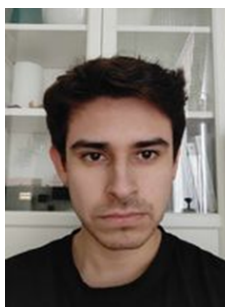

Projekt i softwareudvikling - 02362

Gruppe 7



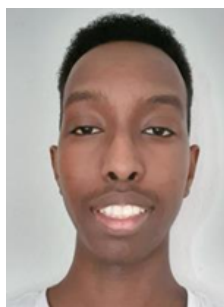
Jasmininder -

s235461



Emil -

s230995



Abaas -

s216173



Ali

s215716

Endelige Projektaflevering (rapport of software)

5 maj, 2024

Indholdsfortegnelse

Indholdsfortegnelse	1
Ansvarsområder ifm. rapporten	2
Introduktion	4
Conceive	5
Taksonomi	5
Krav og krav prioritering (MosCoW)	6
Databasekrav - Hvad skal gemmes?	8
Udgangspunktet - GUI og MVC	8
Use cases	9
Brief Use Cases:	9
Domænemodel	11
Aktivitetsdiagram	13
Design	15
Software Design	15
Klassediagram	15
MVC-arkitektur	15
Board og JSON	16
Sekvensdiagram over spillets start	17
Observer-designmønstret	18
Databasesdesign	19
Implementering	19
Model-klasser	19
Spillets flow	20
DrawBoardElements metoden	21
ConveyorBelt (doAction metoden)	22
JSON	23
Exceptions	26
Observer	28
Database Tilknytning	28
Generics	30
Rekursion	30
Javadocs	31
Operate (udviklingsproces og test)	31
Udviklingsproces og brugte værktøjer	31
Værktøjer:	31
Proces:	31
Test	32
J-unit test:	32
Formelle test	32
Håndbog	35
Installation af spillet:	35
Brug af spillet	35

Ordbog	37
Konklusion	38
Bibliografi	39
Bilag	39

Ansvarsområder ifm. rapporten

Diverse	Navn
Introduktion	Ali
Konklusion	Emil og Raed
Korrektur	Alle
Håndbog	Jasminder
Ordbog	Jasminder
Javadocs	Alle

Conceive (analyse)	Navn
Taksonomi	Alle
Krav og krav prioritering (MosCoW)	Emil og Raed
Database Krav	Emil og Raed
Udgangspunktet - GUI og MVC	Ali
Use cases	Ali
Domænemodel, aktivitetsdiagrammer og tilstandsdiagrammer	Alle

Design	Navn
Software Design	
- Klassediagram og MVC arkitektur	Abaas
- Observer - design mønstret	Ali
- Sekvensdiagram	Abaas

- Design af Board	Emil
- Board og JSON	Emil
Observer-designmønstret	Jasminder
Databasedesign	Abaas

Implementering	Navn
Model-klasser	Ali
Spillels flow	Ali, Abaas
DrawBoardElements metoden	Emil
ConveyorBelt (doAction metoden) (Emil)	Emil, Abaas
JSON	Emil, Jasminder
Exceptions	Ali
Database Tilknytning	Emil, Jasminder
Observer	Ali

Operate (udviklingsproces og test)	Navn
Udviklingsproces	Ali, Jasminder
Test	Ali
- Formelle test	Emil, Abaas

Introduktion

Denne rapport dokumenterer arbejdet udført i forbindelse med "Projekt i Softwareudvikling (02362)", der omfatter analyse, design og implementering af spillet RoboRally. RoboRally, et amerikansk brætspil, er blevet omformet til en digital udgave kodet i Java og understøttet af en MySQL-database til at bevare spildata og sikre kontinuitet mellem sessioner. Selvom grundlæggende GUI (Grafisk Brugergrænseflade) blev leveret af opgavestilleren, Ekkart Kindler, er den blevet modificeret og forbedret i denne implementering. I RoboRally starter spillerne med at planlægge fem træk ved hjælp af programmeringskort, som tilfældigt fordeles ved starten af hver runde. Dette skaber en differentieret strategisk udfordring, da spillerne skal forudse og reagere på banens hindringer, såsom vægge og transportbånd. Målet er at lande på alle banens checkpoints i den rigtige rækkefølge for at vinde spillet.

Conceive

Taksonomi

For at forstå RoboRally og dets regler har vi lavet en navneords analyse baseret ud fra RoboRally game guide. Se **bilag 1** og evt **bilag 2**.

RoboRally er et strategisk brætspil fra 2-6 spillere, hvor hver spiller bliver tildelt hver deres robot. Spillet gælder om at være den første spiller til at have været ved alle checkpoint felterne (i den rigtige rækkefølge).

Board og Boardelements

“Boardet”, udgør grundlaget for RoboRally spillet. Boardet består af “Spaces” der er positioner hvor robotterne kan befinde sig. Udover robotterne kan der også være “BoardElements” som f.eks. kan være “walls”, “Conveyorbelt” eller “pits”. Disse Board elements kan forhindre robotterne i at bevæge sig i visse retninger.

Robots og programcards

Spillerne kan flytte deres robotter igennem spaces ved at programmere dem. Dette sker ved brug af spillernes "Program Deck". Spillernes program deck består af 8 “ProgramCards”, og så snart et programcard er blevet brugt, skal der trækkes et nyt.

Spillet's Faser

Et RoboRally spil består af runder indtil spillet vindes, en runde består af 3 forskellige "Game phases". Jf. reglerne kan spillet også spilles uden upgrade fasen, dette medfører et væsentlig mere simpelt spil.

- **Upgrade phase**, Spillerne kan opgradere deres robotter.
- **Programming phase**, Spillerne programmer deres robotter, samtidigt.
- **Activation phase**, Spillerne aktiverer første programcard fra deres “programregister”, derefter, aktiveres alle boardelements. Dette varer ved indtil alle spillernes program cards er blevet aktiveret.

Krav og krav prioritering (MosCoW)

I udviklingen af RoboRally-spillet er det essentielt at identificere og prioritere funktionelle og ikke-funktionelle krav for at spillet opfylder spillernes forventninger og krav. Ved at opdele kravene efter MosCoW-metoden kan vi prioritere krav i henhold til deres vigtighed og indflydelse på funktionalitet og brugeroplevelse.

Projektets krav prioriteret efter MosCoW-metoden:

Funktionelle krav

Must have

1. Spillet skal kunne spilles af 2 til 6 spillere af gangen.
2. Spillet foregår på et spillebræt.
3. Spillerne skal kunne kende deres egen Robot (Der skal være forskel på robotternes udseende).
4. Spillerne skal have en startposition på brættet.
5. Spillet skal have hver deres stak af håndkort, som de skal kunne bruge til at programmere deres robot.
6. Spillet skal have basale programkort ("move forward", "turn right", "turn left")
7. Spillebrættet skal have checkpoints.
8. Spillebrættet skal have walls
9. Der skal være et vinderkriterie
10. Spillerne skal kunne se spillets tilstand på GUI
11. Spillet skal kunne gemmes og genoptages senere

Should have

1. Spillet burde have flere forskellige spilleplader
2. Spillet burde have eksplicitte startfelter til hver robot
3. Spillet burde have Conveyor Belt
4. Spillerne burde kunne skubbe hinanden
5. Spillet burde have interactive programkort f.eks (left or right)
6. Spillet burde afsluttes fornuftigt
7. En savegame og loadgame funktion

Could have

1. Spillet kunne have pits, spam card og reboot token.
2. Spillet kunne have en upgrade phase
3. Spillet kunne have energy cubes og energy spaces.
4. Spillet burde have flere boardelements (laser, pushpanels, gears)
5. Robotter kunne have lasere
6. Pænere grafik

Won't have

1. Spillet vil ikke have en online multiplayer-tilstand.
2. Spillet vil ikke have et scoringssystem for at bestemme vinderen.
3. Spillet vil ikke have Priority antenne

ikke-funktionelle krav

Must have

1. Spillet skal reagere hurtigt og køre uden forsinkelser.

Won't have

1. Spillet vil ikke have 3D GUI

2. Spillet vil ikke have flere sprog

Databasekrav - Hvad skal gemmes?

Formålet med at tilknytte spillet til en database er at kunne gemme og genlade et spil i og fra en database.

Vores software skal kunne gemme;

1. Antallet af spillere
2. Spillernes placering på spillepladen
3. Spillernes program kort
4. Spillernes programkort i registeret
5. Den aktuelle spiller
6. Spillets aktuelle fase
7. Hver spillers checkpoints

Udgangspunktet - GUI og MVC

Projektets fundament er skabt ud fra et leveret program-skelet og en funktionel GUI. Programstrukturen er baseret på Model-View-Controller (MVC)-mønstret, som adskiller ansvarsområderne i applikationen i tre hovedkomponenter: Model, View og Controller.

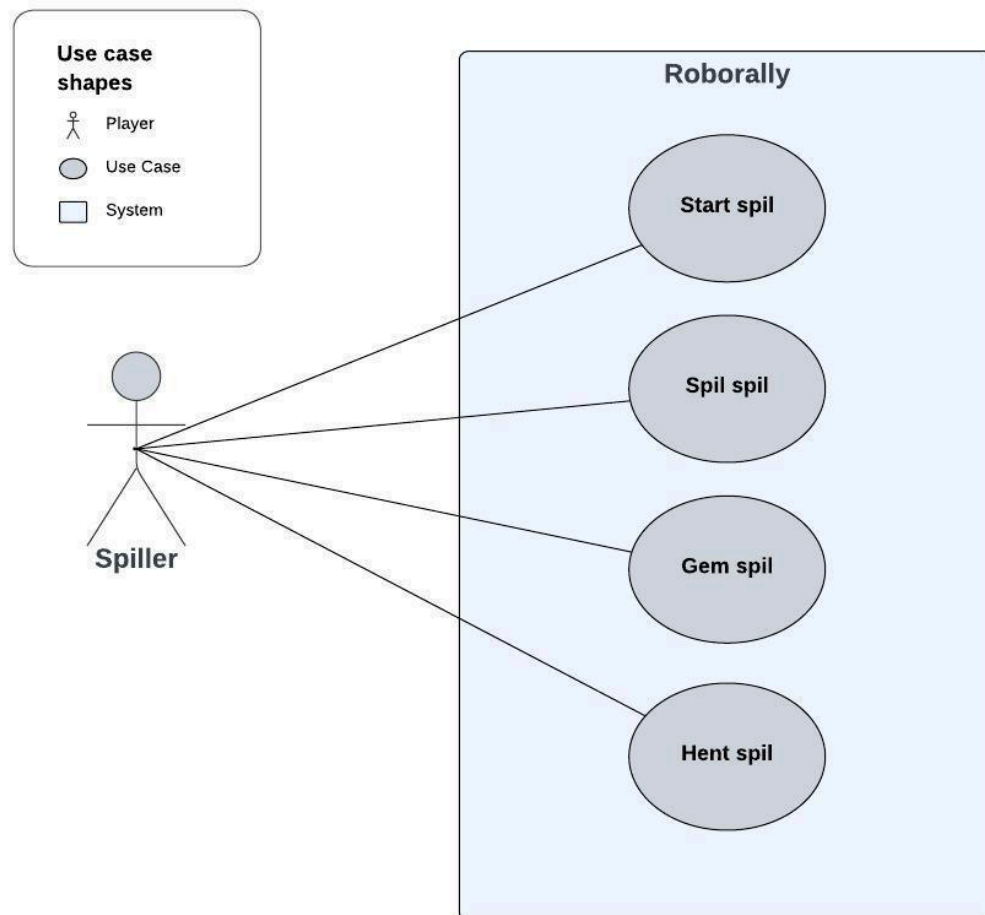
MVC-mønstret har en struktureret applikationsdesign ved at opdele datalogik, brugergrænseflade (UI) og input logik i separate dele. Modellen håndterer datahåndtering og accepterer brugerinput fra controlleren. View repræsenterer brugergrænsefladen, mens Controlleren styrer input logikken og koordinerer mellem Model og View.

Den givne GUI har dog nogle begrænsninger i forhold til interaktion med brugeren. GUI'en begrænser brugernes muligheder for uopfordret interaktion, da de primært skal reagere på programmets anmodninger om input. Dette skaber udfordringer i implementeringen af MVC-mønstret, da brugeren ikke kan interagere frit med programmet.

På trods af disse udfordringer tilbyder MVC en struktureret tilgang til at organisere ansvarsområderne og opdele kompleksiteten i applikationen. Denne opdeling muliggør fokus på specifikke områder af implementeringen ad gangen.

Use cases

Roborally har 1 aktør som er spilleren. Nedenfor i figur 1 ser vi nogle typiske eksempler som spilleren vil komme ud for ved at spille Roborally.



Figur 1 - Use case

Brief Use Cases:

Use case:	Start spil
ID:	UC_START
Aktør(er):	Spilleren
Kort beskrivelse:	Spilleren vælger i menuen, om de vil starte et nyt spil. I et nyt spil bestemmer man antal spillere, fra to op til seks, og begynder spillet.

Use case:	Spil spil
ID:	UC_SPIL
Aktør(er):	Spilleren
Kort beskrivelse:	Denne use case beskriver processen, hvor en spiller deltager i spillet ved at navigere fra startfeltet til forskellige checkpoints på spillebrættet. Spilleren bruger programmeringskort til at bestemme bevægelser og handlinger for sin robot på brættet. Målet er at nå alle forudbestemte checkpoints i en specificeret rækkefølge. Den første spiller, der når alle checkpoints, erklæres som vinderen af spillet.

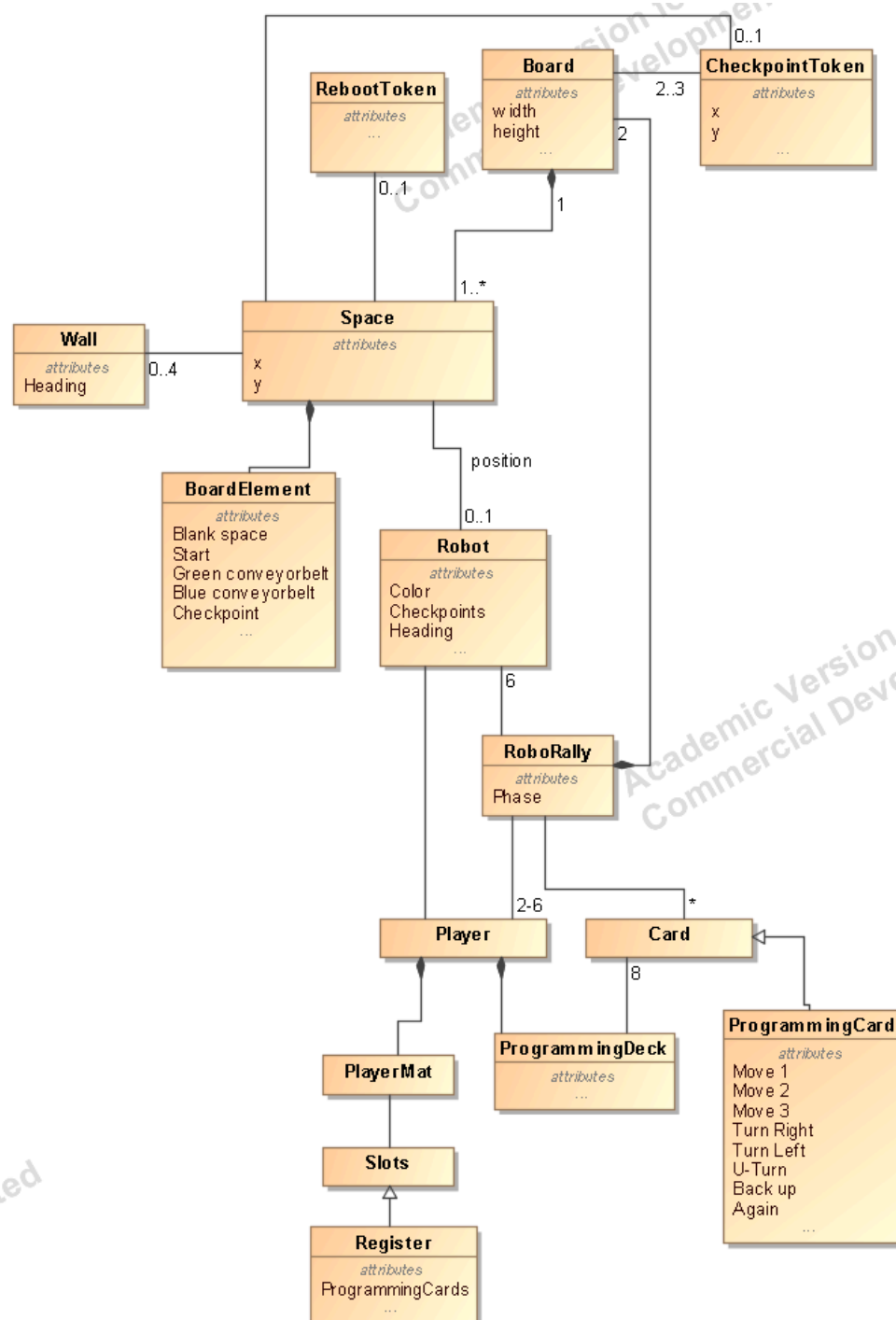
Use case:	Gem spil
ID:	UC_GEM
Aktør(er):	Spilleren
Kort beskrivelse:	En spiller kan til enhver tid gemme deres igangværende spil gennem menuen ved at vælge "gem spil". Dette gemmer spillet i databasen, hvor det kan hentes frem og fortsættes på et senere tidspunkt.

Use case:	Hent spil
ID:	UC_HENT
Aktør(er):	Spilleren
Kort beskrivelse:	Muligheden for at indlæse spil, som tidligere er gemt af spilleren, er lagret i databasen og kan frembringes nøjagtigt som spillet blev forladt.

Domænemodel

Nedenfor ses en domænemodel, udarbejdet ud fra taksonomien og RoboRally game guide.

Ved at udlede denne domænemodel ud fra taksonomien kan vi visualisere spillets konceptuelle struktur og forholdet mellem elementerne.



Figur 2 - tilpasset domænemodel

Vi laver 2 domænemodeller, den ovenfor er en tilpasset domænemodel, som kun indeholder relevante elementer for vores begrænsede version af roborally, og som ikke indeholder alle elementer fra spillets regler.

I bilag 3 ses en mere fuldstændig domænemodel, som indeholder en del flere elementer. Vi laver denne domænemodel for at få et overblik over det komplette roborally spil, uden at begrænse domænemodellen i forhold til vores krav.

Den tilpassede domænemodel

RoboRally er selvfølgelig det centrale for roborally domænet. Vores version af RoboRally består af

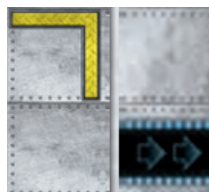
- 2 boards
- Cards
- 2-6 players
- 6 robotter.

Space

1 board består af flere spaces, de er relateret med en composition, da spaces ikke kan leve uden board. Spaces er associeret med Robot, da et space kan indeholde 0..1 Robots. Der kan altså være en robot på et felt, men det er også muligt at der ikke er nogle robotter på feltet.

BoardElement har en relation til space, da et space består af et boardelement. (Et boardelement kan også være et “Blankspace”)

Walls er ikke en attribut til boardelement, da walls har en anden cardinalitet i relationen til Space. Dette skyldes at walls har en heading og at der derfor kan være flere walls på et space, men der kan f.eks ikke være flere conveyerbelt på et felt. (se billede nedenfor)



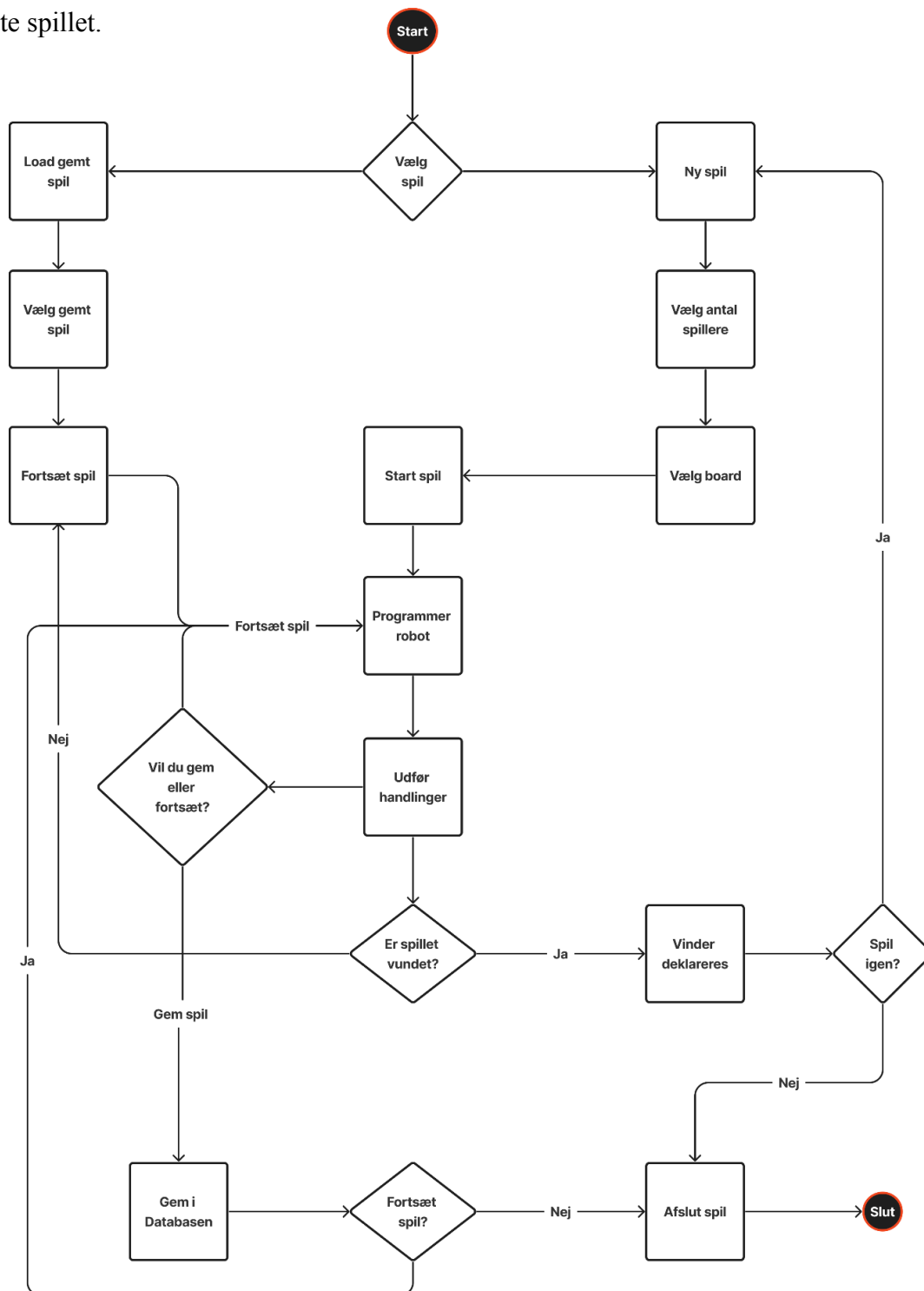
4 spaces, 2 walls, 1 conveyorbelt

Player

1 spiller er associeret med 1 robot, derudover har en spiller programcard i et register(spillerens program) og nogle i hånden (Programming Deck). Programming Card nedarver fra kort, dette er ikke særlig relevant for os da vi ikke implementere andre korttyper.

Aktivitetsdiagram

Dette aktivitetsdiagram afspejler den iterative natur af spiloplevelsen og giver en visuel repræsentation af de mulige stier en spiller kan tage gennem spillet, herunder at starte et nyt spil, indlæse et eksisterende spil, programmere robotter, håndtere spillets tilstand og til sidst afslutte spillet.



figur 3 - Aktivitetsdiagram

Aktivitetsdiagrammet for RoboRally spillet starter med en indledende handling, hvor spilleren står over for et valg om at starte et nyt spil eller indlæse et eksisterende spil fra databasen. Hvis spilleren vælger at starte et nyt spil, fortsætter flowet med valget af antal spillere og valget af spillebrættet. Dette leder derefter til handlingen "Start spil".

Når et spil er startet, bevæger flowet sig ind i spillets hovedloop, hvor spilleren programmerer deres robot og efterfølgende ser robotten udføre handlinger på brættet. Efter denne fase når spilleren et beslutningspunkt, hvor de kan vælge at gemme det nuværende spil i databasen eller fortsætte uden at gemme. Hvis spilleren vælger at gemme, fører flowet ned til en handling, der gemmer spillets tilstand i databasen, hvorefter er der et beslutningspunkt, hvor man kan vælge at fortsætte spillet eller afslutte spillet efter man har gemt.

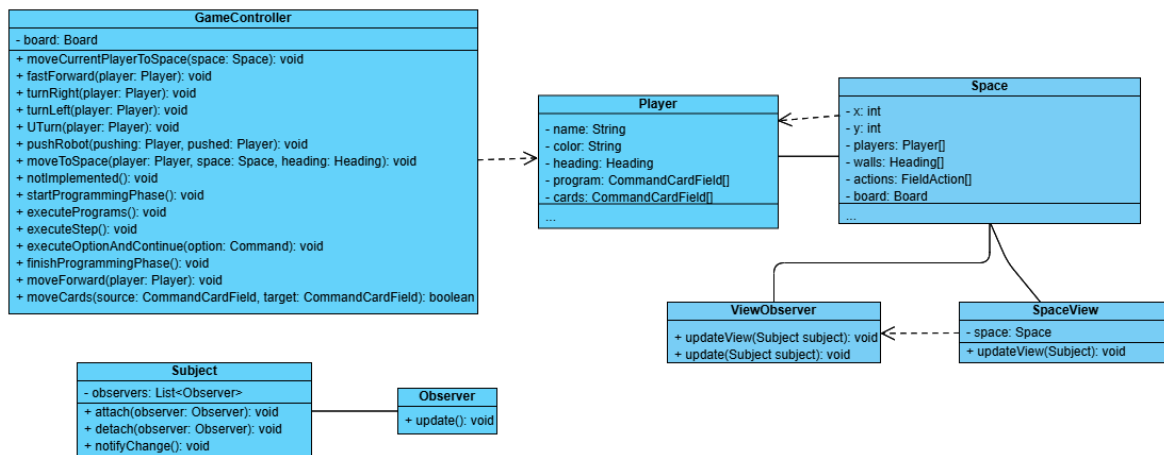
Et andet vigtigt beslutningspunkt i hovedloopen afgør, om spillet er vundet. Hvis spillet ikke er vundet, fortsætter spillet i loopet indtil der findes en vinder. Hvis spillet er vundet, bliver vinderen deklareret, og spilleren kan vælge imellem at spille igen eller afslutte spillet. Ved valget om at spille igen føres de tilbage til start ved valg af antal spillere.

Softwaredesign

Klassediagram

På vores klassediagram, har vi mulighed for at få et bedre overblik over hvordan det hele hænger sammen.

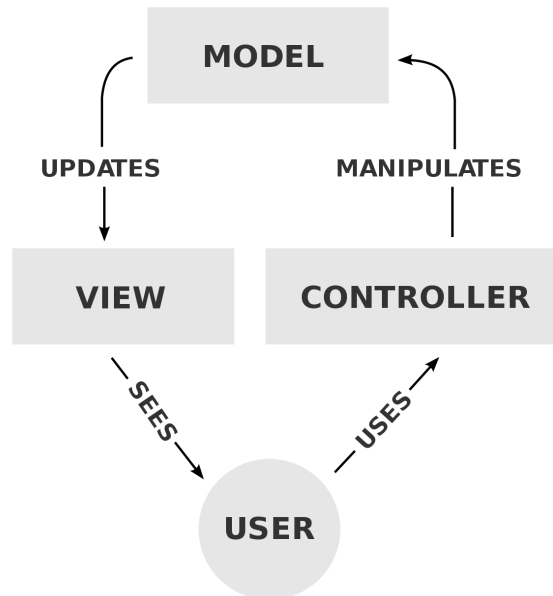
GameController-klassen, som er en del af vores Controller af MVC, manipulerer Player-klassen, som så er en del af Model-delen. GameController'en videregiver informationer som brugeren har tastet, som så sendes videre til Player-klassen, som så indeholder information om hvor spilleren er i spillet. Det bliver derfor sendt videre til Space-klassen, som så opdaterer de nye oplysninger den har fået til SpaceView, som er en del af View-delen, der så til sidst viser spilleren hvor spillerens nye placering er henne. Det er blevet via `updateView(Subject): void`.



figur 4 - Klassediagram

MVC-arkitektur

Model-View-Controller (MVC) er et designmønster, som har til formål at opdele ansvarsområder i form af klasser og deres funktionalitet. Det giver et bedre overblik over projektets programmeringsdel og gør det nemmere specielt ved større projekter.



figur 5 - Model-View-Controller (MVC)

Model:

Model-delen (M), styres data og spiller logikken, som så sendes videre til View-delen (V). I dette projekt ville det så være vores Board-klasse, Player-klasse og Command-klasse.

View:

View-delen, vises Model-delens data og spiller logik gennem GUI. Der vises f.eks. menuen, spillebrættet og spiller.

Controller:

Controller-delen, kan brugeren interagere med systemet, hvor den modtager input og sender det videre til Model og derefter tilbage til View. Det kan f.eks være når brugeren skal vælge antal spillere gennem ApplicationController-klassen i Controller og sende det videre til Player-klassen i Model.

Board og JSON

I dette projekt vil vi designe indlæsningen af board ved brug af JSON filer og Gson bibliotek.

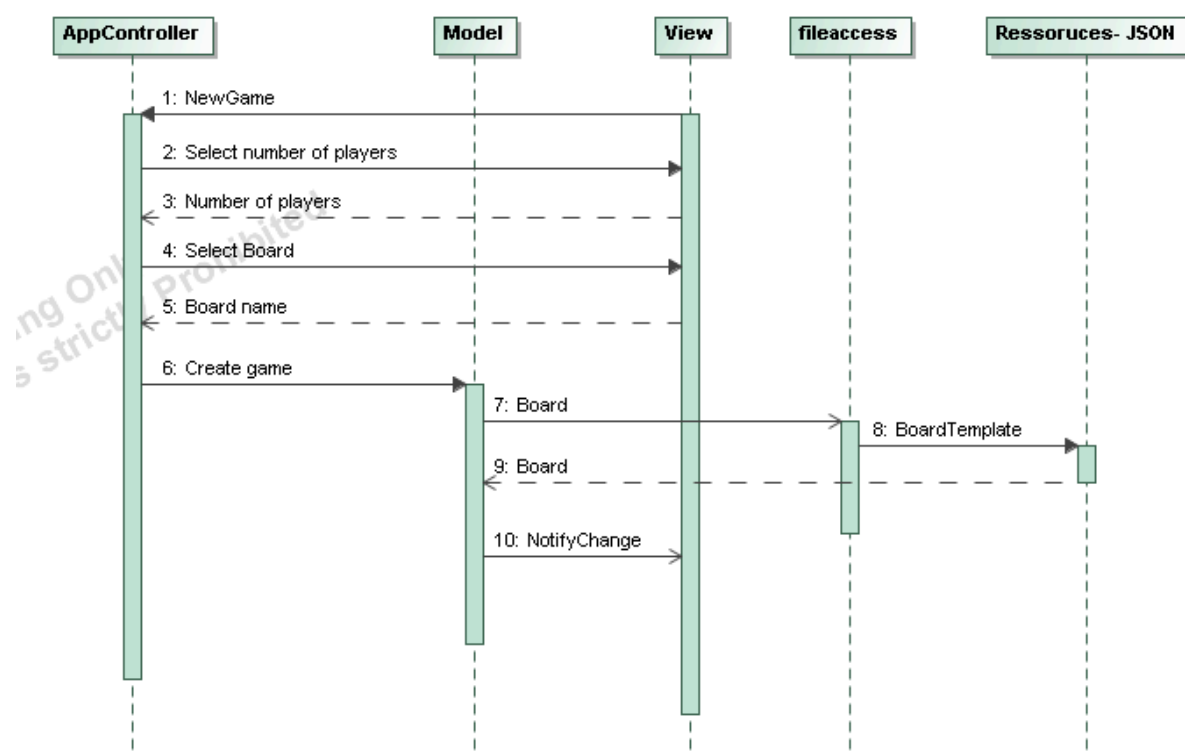
Dette vil gøre det nemmere at tilføje flere boards til spillet, da man bare kan oprette en ny JSON fil og dermed designe et nyt board uden at skulle ændre i selve java koden.

Ved hjælp af JSON og Gson kan vi gøre det meget nemmere at arbejde med data i vores program. I stedet for at skulle tænke på at konvertere data til og fra klasser, kan vi bare læse og skrive data direkte som objekter.

Det betyder, at vi ikke behøver at bekymre os om de tekniske detaljer ved at håndtere filer på et lavt niveau.

Sekvensdiagram over spillets start

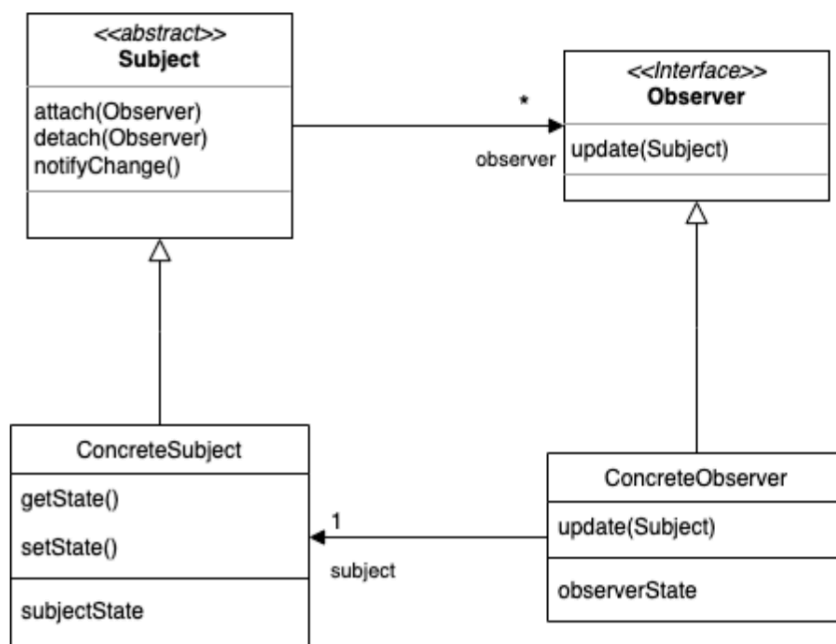
Denne model illustrerer sekvensen mellem pakkerne, når man starter et nyt spil..



figur 6 - Sekvensdiagram

Observer-designmønsteret

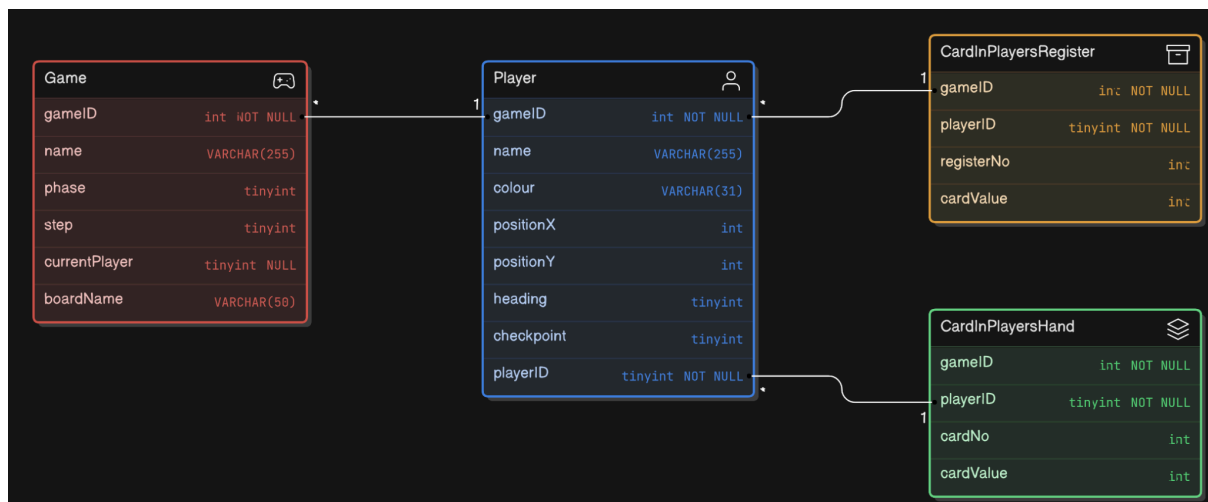
I forbindelse med anvendelsen af Observer-designmønsteret i vores projekt, har dette adfærdsmønster spillet en central rolle i styringen af opdateringer til view-laget. Observer-mønsteret etablerer et en-til-mange forhold mellem objekterne i spillet, hvilket sikrer, at ændringer i model-klassernes tilstand medfører automatiske opdateringer i view-klasserne. Dette sker gennem `notifyChange()`-metoden, som aktivt underretter alle registrerede observers om ændringerne, der herefter kan opdatere GUI'en via `update()`-metoden. Som illustreret i Figur 5, er Subject en abstrakt klasse, der faciliterer tilknytningen og fra koblingen af observers, mens ConcreteSubject vedligeholder objektets tilstand og underretter observers om eventuelle ændringer. Observer-klassen fungerer som en interface for disse opdateringer, og ConcreteObserver implementerer specifikke responsmekanismer, som reagerer på underretninger fra subjectet. I projektet bruges denne struktur til at sikre en effektiv kommunikation mellem model- og view-lagene, hvilket er afgørende for at opretholde GUI'ens aktualitet og reaktivitet.



Figur 7 - Observer-designmønsteret

I projektet anvendes dette mønster for at sikre, at view-klasserne automatisk og effektivt reagerer på ændringer i model-klasserne, hvilket resulterer i en synkroniseret opdatering af GUI'en. Med andre ord, ved hver ændring i model-klassernes tilstand, vil metoden `notifyChange()` sende besked til view-klasserne om at opdatere tilstanden ved at køre `update()` metoden.

Databasedesign



Figur 8 - Design over Database

For at kunne gemme spillet i en tilstand, som man kan vende til uden at det er slettet, har vi derfor oprettet en database med 4 tabeller, som ses ovenover.

Det vigtige er gameID, som giver det specifikke spil noget man kan genkende det på, og hvis man er flere spillere på det samme spil, kan man også genkendes på playerID.

Implementering

Model-klasser

Model-klasserne af de centrale komponenter i MVC (Model-View-Controller) arkitekturen, hvor de specifikt repræsenterer 'Model' delen. Disse klasser administrerer spillets datastruktur, logik og regler, og forbinder controller og view-klasserne. Domænemodellen viser alle disse implementeringer og de yderligere tilføjede implementeringer. Nedenstående ses nogle af de vigtigste klasser i model:

- **Board:** Denne klasse repræsenterer det fysiske spillebræt og styrer placeringen og logik af spillets elementer som felter og barrierer. Board-klassen er fundamentet for spillets regelsæt og interaktioner på brættet.
- **Player:** Klassen definerer spillerne i spillet, deres attributter og metoder til interaktion med spillet. Den omfatter funktioner til at styre spillernes positioner, sundhedstilstand og inventar, hvilket er afgørende for spillets dynamik.

- **Space:** Ansvarlig for at håndtere de enkelte spilfelter på brættet. Space-klassen inkorporerer logik til at bestemme felternes adfærd og deres interaktioner med spillere og spilobjekter, såsom vægge og skub-mekanismer.
- **Command:** Dette enum styrer de tilgængelige handlinger, som spillernes robotter kan udføre. Det er tæt integreret med gameplay-mekanismerne og definerer, hvordan kommandoer oversættes til handlinger på brættet.
- **CommandCardField:** Denne klasse administrerer anvendelsen af programmeringskortene, som er væsentlige for at styre robotternes bevægelser og handlinger. Det omhandler kortenes funktionalitet og deres indvirkning på spillet.

Spillets flow

Spillets flow styres gennem GameController-klassen og initieres i metoden `startProgrammingPhase`, hvor spillet forbereder og distribuerer kommandokort til spillerne. Når programmeringsfasen er afsluttet, aktiveres metoden `finishProgrammingPhase`, som overgår til aktiveringsfasen.

Under aktiveringsfasen, håndteres spillets logik primært af metoden `executePrograms`. Denne metode indleder en løkke, som fortsætter så længe spillet er i aktiveringsfasen. I denne fase kaldes `executeNextStep` metoden gentagne gange. Metoden starter med at bekræfte, hvilken spiller, der er `currentPlayer`, og verificerer at spillet stadig er i aktiveringsfasen og at `currentPlayer` ikke er NULL. Den tjekker også om den aktuelle rækkefølge af handlinger ikke overskrider det tilladte område.

Inde i `executeNextStep`, håndteres det aktuelle `CommandCard` fra `currentPlayer`. Hvis `CommandCard` ikke er NULL, undersøges det specifikke `Command` for kortet. Hvis kortet kræver interaktion (for eksempel et "venstre" eller "højre" drejekort), håndteres denne interaktion. Hvis ikke, eksekveres kommandoen direkte.

Herefter opdateres spillets tilstand baseret på feltaktioner. Det gøres ved at kalde `currentPlayer.getSpace().getActions()`. Hvis der findes handlinger tilknyttet det felt, hvor spilleren befinder sig, eksekveres disse handlinger en efter en.

Når alle spillernes kommandoer er blevet eksekveret for en runde, tjekker spillet, hvem der bliver den næste spiller ved at anvende `setNextPlayerToCurrent` metoden og fortsætter derefter runden. Når alle spillere har afsluttet deres handlinger, og runden er færdig, indledes en ny programmeringsfase, og cyklussen gentages.

DrawBoardElements metoden

Metoden `drawBoardElements()` er ansvarlig for visuelt at præsentere de forskellige board elementer som vi har implementeret. Metoden starter ud med at oprette en `ImageView` som er importeret med `javaFX`. `ImageView` kan bruges til at vise billeder, i vores tilfælde billeder af board elementer.

Herefter benytter vi en `switch` statement baseret på typen board element der skal tegnes/illustreres

```
192     public void drawBoardElements() { 1 usage
193         ImageView imageView = new ImageView();
194
195         switch (space.getActions().get(0).getClass().getName()) {
196             case "dk.dtu.compute.se.pisd.roboreally.controller.ConveyorBelt":
197                 ConveyorBelt conveyorBelt = (ConveyorBelt) space.getActions().get(0);
198                 drawConveyorBelt(conveyorBelt, imageView);
199                 break;
200             case "dk.dtu.compute.se.pisd.roboreally.controller.Checkpoint":
201                 Checkpoint checkpoint = (Checkpoint) space.getActions().get(0);
202                 drawCheckpoint(checkpoint, imageView);
203                 break;
204             case "dk.dtu.compute.se.pisd.roboreally.controller.StartSpace":
205                 drawStartSpace(imageView);
206                 break;
207             case "dk.dtu.compute.se.pisd.roboreally.controller.Pit":
208                 drawPit(imageView);
209                 break;
210             case "dk.dtu.compute.se.pisd.roboreally.controller.Reboot":
211                 Reboot reboot = (Reboot) space.getActions().get(0);
212                 drawReboot(reboot, imageView);
213                 break;
214             default:
215                 break;
216         }
217         if (imageView.getImage() != null) {
218             FitPictureToSpace(imageView);
219         }
220
221     }
```

Udklip af drawBoardElements metoden

Nedenfor i if statementet kaldes metoden FitPictureToSpace. Denne metode tilpasser jpg-billede filerne ved at bruge fitWidthProperty() og fitHeightProperty() til at tilpasse billederne til felt størrelserne. Ved at gøre det på denne måde undgår vi at billedet bliver udstrakt, i forhold til dens oprindelige bredde og højde.

ConveyorBelt (doAction metoden)

Denne metode tager den aktuelle spiller og bevæger det 1 felt i retningen af hvor conveyorbeltet peger hen.

Her er nogle situationer hvor vi har været udfordret med at implementere logikken.

(Se billederne fra venstre mod højre og se bilag 5)

Såfremt næste space er frit, flyttes den aktuelle spiller til den næste plads. Dette gøres ved at kalde pushRobot-metoden, som vi har implementeret i forbindelse med kollision af 2

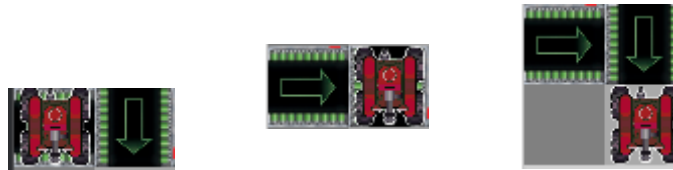


robotter.

Hvis pladsen er optaget, forsøger metoden at skubbe den anden spiller i samme retning (ved brug af pushRobot()).

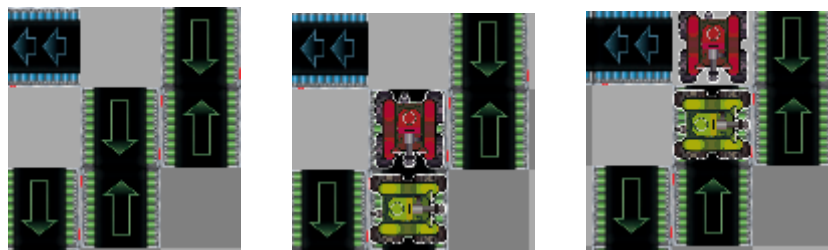


Hvis conveyorbeltet skubber en robot over til et andet conveyor belt, kaldes metoden rekursiv. Før doAction() bliver kaldt sættes nextPlayer.setActivated(true).



Dette sikrer at robotten ikke bliver flyttet 2 gange.

I en situation hvor 2 robotter står på hver sit conveyorbelt som peger ind mod hinanden, bør begge robotter blive stående. Dette sikrer vi med nextPlayer.setActivated(true). Nedenfor er et eksempel hvor next Player.set Activated(false), hvilket resultere i at



metoden håndterer situationen forkert.

JSON

Templates

Denne klassediagram til hvordan en spilleplade og dens element skal bygges op.

width er udtryk for hvor mange felter spillepladen har vertikalt, og heigth udtrykker antallet af felter horisontalt. På samme måde har vi implementeret checkpoints.

Ud over en højde og en bredde består en spilleplade også af spaces. For at holde styr på spaces, er der implementeret en ArrayList som består af SpaceTemplate.


```

34 public class BoardTemplate { 5 usages
35     public int width; 2 usages
36     public int height; 2 usages
37     public int checkpoints; 1 usage
38     public List<SpaceTemplate> spaces = new ArrayList<>(); 2 usages
39 }

```

Spacetemplate har også en position x og y

```

35 public class SpaceTemplate { new *
36
37     public int x; 2 usages
38     public int y; 2 usages
39
40     public List<Heading> walls = new ArrayList<>(); 2 usages
41     public List<ActionTemplate> actions = new ArrayList<>(); 2 usages
42
43 }

```

Indlæsning af JSON- filen

i LoadBoard-klassen håndteres indlæsningen af spillebræt fra en JSON-fil med hjælp fra Gson biblioteket, som gør det muligt at serialisere Java-objekter til JSON. Denne del af implementeringen tillader os at opsætte spilobjekterne ud fra dataene i en JSON-fil.

Stien til mappen hvor JSON-filerne er placeret. LoadBoard tager navnet på spillebrættet som parametre, såfremt navnet er “null” returneres et Defaultboard.

Herefter anvendes classloader til at læse den pågældende JSON-fil som et inputStream. Når inputStream er oprette, bruges GsonBuilder til at oprette et Gson-objekt, i forbindelse med dette anvendes en adapter.

Derefter oprettes JsonReader, der indlæser JSON-filen fra inputStreamen. Derefter kan dataene deserialiseres til BoardTemplate objekterne. Board-objektet bliver oprettet baseret ud fra BoardTemplate-objektet. JsonReader lukkes for ikke at bruge unødige ressourcer.

```

41 public class LoadBoard { 6 usages
42
43     private static final String BOARDSFOLDER = "boards"; 2 usages
44     private static final String DEFAULTBOARD = "defaultboard"; 1 usage
45     private static final String JSON_EXT = ".json"; 2 usages
46
47     @ public static Board loadBoard(String boardname) { 2 usages
48         if (boardname == null) {
49             boardname = DEFAULTBOARD;
50         }
51
52         ClassLoader classLoader = LoadBoard.class.getClassLoader();
53         InputStream inputStream = classLoader.getResourceAsStream( name: BOARDSFOLDER + "/" + boardname + "." + JSON_EXT);
54         if (inputStream == null) {
55             return new Board( width: 8, height: 8, boardname);
56         }
57
58         // In simple cases, we can create a Gson object with new Gson():
59         GsonBuilder simpleBuilder = new GsonBuilder().
60             registerTypeAdapter(FieldAction.class, new Adapter<FieldAction>());
61         Gson gson = simpleBuilder.create();
62
63         Board result;
64         JsonReader reader = null;
65         try {
66             reader = gson.newJsonReader(new InputStreamReader(inputStream));
67             BoardTemplate template = gson.fromJson(reader, BoardTemplate.class);
68             result = new Board(template.width, template.height, boardname);
69             for (SpaceTemplate spaceTemplate: template.spaces) {
70                 Space space = result.getSpace(spaceTemplate.x, spaceTemplate.y);
71                 if (space != null) {
72                     space.getActions().addAll(spaceTemplate.actions);
73                     space.getWalls().addAll(spaceTemplate.walls);
74                 }
75             }
76             reader.close();
77             return result;
78         } catch (IOException e1) {
79             if (reader != null) {
80                 try {
81                     reader.close();
82                     inputStream = null;
83                 } catch (IOException e2) {}
84             }
85             if (inputStream != null) {
86                 try {
87                     inputStream.close();
88                 } catch (IOException e2) {}
89             }
90         }
91         return null;
92     }

```

udklip fra LoadBoard klassen,

JSON-filer

Nedenfor er vist et udklip fra en JSON fil hvor man kan se hvordan koden ser ud til at lave feltet

“x”: 2 og “y”: 0.

“CLASSNAME” indikere hvilken boardelement (Conveyorbelt)

“heading” Hvilken retning (EAST)

“doublecb” false hvis det er grøn conveyor belt og true hvis det er blå conveyor belt.

Se evt. bilag 2 på feltet: “x”: 2 og “y”: 0.

```
1  {
2    "width": 13,
3    "height": 10,
4    "checkpoints": 2,
5    "spaces": [
6      {
7        "walls": [],
8        "actions": [
9          {
10             "CLASSNAME": "dk.dtu.compute.se.pisd.roborally.controller.ConveyorBelt",
11             "INSTANCE": {
12               "heading": "EAST",
13               "doublecb": false
14             }
15           }
16         ],
17         "x": 2,
18         "y": 0
19       },
```

udklip af JSON filen “Risky Crossing.JSON”

Exceptions

Vi har implementeret exception-håndtering for at styre særlige tilfælde, hvor spillets normale flow afbrydes på grund af ulovlige eller umulige træk. Et primært eksempel på dette er ImpossibleMoveException, en exception der kastes, når et træk ikke kan gennemføres som planlagt på grund af spillets regler eller fysiske barrierer på spillebrættet.

Derudover implementerer vi try-catch blokke for at fange og håndtere SQLExceptions, som kan opstå under database operationer såsom tilslutning, data indsættelse , opdatering og læsning. Dette sikrer, at vores applikation kan reagere passende på fejl uden at afbryde brugerens oplevelse. For eksempel:

I Connector.java bruges catch til at fange SQLException når der forsøges at oprette forbindelse til databasen. Hvis der opstår en fejl under forbindelsen, udskrives fejlen ved hjælp af e.printStackTrace(), som giver detaljer om fejlen:

```
Connector() {
    try {
        // String url = "jdbc:mysql://" + HOST + ":" + PORT + "/" + DATABASE
        String url = "jdbc:mysql://" + HOST + ":" + PORT + "/" + DATABASE +
        connection = DriverManager.getConnection(url, USERNAME, PASSWORD);

        createDatabaseSchema();
    } catch (SQLException e) {
        e.printStackTrace();
        // Platform.exit();
    }
}
```

udklip af exception i vores DB - Connector.java

Eksempel 2:

I Repository.java, under createGameInDB-metoden, bruges catch også til at fange SQLException når der forsøges at indsætte et nyt spil i databasen. Denne metode er kritisk, da den involverer flere trin, hvor der kan opstå fejl, såsom at oprette spilposter, spillere og kort:

```
} catch (SQLException e) {
    e.printStackTrace();
    System.err.println("DB error");

    try {
        connection.rollback();
        connection.setAutoCommit(true);
    } catch (SQLException e1) {
        e1.printStackTrace();
    }
}
```

udklip af exception i vores DB - Repository.java

Under indsættelse af et nyt spil, hvis der opstår en SQLException, aktiverer dette den primære catch blok. Her logges fejlen først detaljeret med e.printStackTrace(), der giver indsigt i problemets natur, efterfulgt af en simpel fejlmeddelelse System.err.println("DB error") til systemets fejllog.

Ved fejl forsøges en transaktions-rollback med `connection.rollback()`, hvilket er afgørende for at forhindre halvfærdige data i at forblive i databasen og dermed bevare dataintegriteten. Efter rollback-processen genetableres automatisk commit tilstanden ved `connection.setAutoCommit(true)`, hvilket normaliserer forbindelsen for fremtidige operationer.

Hvis rollback fejler, en sjælden men kritisk hændelse, håndteres dette i en sekundær catch blok, hvor fejlen logges med `e1.printStackTrace()`. Dette sikrer dokumentation og muliggør senere fejlsøgning og korrektion.

Observer

Observer-mønstret anvendes til at sikre, at ændringer i model-klassernes tilstand bliver korrekt reflekteret i spillets brugergrænseflade (GUI). I vores projekt bruges mønstret sådan, at når en tilstandsændring opstår i en af model-klasserne, fanges denne ændring via en `notifyChange()` metode, som er implementeret i disse klasser gennem nedarvning fra en abstrakt Subject klasse.

Når `notifyChange()` metoden kaldes, trigger den en `update()` metode i View-klassen. Denne `update()` metode er ansvarlig for at opdatere GUI'en for at afspejle de ændringer, der er foretaget i modellen. Dette sikrer en synkronisering mellem spillets logik (model) og det, brugerne ser og interagerer med på skærmen (view).

Eksempelvis, i Player-klassen, som er en af model-klasserne, vil enhver ændring i spillerens tilstand (som position) udløse et kald til `notifyChange()`. Dette fører til, at den tilknyttede View-klasse opdateres gennem `update()` funktionen, hvilket sikrer, at alle relevante visuelle repræsentationer af spillerens tilstand er opdateret i realtid.

Database Tilknytning

Database tilknytningen håndteres gennem DAL-pakken (Data Access Layer). Denne pakke fungerer som en bro mellem systemet og databasen.

Databaseforbindelse

Forbindelsen til vores MySQL-database sker gennem Connector-klassen. Denne klasse er ansvarlig for at etablere forbindelse til databasen ved hjælp af forbindelses parametrene, vært, port, database, brugernavn og adgangskode. Hertil gør vi brug af en JDBC driver for at gøre det muligt at få Java og database til at kommunikere. Denne driver er tilføjet som en dependency til pom.xml filen.

```
38 class Connector { 3 usages
39
40     private static final String HOST = "localhost"; //<-- update this to your own host 1usage
41     private static final int PORT = 3306; //<-- update this to your own port 1usage
42     private static final String DATABASE = "pisd"; //<-- update this to your own database 1usage
43     private static final String USERNAME = "root"; //<-- update this to your own username 1usage
44     private static final String PASSWORD = "*****"; //<-- update this to your own password 1usage
45
46     private static final String DELIMITER = ";"; 1usage
```

Udklip af connector klassen

I connector klassen kaldes metoden createDatabaseSchema(). denne metode læser vores SQL-script fra ressource pakken, hvorefter den kan designe databasen.

Prepare statements

I repository-klassen bruges prepared statements. Det gør vi da man er mindre tilbøjelig til at lave fejl når man opdaterer databasen. Nedenfor er et prepared statement som har til formål at hente data fra tabellen “player”, baseret på et givet gameId.

```
594 /**
595  * The prepared statement to send to the database in SQL that selects everything in the Player table that matches the gameId
596  * @Author Ekkart Kindler
597  */
598 private static final String SQL_SELECT_PLAYERS = 1 usage
599     "SELECT * FROM Player WHERE gameId = ?";
600
601 private PreparedStatement select_players_stmt = null; 3 usages
602
603 private PreparedStatement getSelectPlayersStatementU() { 2 usages
604     if (select_players_stmt == null) {
605         Connection connection = connector.getConnection();
606         try {
607             select_players_stmt = connection.prepareStatement(
608                 SQL_SELECT_PLAYERS,
609                 ResultSet.TYPE_FORWARD_ONLY,
610                 ResultSet.CONCUR_UPDATABLE);
611         } catch (SQLException e) {
612             // TODO error handling
613             e.printStackTrace();
614         }
615     }
616     return select_players_stmt;
617 }
```

prepared statement

Generics

Hvad ved vi generelt om generics?:

Generics i programmering kommer i tale til en funktion eller konstruktion, der giver mulighed for at skrive kode, som kan håndtere forskellige datatyper uden at vide hvad de præcis er, eller skal udfører på forhånd. Ved hjælp af generics kan man udvikle genanvendelig og fleksibel kode, der kan tilpasses til forskellige typer data, samtidig med at typen sikres.

Hvor har vi brugt generics? Herunder forklares nogle eksempler:

I vores kode kan man fange et eksempel på generics i (APPcontroler). Her programmerer vi således at, en liste oprettes og derefter lader den holde på information omkring typen af strings og interger's, herunder tager vi brug af Json.

Derudover bruges der også genericsm når et nyt bord bliver oprettet.

Rekursion

Hvad er rekursion:

I programmeringsfaget kender vi rekursion som en funktion, der kalder tilbage på sig selv ved evt. problemløsning eller lignende. Dette er i sært brugbart, når problemet ikke er lokaliseret et sted, med derimod har et større omfang der skal gennemtjekkes, men stadig løses af samme funktion.

Generelt består denne metode af to punkter. En base, også kendt som et stoppunkt der afslutter processen, og et rekursivt hvor funktionen kalder tilbage på sig selv.

Rekursion kan bruges til at implementere algoritmer som fx faktorielberegning, Fibonacci-sekvensgenerering og gennemgang af træstrukturer.

Hvor har vi brugt det? Herunder forklares dette med et par eksempler fra vore software:

Her har vi taget brug i vores Movetospace, hvor metoden som ses kalder på sig selv, for at spillet skal kunne fortsætte. Dette sker ved, vi rykker vores brik til en position hvor der

allerede står en brik. Her bliver den før stående brik altså skubbet, og dermed bruger metoden rekursion.

Dette kunne evt. også skabe en kædereaktion, hvis andre brikker også står i vejen.

Javadocs

I vores Javadocs-dokumentation har vi dækket vigtige klasser og metoder, der ikke er omtalt i rapporten. Dokumentationen fremgår i begyndelsen af hver klasse eller metode og består typisk af få, koncise linjer. Formålet med Javadocs har primært været at dokumentere indgangsdata, funktionalitet og udgangsdata for hver klasse og metode, således at formålet og brugen af disse er klart og tydeligt defineret.

Operate (udviklingsproces og test)

Udviklingsproces og brugte værktøjer

Værktøjer:

I projektet har vi anvendt MySQL Workbench til databasestyring og IntelliJ IDEA til softwareudvikling, skrevet i programmeringssproget Java. Maven er benyttet til at håndtere afhængigheder, især for GUI-komponenter. Til versionskontrol og samarbejde har vi benyttet GitHub, hvilket har muliggjort effektiv kodehåndtering, herunder pull requests og code reviews.

Proces:

Vores udviklingsproces har været baseret på agile metoder med ugentlige møder, hvor vi har revideret og planlagt projektets fremgang. Dokumentationen er udarbejdet sideløbende med udviklingen.

Test

J-unit test:

Vi har anvendt JUnit-tests for at validere funktionaliteten af vores spillogik, specifikt gennem tests af GameController-klassen. Disse tests er essentielle for at sikre, at alle spilmekanikker fungerer korrekt og som forventet under forskellige spilscenarier.

Eksempler på JUnit Tests:

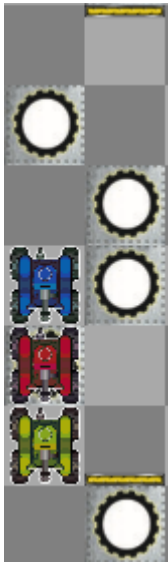
1. Test af Spillerbevægelser:
 - `movePlayerTurnAndMove()`: Tester korrekt bevægelse og drejning af spilleren på brættet. Denne test sikrer også, at spillerens oprindelige position bliver frigjort korrekt.
2. Test af Bagudgående Bevægelse:
 - `movePlayerBackward()`: Bekræfter at spilleren kan bevæge sig bagud og lander på den forventede position.
3. Test af Skubning af Robotter:
 - `pushPlayer()`: Verificerer, at en robot korrekt kan skubbes til en ny position på brættet.
4. Test af Drejninger:
 - `turnRight()` og `turnLeft()`: Disse tests tjekker, at spillerens orientering opdateres præcist ved drejninger.
 - `makeUTurn()`: Sikrer, at en spiller korrekt kan udføre en 180-graders vending.

Ved at implementere disse JUnit-tests garanterer vi, at spillets kernefunktionaliteter er pålidelige og fejlfri

Formelle test

Test case ID	TC_01 <code>movePlayerTurnAndMove</code>
Summary	Test om spilleren rykker til de rigtige felter når <code>movePlayer(Player player, int SpacesToMove,</code>

	boolean MoveBackwards) metoden bruges.
Requirements	Spilleren skal bevæge sig til det rigtige felt ved brug af gameController.movePlayer()-metoden.
Preconditions	width: 10 height:10 checkpoints: 2 boardname: "board"
Test procedure	<pre> 17 @Test 18 void movePlayerTurnAndMove() { 19 Player player1 = new Player(board, color: null, name: "Player1"); 20 player1.setSpace(gamecontroller.board.getSpace(x: 0, y: 0)); 21 assertEquals(player1.getSpace(), gamecontroller.board.getSpace(x: 0, y: 0)); 22 gamecontroller.movePlayer(player1, SpacesToMove: 3, MoveBackwards: false); 23 assertEquals(player1.getSpace(), gamecontroller.board.getSpace(x: 0, y: 3)); 24 player1.setHeading(Heading.EAST); 25 gamecontroller.movePlayer(player1, SpacesToMove: 2, MoveBackwards: false); 26 assertEquals(player1.getSpace(), gamecontroller.board.getSpace(x: 2, y: 3)); 27 } 28 Assertions.assertNull(board.getSpace(x: 0, y: 0).getPlayer()); 29 Assertions.assertNull(board.getSpace(x: 0, y: 3).getPlayer()); 30 31 }</pre>
Expected result	<ol style="list-style-type: none"> 1. Spiller 1 burde været på felt (0,0) 2. Spiller 1 burde være på felt (0,3) 3. Spiller 1 burde være på felt (2,3) 4. felt (0,0) og (0,3) burde være tomme
Actual result	<ol style="list-style-type: none"> 1. Spiller 1 er på felt (0,0) 2. Spiller 1 er på felt (0,3) 3. Spiller 1 er på felt (2,3) 4. felt (0,0) og (0,3) er tomme
Status	Passed
Tested by	Emil Hansen, s230995
Test environment	IntelliJ IDEA Ultimate 2024.1

Test case ID	TC_02 Manuel blackbox test
Summary	Test om de spiller1 (Rød), spiller2 (Grøn) og spiller3 (Blå) rykker til de rigtige felter når Spiller 3 aktivere Programmeringskortet "Forward 3"
Requirements	Spilleren skal bevæge sig til det rigtige felt ved brug af gameController.movePlayer()-metoden.
Preconditions	<p>New game boardname: "Risky crossing" Spiller 3 inden denne situation aktiveret: "Turn right" "Move 1" "Turn left" "Move 1"</p> 
Test procedure	Jeg flytter commando kortet "Move 3" op i det første felt i registeret, trykker "Execute Program", og ser hvor spillernes fysiske position er.
Expected result	- (x,y)

	<ul style="list-style-type: none"> - Spiller 1 burde været på felt (0,6) - Spiller 2 burde være på felt (0,7) - Spiller 3 burde være på felt (0,5)
Actual result	<ul style="list-style-type: none"> - Spiller 1 er på felt (0,6) - Spiller 2 er på felt (0,7) - Spiller 3 er på felt (0,5)
Status	Passed
Tested by	Emil Hansen, s230995
Test environment	IntelliJ IDEA Ultimate 2024.1

Håndbog

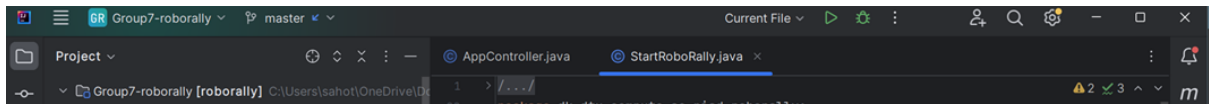
Installation af spillet:

1. For at kunne køre vores Roborally skal man have installeret IntelliJ.
2. Efter at have installeret dette, skal man nu åbne dette Github Link

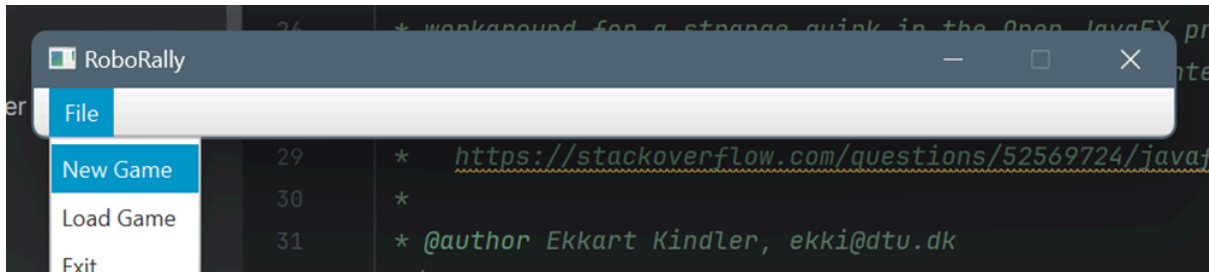
https://github.com/Alihoussein2200/roborally_gruppe7
3. Herefter skal man åbne IntelliJ og herefter vælge File > New > Project, nu skal man her i URL indsætte vore Github link som er tildelt herover.
4. Nu er man klar til at køre spillet

Brug af spillet

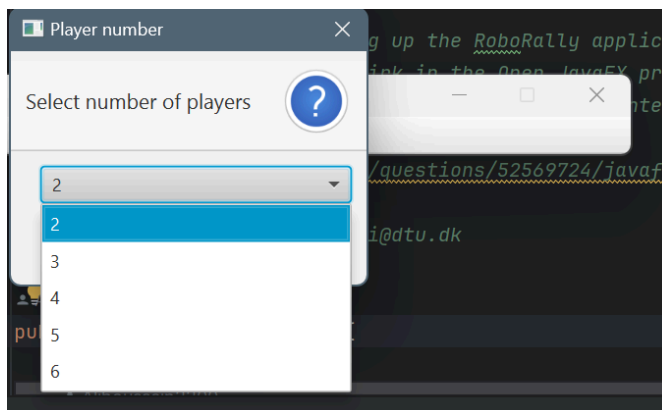
1. Når man har sin IntelliJ klar, starter man spillet ved at trykke på play Ikonet:



2. Nu vælg File > New game, hvis man vil indhente et tidligere spil trykker man "Load game"



3. Nu har du mulighed for at vælge antal spillere, efter dette kommer din spilleplade op med det antal spillere du har valgt, og du er nu klar til at spille:



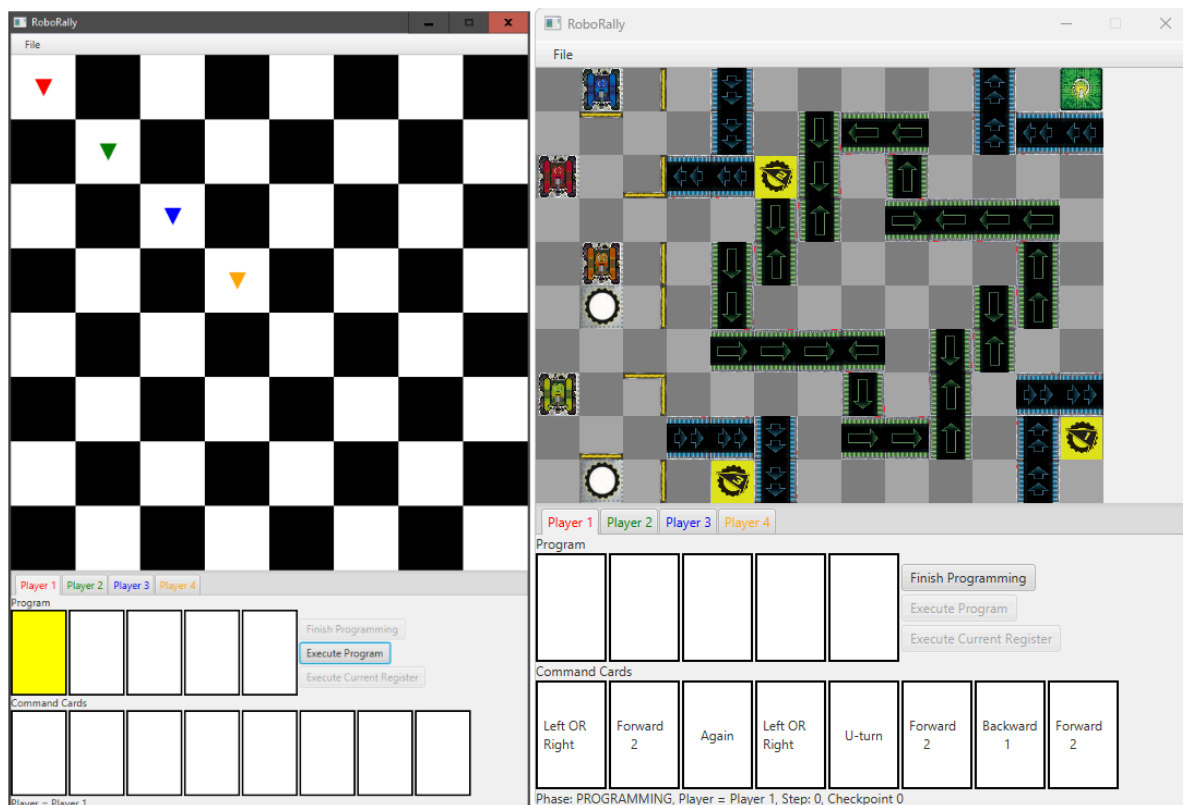
4. Herefter kan spillet påbegyndes og alle brikker skulle gerne have tildelt hver deres startposition.
5. Nederst på siden kan pointtælleren, samt info ses for hver spiller.
6. Derefter er spillet klar til brug og du skal nu følge reglerne vedhæftet.

Ordbog

ORD	Fortolkning/betydning
Conveyor belt	Transport af robot ved hjælp af transportbånd
Checkpoints	Spilleren må erhverve checkpoints i en sekventiel orden. for at nå et checkpoint, skal spilleren befinde sig ved enden af en række. Spilleren har mulighed for at nå checkpoints fra enhver retning.
Token	Spillerens brik
Programmeringskort	Programmeringskortene udgør de kort, som spilleren anvender til at styre deres robots bevægelser
Programmeringsfasen	Spillerne trækker programmeringskortene og arrangerer dem i den ønskede rækkefølge for at planlægge deres bevægelser, som de ønsker, at deres robot skal udføre.
Væg/walls	En væg hvorfra spillerne ikke kan bevæge sig gennem.
Board elements	Ved slutningen af hvert register aktiveres et bræt-element. Dette element påvirker kun en robot, når den befinder sig ved enden af et register.

Konklusion

I dette projekt har vi udviklet en version af brætspillet RoboRally ved at følge fremgangsmåden “CDIO”. I forhold til de funktionelle krav er vi kommet i mål med alle “must have” og “should have” på nær “load game”. Her er vi kommet delvist i mål, vi har implementeret “load game” men spillet loader ikke spillernes håndkort og registerkort. Vi er påbegyndt “Could have” kravene, hvor vi er i gang med at implementere Pits og "Out of Map", altså hvis man går ud over spille brættets kant. Dette har vi dog ikke færdiggjort. Vores ene ikke-funktionelle krav er vi kommet i mål med. I forhold til moscow prioritering, har vi ikke fulgt den 100%. f eks er vi påbegyndt nogle “could have” krav, før vi har løst vores problem med “load game”. Vi er også kommet rimelig godt i mål med at gøre grafikken pænere, ved at sætte billedfiler ind, i stedet for at bruge javafx-shapes.



Før og efter billede

Bibliografi

02362 Project in software Development Spring 24: <https://learn.inside.dtu.dk/d2l/home/187724>

Model-View-Controller: [Java SE Application Design With MVC](#)

ROBORALLY Game Guide

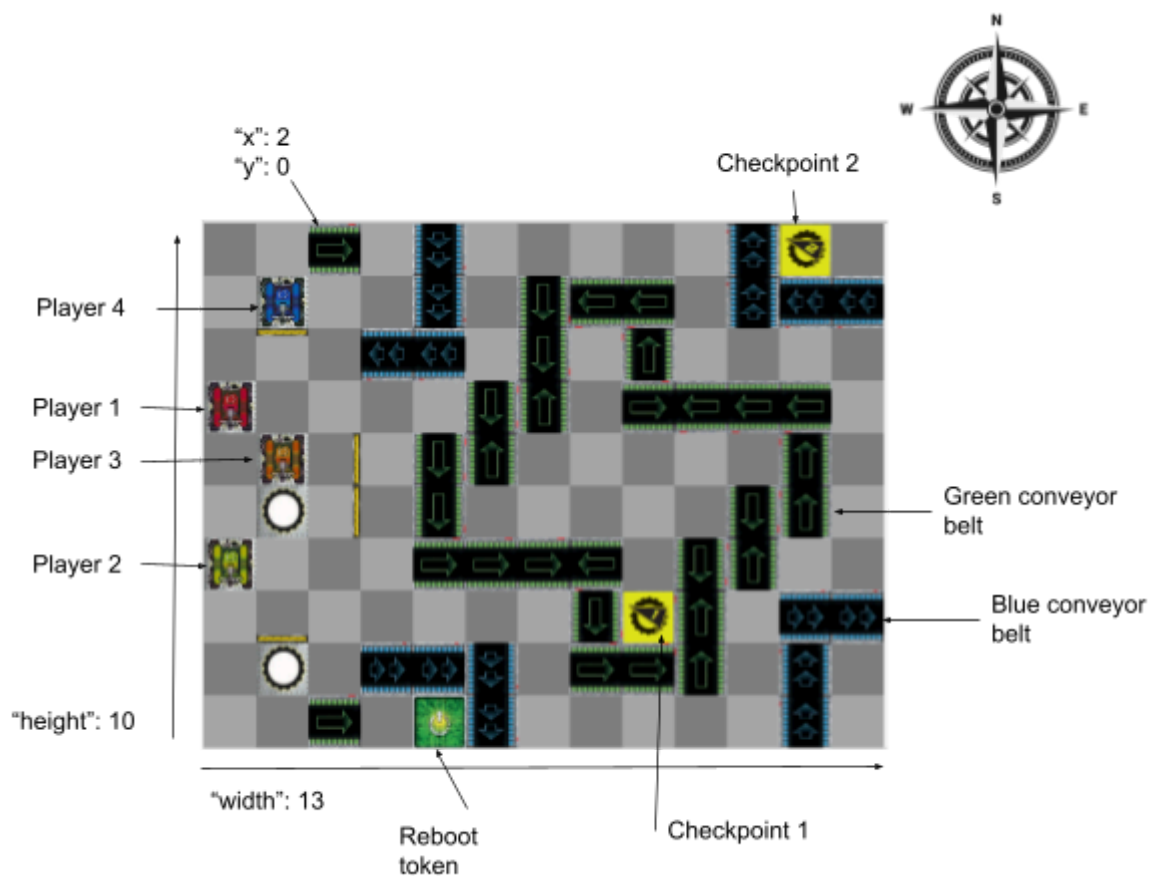
Bilag

Bilag 1 - Taksonomi

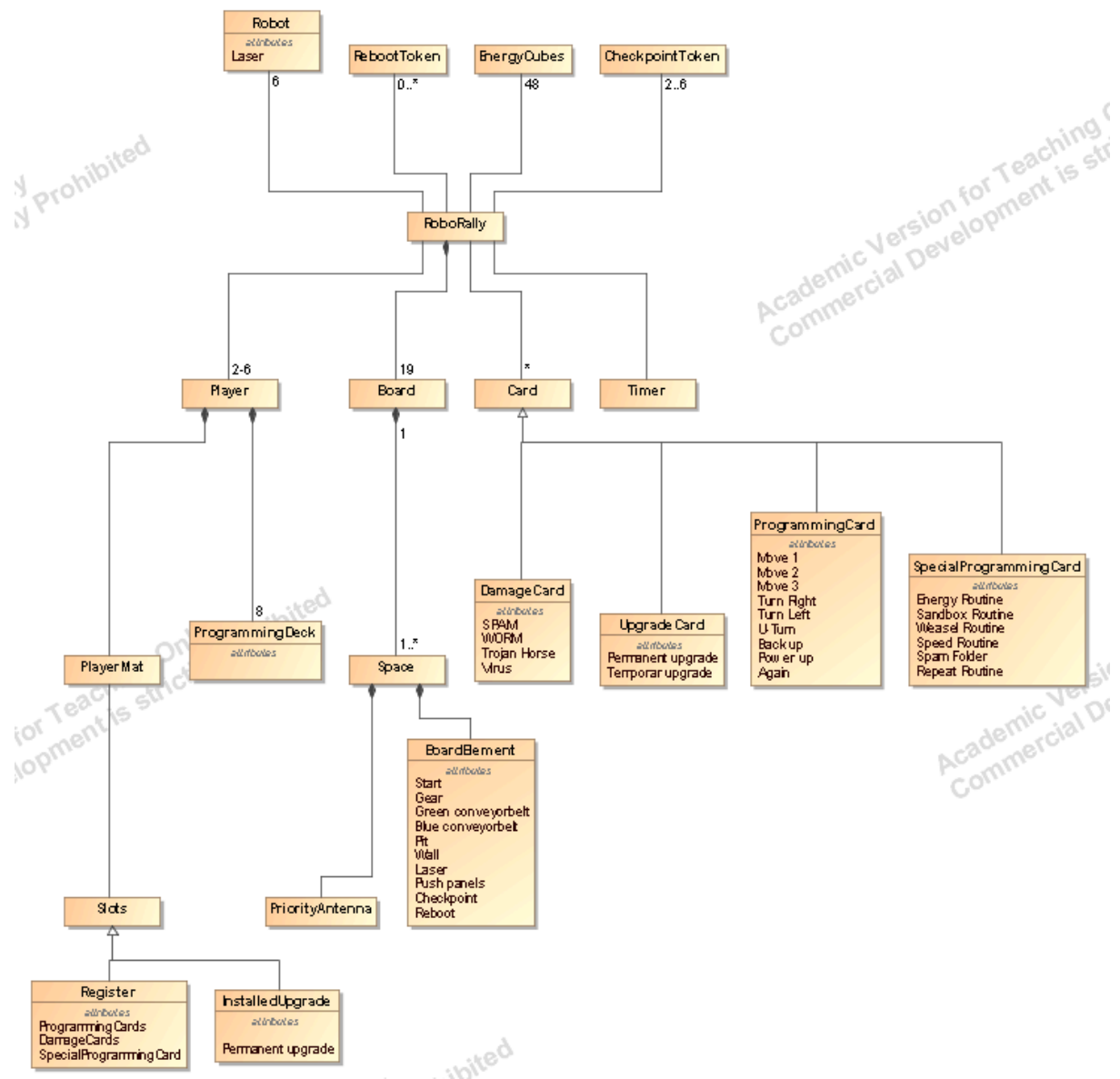
Navneord	Udsagnsord
<ul style="list-style-type: none">• Game• Board• Robot/Player• Program• Programming deck• ProgramCard• Space• Gamephase• Wall• Playermat• Boardelement• Robot direction• Wall• Conveyor Belt• Conveyor belt direction• Gear• Gear direction• Pit	<ul style="list-style-type: none">• Move Robot• Turn Robot• Push Robot• Fall (Robot)• Reverse Robot• Stop Robot• Activate Ability• Pick up checkpoint• Program Robot• Load game• Save game• Start Game

- Push panels
- Push panel direction
- Board laser
- Robot laser direction
- Robot laser
- Priority Antenna
- Special-programming card
- Damagecard
- EnergyCube
- Energy reserve
- Energy space
- Upgrade shop

Bilag 2 - RoboRally spil, Board: risky crossing, Antal spillere: 4



Bilag 3 - Domænemodel for det fuldstændige RoboRally spil



Bilag 4 - SQL script

```

2  SET FOREIGN_KEY_CHECKS = 0;;
3
4  CREATE TABLE IF NOT EXISTS Game (
5  gameID int NOT NULL UNIQUE AUTO_INCREMENT,
6
7  name varchar(255),
8
9  phase tinyint,
10 step tinyint,
11 currentPlayer tinyint NULL,
12 boardName varchar(50),
13
14 PRIMARY KEY (gameID),
15 FOREIGN KEY (gameID, currentPlayer) REFERENCES Player(gameID, playerId)
16 );
17
18 CREATE TABLE IF NOT EXISTS Player (
19 gameID int NOT NULL,
20 playerId tinyint NOT NULL,
21
22 name varchar(255),
23 colour varchar(31),
24
25 positionX int,
26 positionY int,
27 heading tinyint,
28 checkpoint tinyint,
29
30 PRIMARY KEY (gameID, playerId),
31 FOREIGN KEY (gameID) REFERENCES Game(gameID)
32 );
33
34 CREATE TABLE IF NOT EXISTS CardInPlayersHand (
35 GameID INT NOT NULL,
36 PlayerID TINYINT NOT NULL,
37 CardNo INT,
38 CardValue INT,
39
40 PRIMARY KEY(PlayerID, GameID, CardNo),
41 FOREIGN KEY(gameID, playerId) REFERENCES Player(gameID, playerId)
42 );
43
44 CREATE TABLE IF NOT EXISTS CardInPlayersRegister (
45 GameID INT NOT NULL,
46 PlayerID TINYINT NOT NULL,
47 RegisterNo INT,
48 CardValue INT,
49
50 PRIMARY KEY(PlayerID, GameID, RegisterNo),
51 FOREIGN KEY(gameID, playerId) REFERENCES Player(gameID, playerId)
52 );
53
54
55
56 SET FOREIGN_KEY_CHECKS = 1;;

```

Bilag 5 - DoAction()

```
66 @Override 5 usages
67 public boolean doAction(@NotNull GameController gameController, @NotNull Space space) {
68     Player currentPlayer = space.getPlayer();
69     ConveyorBelt conveyorBelt = (ConveyorBelt) space.getActions().get(0);
70     Heading currentHeading = conveyorBelt.getHeading();
71     if (currentPlayer != null) { // Tjekker om der er en spiller på pladsen
72         Space nextSpace = gameController.board.getNeighbour(space, currentHeading); // Finder den næste plads i retningen af transportbåndet
73         Player nextPlayer = nextSpace.getPlayer(); // Spilleren på den næste plads
74
75         if (nextPlayer == null) { // Hvis den næste plads er tom
76
77             gameController.pushRobot(currentPlayer, currentHeading); // Flytter den aktuelle spiller til den næste plads
78             return true;
79         } else {
80
81             if (gameController.board.getPlayerNumber(currentPlayer) < gameController.board.getPlayerNumber(nextPlayer)) { // Kontrollerer spillernes nummer i spillerlisten for at undgå gentagelser
82
83                 if (!nextSpace.getActions().isEmpty()) {
84
85                     if (nextSpace.getActions().get(0).getClass() == ConveyorBelt.class) { // Hvis der er endnu et transportbånd på den næste plads
86
87                         nextPlayer.setActivated(true); // Markerer den næste spiller som aktiveret for at undgå gentagelser
88
89                         doAction(gameController, nextSpace); // Rekursivt kald af metoden
90
91                         gameController.pushRobot(currentPlayer, currentHeading); // Flytter den nuværende spiller til den næste plads
92                     } else {
93
94                         gameController.pushRobot(currentPlayer, currentHeading); // Flytter den nuværende spiller til den næste plads
95                     }
96                 } else {
97
98                     gameController.pushRobot(currentPlayer, currentHeading); // Flytter den nuværende spiller til den næste plads
99                 }
100                 return true;
101             } else {
102
103                 if (nextPlayer.getSpace() == nextPlayer.getPrevSpace()) { // Hvis den næste spiller er tilbage på sit forrige plads
104
105                     gameController.pushRobot(currentPlayer, currentHeading); // Flytter den nuværende spiller til den næste plads
106                     return true;
107                 } else {
108
109                     nextPlayer.setSpace(nextPlayer.getPrevSpace()); // Flytter den næste spiller tilbage til sin forrige plads
110                     return false;
111                 }
112             }
113         }
114     }
115     return false;
116 }
```