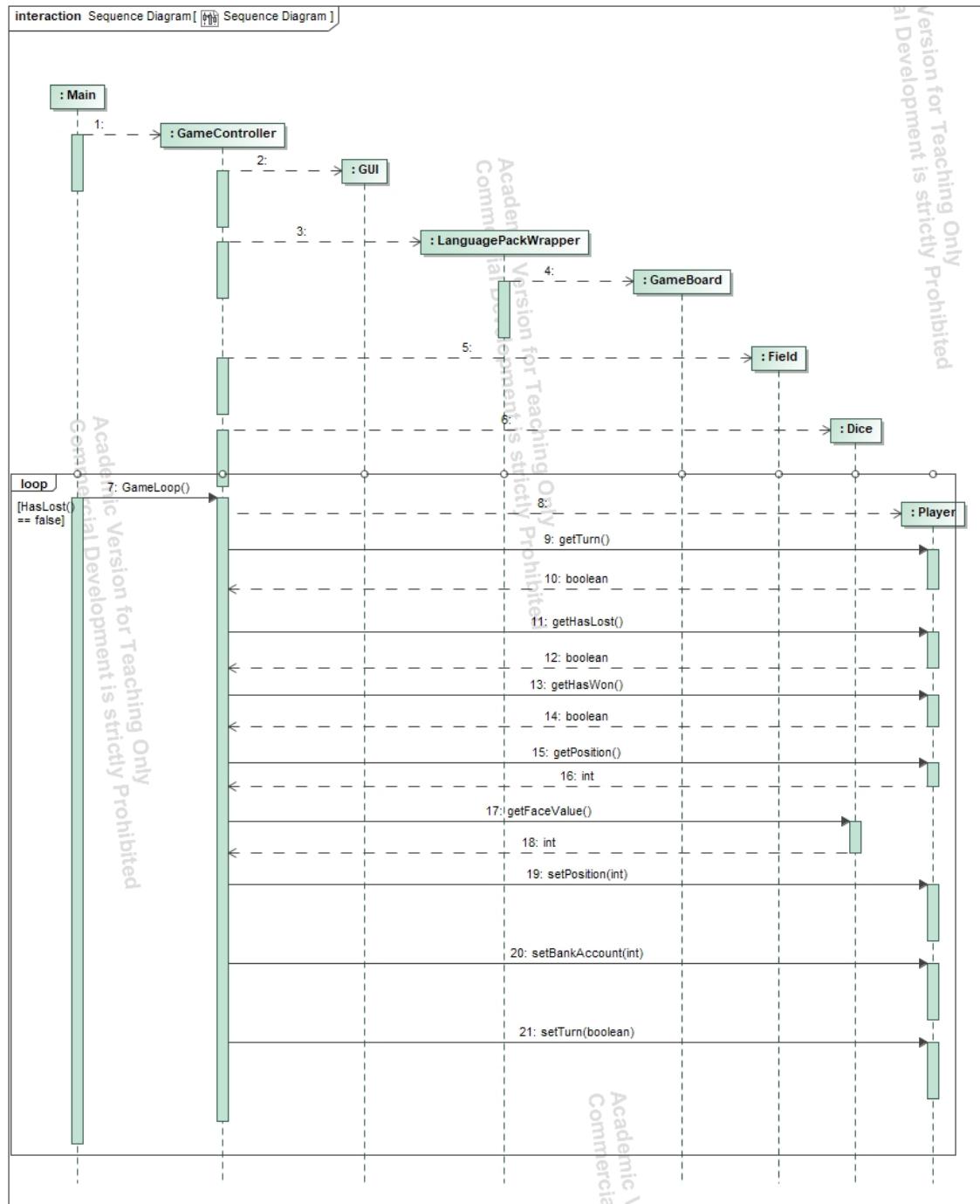
2.2 Designklassediagram

Designklassediagrammet kan ses i bilag2.

2.3 Sekvensdiagram

Herunder ses et sekvensdiagram over den del, som bliver tjekket igennem ved hver spillers tur. Diagrammet er lavet for at tydeliggøre, hvordan de forskellige klasser opretter hinanden. Som det ses på diagrammet, opretter GameController størstedelen af klasserne, mens nogle klasser opretter andre klasser. Det ses også, at det hele køre i et loop lige så længe at spilleren ikke har tabt.

Sekvensdiagrammet findes på næste side.



FIGUR 3.2 - SEKVENSDIAGRAM

2.4 GRASP-mønstre

Indeholder citater fra CDIO2.

Koden er udviklet med henblik på at følge GRASP-princippet ”High Cohesion - Low Coupling” (Dansk: Høj samhørighed - lav kobling). Dette gælder for klasserne imellem, som skal være så uafhængigt fungerende som muligt, men alligevel samarbejde på bedst mulig måde. Udførelsen af dette kan ses på designklassediagrammet (Bilag2), hvor ingen klasser er direkte afhængige af hinandens attributter.

Hvis man ændrer på en klasse, har det ikke effekt på de tilhørende samarbejdende klasser, mange af klasserne kan derfor ”tages ud” og genbruges i andre spil.

Det er forsøgt at få overholdt følgende andre GRASP-principper:

- Creator - Nye instanser af klasser bliver oprettet, hvor de bliver benyttet.
- Controller - Controlleren er i dette tilfælde GameController-klassen, som ”styrer” de andre klasser og instanser heraf.
- Information Expert - Hver klasse indeholder kun informationer tilhørende deres ansvarsområde.

3 Implementering

3.1 Overvejelser

Kravændringer

Under implementeringen blev det besluttet at ændre figurkortene fra:

- Hund, kat, båd og bil

Til:

- Car, racecar, tractor og ufo

Denne beslutning blev taget, da det ikke var muligt at ændre i GUI-klasserne, og det derfor ville blive omfattende at ændre figurkortene fra GUI-standarden.

Brug af tidligere CDIO-kode

Det er blevet valgt at genanvende kodedele fra CDIO1 og CDIO2.

Dice-klassen er blevet direkte genanvendt, og basis fra *Field* og *LanguagePack* klassen er yderligere blevet anvendt.

Et par klasser der umiddelbart spillede en stor rolle i tidligere projekter, som er blevet deklareret unødvendige i dette er *DiceCup* og *FieldFactory*.

DiceCup blev fjernet, da der kun er en enkelt terning.

FieldFactory blev fjernet da det ikke var særlig effektivt at skulle producere et nyt felt, hver gang det skulle anvendes. Felterne ligger nu i stedet for under *GameBoard* klassen.

3.2 Dokumentation

Indeholder citater fra CDIO2.

Programmet er løbende dokumenteret i Javadoc.

Den interaktive Javadoc-dokumentation er i HTML-format og fungerer som en webside. Den kan ses under mappen Javadoc i den afleverede zip-fil, hvor den åbnes ved at åbne index.html-filen i den foretrukne webbrowser.

I dokumentationen er alle klasser og metoder beskrevet på engelsk. Dermed kan der videreudvikles og vedligeholdes på koden af andre udviklingshold, danske såvel som internationale.

3.3 Nedarvning og abstrakt-klasser

I et objektorienteret programmeringssprog som Java kan en klasse nedarve fra en anden klasse. Har man f.eks. en klasse A, som nedarver fra en anden klasse B, så vil A have alle de samme synlige egenskaber og metoder som i klassen B. Eftersom A er en udvidelse af B, kan den godt have flere synlige metoder, og derfor kan man frit konvertere et objekt af typen A til et nyt objekt af typen B, men ikke omvendt.

Et interface kan betegnes som en klasse, hvor alle metoder og egenskaber er abstrakte, dvs. at et interface ikke implementerer nogle værdier eller standard funktioner. En abstrakt klasse kan derimod godt have ”standard”implementationer af dens metoder og/eller egenskaber, men kan stadig ikke direkte instansieres.

Et eksempel på et interface, der blev brugt i Matador-spillet ses forneden:

```
1 package main.Spil.Model;  
2  
3 public interface FieldActionListener {  
4     void onFieldLandedOn(FieldAction action);  
5 }
```

Dette interface beskriver at en klasse, som implementerer en ”FieldActionListener” skal have en metode ”onFieldLandedOn(FieldAction action)”. I Matador-spillet er der en liste over FieldActionListener. Eventet betragtes, når en spiller lander på et felt: Her kaldes alle FieldActionListener’s i listen.

I koden er der lavet forskellige implementationer af FieldActionListener, f.eks. PropertyFieldActionListener, som afgør om det felt, man lander på, er et ”ownable”felt. Herefter afgøres det om feltet er ejet/kan købes, eller om spilleren som landede på feltet skal betale en leje.

En anden implementation af FieldActionListener er klassen JailFieldActionListener, som afgør om feltet, man har landet på, er et ”go to prison”felt, og hvis det er så flytter den bilen til prison-feltet og trækker 2 penge fra spilleren.

Der er altså et ”event”system med en masse forskellige typer klasser med forskellige formål, som alle bliver kaldt, når spilleren lander på et felt.

3.4 Gennemgang af GameController'en

GameController

- *GameController()*
 - Metoden GameController() er konstruktøren i klassen, derfor deler den navn med klassen.
 - Det er her spillet bliver initialiseret, inden metoden startGame() bliver kørt i Main-klassen.
- *startGame()*
 - Dette er en forholdsvis stor metode ansvarlig for både at starte spillet og køre det.
Den starter med at lave en ny instans af terningeklassen med 6 sider.
Derefter går den i et while loop, som kører så længe, der ikke er fundet en taber.
I dette while loop er der et for-loop, som kører hver spillers tur igennem, ved at have en variable der går én værdi op hver gang loopet er kørt igennem, og så resetter når den når maks værdi.
I loopet bliver terningen rullet vha. die.roll(), bilen bliver rykket vha. moveCar() og alle events bliver kaldt.
- *clampPosition()*
 - Sørger for at spillernes position på brættet holder sig indenfor 0-24, altså antal felter på brættet.
- *moveCar()*
 - Rykker bilen frem et felt ad gangen for den nuværende spiller. Antal ryk er baseret på øjnene på terningeslaget.
- *updateCar()*
 - Sørger for at fjerne og tilføje bilens brik i samarbejde med moveCar() fra felt til felt, som skaber en illusion af animation.
- *sleep()*
 - Et genanvendt stykke kode fra TestRunExampleGame, som er ansvarlig for at lave pause mellem events i spillet.

3.5 Versionsstyring

GitHub er anvendt til Versionsstyring og parallel udvikling.

Link til git-repository: <https://github.com/Nicklas185105/CDIO3>

Projektet importeres i IntelliJ på følgende måde, hvis IntelliJ åbner et projekt:

1. Åben IntelliJ.
2. Tryk på ”File” i den øverste menubar til venstre.
3. Hold nu musen over ”New” så den udvider sig.
4. Herfra skal musen holdes over ”Project from Version Control”, så den også udvider sig.
5. Tryk her på ”Git” så der bliver åbnet et pop op vindue.
6. I boksen til ”URL” skrives linket ind.
 - Det kan være en god ide at trykke på knappen ”Test”, for at sikre sig at linket fungerer.
7. Tryk på ”Clone” og afvent IntelliJ får hentet koden ned fra GitHub.

Projektet importeres i IntelliJ på følgende måde, hvis IntelliJ åbner sit start vindue:

1. Åben IntelliJ.
2. Tryk på ”Check out from Version Control” nederst i midten.
3. Tryk her efter på ”Git” så der bliver åbnet et pop op vindue.
4. I boksen til ”URL” skrives linket ind.
 - Kan være en god ide at trykke på knappen ”Test”, for at sikre sig at linket fungere.
5. Tryk på ”Clone” og afvent IntelliJ får hentet koden ned fra GitHub.

3.6 Konfigurationsstyring

Udviklings- og produktionsplatform:

- Windows 10 version 1803 build 17134.345
- Java version 10.0.2
- IntelliJ version 2018.2.2
- matadorgui.jar version 3.1.4

Se anvendte udviklingsværktøjer i Bilag1.

4 Test

4.1 Test-cases

Der er tre test-cases til spillet, som er baseret på punkter, der har forsaget problemer i løbet af arbejdsprocessen.

1. Vælg Navn

- Main success
 - Spiller 1 vælger et navn og en figur.
 - Spiller 2 vælger et andet nav og en figur.
 - Begge spillere bliver registreret i siden af GUI'en og spillet går i gang.
- Fail case
 - Spiller 1 vælger et navn og en figur.
 - Spiller 2 vælger det samme navn og en figur.
 - Kun en af spillerne bliver registreret korrekt i GUI'en, men spillet fortsætter og virker som det skal.

2. Vælg Figur

- Main Succes
 - Spiller 1 vælger et navn og en figur.
 - Spiller 2 vælger et navn og en figur.
 - Random effekten gør at spillerne får forskellige farver.
- Fail case
 - Spiller 1 vælger et navn og en figur
 - Spiller 2 vælger en navn og en figur.
 - Random effekten gør at spillerne får samme farve. Dette gør det meget svært at se hvilken bil der er ejet af hvilken spiller.

3. Valg af vinder

- Main Success
 - En spiller løber tør for penge.
 - Spillet annoncere vinderen.
- Fail Case
 - Efter vinderen er annonceret, trykker en af spillerne for hurtigt på knappen/enter
 - Det er nu ikke længere muligt at se hvem vandt.

4.2 White box test

JUnit

Der er udarbejdet enhedstests på klasserne Dice, Field, Player og Gameboard. Der er anvendt testing-frameworket (JUnit 5).

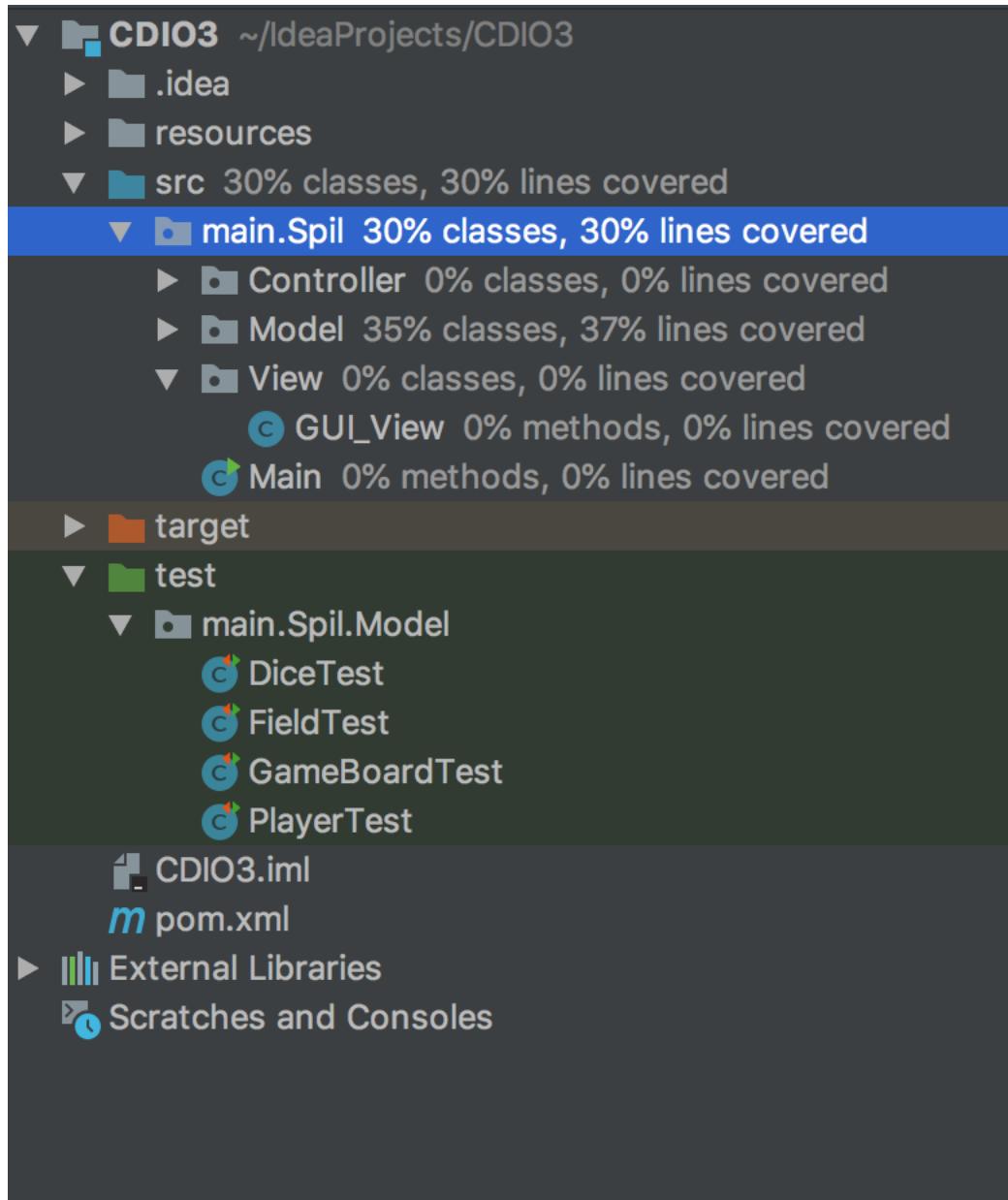
FieldTest: Metoden `toGUI` bliver testet, som er metoden, der opretter felterne på GUI'en. Der bliver oprettet en instans af et felt med tilhørende: navn, værdi, felttekst og felstype. Der bliver oprettet et `GUI_Field`, udfra metoden `field.toGUI`. Derefter benyttes der `Assert.assertEquals()`, til at tjekke efter, at det felt der blev oprettet i GUI'en har det samme navn, værdi, felttekst og felstype, som feltet blev sat til, til at starte med. Derved kan der afgøres om `toGUI` metoden virker korrekt.

PlayerTest: Der bliver testet på metoden `getName`, `setPosition` og `getPosition`. Der bliver oprettet en instans `player` af `player` og variablerne i `player` bliver tildelt udtryk. Derefter undersøges med `Assert.assertEquals()` at når metoden `getName` kaldes, om den så returnerer det korrekte navn.

GameBoardtest: Tester metoden `getFields` og `getGuiFields`, opsætter unit-tests som tjekker, at der bliver oprettet 24 felter og derefter tjekker, at den konverterer alle de 24 felter til en GUI felter og det præcise antal af 24 felter er korrekt.

Code coverage

30 procent af klasserne og metoderne er dækket med enhedstestene.



4.3 Black box test

Brugerinteraktionstest

Der vil blive kørt i alt fire brugerinteraktionstest mellem to forskellige anonyme spillere (fremover kaldet ”John” og ”Bob”). Disse personer har ikke nogen indsigt i koden.

| Spil | Spiller | Kommentar |
|------|---------|---|
| 1 | John | Det var vel meget sjovt, er godt nok et børnespil så ik så spændende. |
| 1 | Bob | Jeg tabte :(|
| 2 | John | Det var svært at se forskel på bilerne, da de var samme farve. |
| 2 | Bob | Farverne på spillet er ret ubehageligt for øjnene |
| 3 | John | Chancekortene er lidt kedelige |
| 3 | Bob | Ville være dejligt at kunne se hvem der ejer hvad |
| 4 | John | Ingen kommentar |
| 4 | Bob | Ingen kommentar |

5 Konklusion

Der er blevet udviklet et Monopoly-Junior-spil med tilhørende GUI. Spillet fungerer efter hensigten.

-

5.1 Evaluering

Gruppen har været under tidspres, hovedsageligt fordi et af gruppemedlemmerne går på ITØ-linjen fremfor Software-linjen. Konsekvensen har været at gruppemedlemmerne ikke har arbejdet på opgaven onsdage og torsdage.

Et forslag til fremtidig forbedring kunne være, at der først dannes grupper 1-2 uger efter studiestarten, samt at der fra undervisers side rådgives mod at danne gruppe på tværs af studieretningerne.

Gruppen har under arbejdet indset behovet for en gruppekontrakt, der sikrer alle gruppemedlemmers lige deltagelse, overholdelse af aftaler, stabil kommunikation m.v.

Gruppen har gennem hele projektarbejdet været meget utilfreds med den påkrævede GUI. Den er dårligt udført med kode, som modstrider undervisningen i god kodeskik. I øvrigt var 3.0.0-GUI-versionen, som der blev henvist til i opgavebeskrivelsen ikke kompatibel med Monopoly-spillet, da den var lavet kun til det store Matador-spil. Efter at have brugt meget tid på at forsøge at ændre i GUI'en, bl.a. ved at nedarve fra den, endte det med at den rigtige GUI-version (3.1.4) blev fundet, denne virkede bedre, men langt fra optimalt.

Følgende punkter blev nedprioriteret pga. manglende tid:

- Integrationstest
- Enkelte regler og spildetaljer som f.eks. salg af ejendomme og udtrækning af en vinder på baggrund af ejendomsporteføljen fremfor udelukkende saldobalanceens størrelse.
- Designklassediagrammet gav problemer. Til sidst blev det skrottet, og der blev til nøds implementeret et genereret diagram fra intelliJ.
- Test-cases blev mangelfulde grundet tidspres og lav kreativitetsniveau.

Bilag

Bilag 1 - Anvendte værktøjer

Følgende værktøjer er anvendt til projektet:

- Overleaf - parallel tekstbehandling I LaTeX
- IntelliJ - Javaudvikling
- GitHub - Versionsstyring og parallel udvikling
 - (Repo: <https://github.com/Nicklas185105/CDIO3>)
- MagicDraw - UML-diagrammer
- Javadoc - dokumentering af koden
- JUnit - enhedstest

