


# JANUARPROJEKT

---

02121 INTRODUKTION TIL SOFTWARETEKNOLOGI

## Afleveringsgruppe 16:

Abinav Aleti s224786  
Yahya Alwan s224739  
Aslan Behbahani s224819  
Rasmus Wiuff s163977  
[github.com/rwiuff/Reversi](https://github.com/rwiuff/Reversi) 


20. januar 2023

# INDHOLD

	Side
<b>1 Introduktion &amp; Problemanalyse</b>	<b>1</b>
1.1 Tilgængelighed	1
1.2 Forfatterskab	1
1.3 Opgaven	1
1.4 Specifikationer	1
1.5 Målsætninger	1
1.6 Tilgang til opgaveløsningen	2
<b>2 Design</b>	<b>2</b>
2.1 MVC	2
2.2 Main	3
2.3 Controller	3
2.4 Board	3
<b>3 Implementering</b>	<b>3</b>
3.1 Board	3
3.2 Main	5
3.3 Controller	5
<b>4 Test</b>	<b>8</b>
4.1 Board-driver klassen	8
4.2 Unit tests	8
<b>5 Project Management</b>	<b>8</b>
5.1 Tidsplan og tidsstyring	8
5.2 Arbejdsfordeling	9
5.3 Git, Github og Gradle	10
<b>6 Konklusion</b>	<b>10</b>
<b>Figurer</b>	<b>11</b>
<b>Tabeller</b>	<b>11</b>
<b>Appendicer</b>	<b>12</b>
<b>A Afsnitsforfattere</b>	<b>12</b>

## 1 INTRODUKTION & PROBLEMANALYSE

### 1.1. TILGÆNGELIGHED

Hele projektet (inklusive denne rapport) er tilgængeligt online på [Github](#) . I mappestrukturen er *“BasicReversi”* og *“AdvancedReversi”* rodmapper for hver deres projekt som kan importeres i Eclipse og Visual Studio Code. Læs mere i Afsnit 5.3.

### 1.2. FORFATTERSKAB

En oversigt over hvilke gruppemedlemmer der har skrevet hvad i rapporten kan ses i Appendiks A.

### 1.3. OPGAVER

I dette projekt er målet at udvikle et program der implementerer spillet “Reversi”; et brætspil der handler om at vinde mest muligt terræn ved at indkapsle og vende modstanderens brikker<sup>1</sup>. Vi benytter os af regelsættet i projektoplægget, og jf. samme oplæg skrives den grafiske brugerflade med JavaFX.

Med udgangspunkt i Model-Viewer-Controller konceptet (*MVC*) skal der implementeres en klasse med den underliggende logik for spillet, en klasse der implementerer den grafiske brugerflade, og en klasse der binder disse sammen.

### 1.4. SPECIFIKATIONER

Reversi er et relativt simpelt spil med få regler. Vi ved, at vi skal lave et bræt med 8x8 felter. I begyndelsen af spillet skal der placeres 4 brikker af to forskellige farver, én farve pr. spiller. Herefter skal de to spillere placere deres brikker én efter én, og vende modstanderens brikker ved at klemme dem inde mellem ensfarvede brikker. Brikkerne kan kun stilles et sted, hvor man med sikkerhed klemmer en brik af modsat farve inde. Vinderen er den spiller, der har flest brikker på brættet, når alle 64 felter er udfyldt. Alternativt kan man vinde spillet, ved at modstanderen siger ”pas” to ture i træk.

- Begyndende spiller: Tildeles tilfældigt mellem de to spillere. Ved efterfølgende ture, skiftes spillerne.
- Startkonfiguration: Første spiller vælger to placeringer inden for de midterste fire felter. Næste spillers brikker optager resterende to felter og første spiller har næste træk.
- Placering af brikker: En brik må kun ligges på et tomt felt således at en eller flere af modstanderens brikker indkapsles mellem den lagte brik og en af spillers andre brikker. Indkapslingen kan ske ad vertikale, horisontale og diagonale retninger.
- Vending af brikker: Ved placering af en brik vil alle modstanderens, i det øjeblik, indkapslede brikker mellem den lagte brik og spillerens egne brikker blive vendt.
- Spillets afslutning: Når brættet er fyldt eller begge spillere har meldt pas efter hinanden.
- Vinder: Den spiller som har flest brikker (og dermed vundet mest terræn) har vundet spillet.
- Spillet skal være indeholdt i et eksekverbart jar-arkiv som skal kunne køre på alle maskiner (selv uden at JavaFX er installeret på end-user maskinen).

Ud fra disse specifikationer bliver programmet udviklet. Der er artistisk frihed ift. brikkers og brættets farve, hvor stort brættet skal være ift. computerskærmen og hvordan layout’et skulle se ud mht. placering af eventuelle knapper og tekstområde ift. selve brættet.

### 1.5. MÅLSÆTNINGER

Projektets primære mål er at udvikle et program der overholder specifikationerne ovenfor. Spillet har skulle kunne køre, og været testet adskillige gange for robusthed og eventuelle små fejl.

Krav til brugerfladen er at letlæselighed, overskuelighed, og simpel struktur, uden unødvendig redundans. Programmets struktur skal ligeledes være simpelt og uden redundans, således at det er nemt at redigere i koden, for at implementere ændringer undervejs, da dette kan blive en stor udfordring, skulle man have skrevet noget

<sup>1</sup>Reversi på Wikipedia

sammenflettret og kompliceret<sup>2</sup>.

Sekundære mål er at implementere så mange valgfri features som muligt, som vi dømte inde for vores faglige kapacitet. Igen hjælper det her med simpel og overskuelig kode, der gør, at man let kan genbruge noget, man tidligere har skrevet, til at udvide og forbedre programmet.

Et andet vigtigt mål er at arbejde som en gruppe. Med det menes der at lære, hvorledes man arbejder sammen, dele arbejdet op når det kommer til et projekt af denne størrelse, og tage stilling til hvordan ens kode kan sammenflettes med de andre gruppemedlemmers kode.

---

## 1.6. TILGANG TIL OPGAVELØSNINGEN

Projektet har budt på adskillige overvejelser ift. løsning af problemer. Som det første havde vi det visuelle, og dernæst havde vi det logiske. Vi blev enige om at dele arbejdet op i grupper af to og to. Så kunne vi spørge os selv: Hvordan vil vi gribe vores opgave an? Hvad angår den grafiske del, besluttede vi os for at bruge *Scene Builder*<sup>3</sup> til at lave vores *Scene* og nogle af vores *Nodes*. Fordelen her er at have al grafisk info stående på en let overskuelig måde og kun definere interaktionen i vores klasser. Ulempen vil dog være, at det vil tage længere tid at skrive hver enkelt ting op (specificere en knaps layout, størrelse mm). En anden fordel ved Scene Builder er, at det er hurtigt, overskueligt (inde i selve programmet), og det er let at forbinde evt. kode fra andre klasser med det, man har designet i programmet.

Dog er det ikke alt, der ikke let kan løses i Scene Builder. Detaljer som actions og id på nodes er nemmere at redigere direkte i *FXML* filen efter layoutet er defineret i Scene Builder.

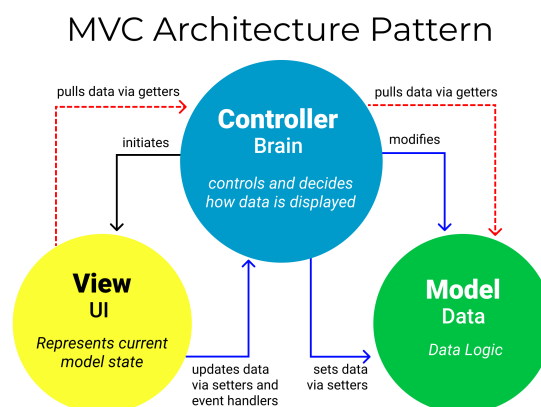
## 2 DESIGN

---

### 2.1. MVC

Vi har som løsning til opgaven valgt at lave 3 klasser. Disse klasser er skabt efter 'Model-View-Controller' - mønstret. Derudover har vi også en *.fxml* fil, der indeholder al information om vores stage og scene. Denne fil redigeres igennem Scene Builder. Dette vælger vi dog ikke at gøre, da det, i vores optik, vil være mere praktisk at designe det grafiske i scene-builder, da man kan placere diverse objekter/nodes som 'Buttons', 'Panels' osv. på 'scene' og redigere dem, uden at skulle have en eksakt viden om de specifikke koordinater, hvorpå vi stiller dem, og uden at skulle skrive deres længder, bredder osv. ned.

Figur 1: Model Viewer Controller konceptet (af Rafael D. Hernandez) [freecodecamp.org](https://www.freecodecamp.org)



Vores klasser er de tre følgende:

---

<sup>2</sup>Spaghettikode

<sup>3</sup>Scene Builder

## 2.2. MAIN

En main-klasse, der fungerer som view; altså alt det, som spilleren kan se. Den sætter altså programmet igang, viser stage, scene, nodes osv. En board-klasse, der fungerer som model; den står for alt al logik, der bruges til at implementerer de forskellige regler og funktioner ved spillet, som f.eks. at vende brikker, der er indeklemte, eller at afgøre, om hvorvidt man kan stille en brik på et bestemt felt.

## 2.3. CONTROLLER

En controller-klasse, der fungerer controller; denne klasse binder board og .fxml-filen sammen. Dette gør den ved at lave en masse metoder, der responderer på et klik med musen eller et klik på en "Button" på "scene", således at der sker nogle ændringer visuelt, og disse ændringer har forklaring med rod i logikken (board-klassen).

## 2.4. BOARD

Som model i MVC konceptet laves en klasse der indeholder brættet som en datastruktur, og som indeholder logikken til at placere brikker, hvilke af modstanderens brikker der bliver indeklemmt samt lovligheden af et givent træk. Board klassen skal kommunikere med controller klassen igennem en række heltals koder.

# 3 IMPLEMENTERING

## 3.1. BOARD

Modellen for brættet implementeres i `Board.java` klassen. Tabel 1 indeholder klassens felter.

*Tabel 1: Felter i `Board.java`*

Felt	Formål
<code>int[][] board</code>	Datastruktur der repræsenterer brættet.
<code>int turnCount</code>	Turnummeret i spillet
<code>int boardsize</code>	Størrelsen $n$ for et $n \times n$ bræt.
<code>int forfeitCounter</code>	Antal gange en spillerne har meldt pas
<code>ArrayList&lt;int[]&gt; flipped</code>	Log over de sidste vendte brikker
<code>HashMap&lt;Integer, String&gt; players</code>	Spillernes interne ID og navne
<code>HashMap&lt;String, HashMap&lt;Integer, HashMap&lt;Integer, Integer[]&gt;&gt;&gt; validMoves</code>	Datastruktur for mulige træk for en given farve, en given tur.

Brættet repræsenteres af en heltalsmatrix med værdierne 0 for tomt felt, 1 for hvid placering og 2 for sort placering. Ved at gemme turnumre og antal gange spillere har meldt pas kan board klassen afgøre hvis tur det er og om spillet er slut. Loggen over vendte brikker endte med ikke at blive brugt, men formålet var at kontrollerklassen skulle vise hints om hvilke brikker der ville blive vendt ved et givent træk. Det blev stemt ned da det gør spillet for nemt. Tabel 2 indeholder klassens metoder.

Tabel 2: Metoder i *Board.java*

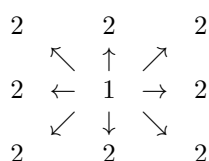
Metode	Formål
<code>Board()</code>	Konstruktør for brættet. Kalder <code>setPlayers()</code> .
<code>getBoard()</code>	Getter for board feltet.
<code>setPiece(int row, int column, int colour)</code>	Sætter manuelt brik, uanset lovlighed. Bruges til unit tests.
<code>getTurn()</code>	Getter for <code>turnCount</code> .
<code>turnClock()</code>	Inrementere <code>turnCount</code> .
<code>getPlayers()</code>	Getter for spiller hashmap.
<code>getValidMoves()</code>	Getter for <code>validMoves</code> hashmap.
<code>setPlayers()</code>	Tildeler tilfældigt spillere en farve.
<code>setPlayers(int id1, String player1, int id2, String player2)</code>	Tildeler spillere specifikke farver.
<code>setPlayerName()</code>	Setter for specifikt spillernavn.
<code>resetBoard()</code>	Nulstiller brættet, turnummer og pastæller.
<code>turnState(int colour)</code>	Kalder <code>moveAnalyser(colour)</code> , og udregner om der er ledige træk.
<code>gameOver()</code>	Afgør om spillet er slut hvis brættet er fyldt, eller der er meldt pas to gange i træk.
<code>initPlace(int row, int column, int colour)</code>	Indeholder logik for at udfylde startfelterne.
<code>place(int row, int column, int colour)</code>	Placere en brik hvis placeringen er tom og indeholdt i <code>validMoves</code> . Kalder <code>flip(move, colour)</code> .
<code>flip(String move, int colour)</code>	Vender brikker ved et lovligt træk.
<code>filled()</code>	Afgører om brættet er fyldt.
<code>checkWinner()</code>	Tæller brikker og afgører vinderen.
<code>moveAnalyser(int colour)</code>	Finder lovlige træk (se Afsnit 3.1.2).
<code>findOwn(int[] checkboard, int i, int j, int direction, int colour)</code>	Findere mulig indkapsling af modstanderen.
<code>saveFlips(int[] ownPiece, int i, int j, int direction)</code>	Gemmer indkapslede brikker.

De fleste metoder er simple if/else eller for-løkker, og de er vel dokumenterede i kildekoden. Den vigtigste funktion i *Board.java* er `moveAnalyser(int colour)` metoden som skal finde og afgøre alle træks lovlighed for et givent bræt setup og en given brikfarve. Metoden gennemgås i Afsnit 3.1.2.

**3.1.1. AdvancedReversi board klasse** Der er enkelte ændringer i den avancerede udgave af spillet. `setPlayers()` metoden ændres til ikke at foretage tilfældige valg, da brugeren selv skal skrive navne ind i systemet. Pastælleren, `gameOver()` og `filled()` fjernes helt, for at overlade denne logik til controller klassen da dette bliver en mere integreret del af brugerinteraktionen. Til gengæld introduceres to metoder: `checkBlackScore()` og `checkWhiteScore()` som bruges til at live-opdatere optaget terræn for de to spillere.

**3.1.2. moveAnalyser() metoden og validMoves feltet** Flere gange i løbet af spillets afvikling er det praktisk at vide om hvilke træk der er lovlige for en given farve. Betragt Figur 2. Her ses at en vilkårlig brik har 8 nabofelter. Særligt fælde er i kanten af brættet. For at undgå særligt fælde blev første implementationsvalg her at udfører alle tests i et bræt med *padding*. Derfor starter `validMoves` ud med at lave en kopi af brættet med en kant af nuller.

Figur 2: En briks naboer



Herefter itererer metoden over alle felter i brættet. Mødes et tomt felt testes alle 8 retninger for en modstander. Hver retning har sin egen heltalskode. Hvis der er en modstander itererer metoden i den retning og leder efter en brik af egen farve for at fastslå en indkapslingsmulighed, med metoden `findOwn()`. Hvis metoden støder på kanten af brættet eller en tom placering indstilles søgningen. Dennes algoritme beskrives mere grafisk i Figur 3.

**Figur 3:** `findOwn()` leder efter en indkapsling i nordvestlig retning (heltalskode 1).

(a) Første skridt	(b) Andet skridt	(c) Tredje skridt																																																
<table><tr><td>1</td><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>2</td><td>0</td><td>0</td></tr><tr><td>0</td><td>0</td><td>2</td><td>0</td></tr><tr><td>0</td><td>0</td><td>0</td><td>1</td></tr></table>	1	0	0	0	0	2	0	0	0	0	2	0	0	0	0	1	<table><tr><td>1</td><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>2</td><td>0</td><td>0</td></tr><tr><td>0</td><td>0</td><td>2</td><td>0</td></tr><tr><td>0</td><td>0</td><td>0</td><td>1</td></tr></table>	1	0	0	0	0	2	0	0	0	0	2	0	0	0	0	1	<table><tr><td>1</td><td>0</td><td>0</td><td>0</td></tr><tr><td>0</td><td>2</td><td>0</td><td>0</td></tr><tr><td>0</td><td>0</td><td>2</td><td>0</td></tr><tr><td>0</td><td>0</td><td>0</td><td>1</td></tr></table>	1	0	0	0	0	2	0	0	0	0	2	0	0	0	0	1
1	0	0	0																																															
0	2	0	0																																															
0	0	2	0																																															
0	0	0	1																																															
1	0	0	0																																															
0	2	0	0																																															
0	0	2	0																																															
0	0	0	1																																															
1	0	0	0																																															
0	2	0	0																																															
0	0	2	0																																															
0	0	0	1																																															

Den fundne nabobriks koordinater gemmes og returneres til `validMoves`. Hvis `validMoves` modtager sådan et koordinat iværksættes `saveFlips` som gemmer de indkapslede brikker i et hashmap efter samme skridt som i Figur 3. Her vil de røde brikkeres koordinater blive gent.

Når `validMoves` har et hashmap med hashmaps af koordinater fra alle retninger gemmes disse i `validMoves` hashmappet med trækets koordinat som nøgle. Tabel 3 viser strukturen af `validMoves`.

**Tabel 3:** Datastrukturen `validMoves` efter afsøgningen i Figur 3

Træk	Retning	Vendbare brikker
<code>HashMap&lt;String,</code>	<code>HashMap&lt;Integer,</code>	<code>HashMap&lt;Integer, Integer[]&gt;&gt;</code>
"3,3"	1	<code>0,{2,2}</code> <code>1,{1,1}</code>

Årsagen til at retning indgår som nøgle i `validMoves` er simpel: Alle værdier skal have en unik nøgle. Hvis ikke f.eks. retning blev brugt ville `moveAnalyser` bare overskrive brikker fundet i en retning med brikker fundet i en tidligere retning.

## 3.2. MAIN

### 3.2.1. Basic

### 3.2.2. Avanceret

**3.2.3. FXML** Brættet er lavet af en 8x8 gridpane med en pane i hver celle. Hver pane indeholdt i gridpanes cellen er tildelt et unikt id. For eksempel er 05 et unikt id tildelt til en pane, hvor 0 repræsenterer rækkeindekset, og 5 repræsenterer kolonneindekset for panes placering i gridpane. Hver pane er tildelt et id for at gøre det nemmere at placere/tegne eller slette en brik fra brættet.

## 3.3. CONTROLLER

### 3.3.1. Basic

**3.3.2. Avanceret** I den avancerede controller klasse er der rykket om på rækkefølgen, hvori metoderne kaldes. Nedenfor ses en tabel, der viser kronologien af de "overordnede" metoder", samt de metoder, de kalder.

*Tabel 4*

Metode	Kalder metoderne:	evt. note
beginGame()	setName()	Denne metode er interessant og unik ift. controller-klassens da den agerer som en metode, man under normale omstændigheder vil se i en "main-klasse, eftersom at metoderne der fungerer som en "opstart", hvor de fremviser stage og scene.
setName()	setNameBtn() checkScore()	
setNameBtn()	in()	
in()	drawCircle()	tildeler spillerne deres farver og viser de fire første mulige træk
firstFour()	update() checkscore()	Får spiller til at sætte de fire første brikker
onPaneClicked()	getRowIndex() getColumnIndex() firstFour() hideLegalMoves() update() checkScore() showLegalMoves()	Styrer hvad der sker, når der bliver klikket på et felt Giver spillerne besked, om hvorvidt deres træk er lovlige Melder pas, når der ikke er noget lovligt træk Annoncerer spillets slutning, når der ikke er nogen lovlige træk tilbage
gameOver()	loadHighScore() checkHighScore()	Erklærer vinderen og opdaterer high-score'n (hvis den er blevet slået)



*Tabel 5: Metoder i Controller.java*

Metode	Kaldte metoder	Beskrivelse
restart(ActionEvent event)	reset() setName()	Genstarter spillet og beder spillerene om at indsatte deres navne i tekstfelte igen. Spiller der lavet første træk i sidste runde, begynder anden
reset()		Iterer gennem brættet og rydder det
update()	drawCircle()	Opdatere brættet og tegner cirkler(brikker) på brættet ifølge Board objektet
exitGame(ActionEvent event)		lukker spille
checkScore()		Opdaterer scoren for begge spillere
saveHighScore(int score,String name)		Gemmer vinderens score og navnet adskilt af komma inde i "Highscore.txt" file, hvis vinderens score er højre end nuværende highscore
loadHighScore()		Læser highscore og spillersnavne fra "Highscore.txt" filen og returnere de som arrayet.
showHighScore(ActionEvent event)	loadHighScore()	Når man trykker på highscore-knappen, vises en advarselsboks/-prompt, der viser highscore og spillerens navn, hvis highscore er indstillet.
surrender(ActionEvent event)		Annoncere overgivelsen af den spiller der trykkede på knappen, og den anden spillers sejr
getRowIndex(MouseEvent event)		Returnerer rækkeindekset for pane, der er klikket på
getColumnIndex(MouseEvent event)		Returnerer søjleindekset for pane, der er klikket på
showLegalMoves(int color)	drawCircle()	Tegner en cirkel hvor spiller kan mulig placer en brik
hideLegalMoves()		Rydder brættet for cirkler, hvilket indikerer mulige træk
mainMenu(ActionEvent event)		Returner tilbage til main-menu af spillet

**3.3.3. Genstarter Spillet** Et af de grundlæggende kriterier for det basis Reversi-spil er evnen til at genstarte spillet uden at genstarte applikationen. Det er gjort muligt at genstarte spillet uden at genstarte programmet ved at oprette en knap. Genstart-knappen havde oprindelig funktionalitet til at genstarte spillet ved at skifte scener, det vil sige at skifte til scenen, hvor ingen brikker er placeret på brættet. Den tom brættet er den samme scene, der vises ved kørsel af FXML-filen i starten af spillet/programmet. Genstart af spillet med at skifte scener var ikke et optimalt og vellykket middel til at genstarte spillet, da det ikke tillod brugeren at spille spillet igen uden at lukke programmet, på grund af den uretmæssige initialisering af spillet samt kun indlæse scenen (tom brættet) med ingen funktionalitet.

For at undgå denne ukorrekte genstart blev hver pane indeholdt i GridPane tildelt et id (fx: 05 repræsenterer Pane i 0. række og 5. kolonne i GridPane). Hver gang man trykker på genstart, vil to for-loops iterere gennem hver række og kolonne i GridPane, få adgang til den unikke identifikator for hver Pane ved hjælp af række- og kolonneindekset og derefter rydder brikkerne på den. At tildele et id til ruderne og bruge for-loops til at iterere gennem Pane og

slet brikkerne placeret på dem, gav den mest effektive og effektive metode til at genstarte spillet igen. Derudover, når spillet genstartes, vil den spiller, der foretager det første træk, før spillet genstartes, nu starte som nummer to.

## 4 TEST

Når vi skulle teste programmet, havde vi forskellige tilgange til det, alt efter hvad vi arbejdede med. Os, der arbejdede med det visuelle, skulle bare åbne programmet og se, om vi var tilfredse med overfladen, og om de diverse knapper gjorde som de skulle, når vi klikkede på dem. Vi skulle ikke arbejde med nogen kompliceret logik.

---

### 4.1. BOARD-DRIVER KLASSEN

Board-driver klasse, der vil være i stand til at kunne teste "board"klassens funktionsdygtighed. Dette gør vi, eftersom der ikke vil være mulighed for at teste logikken rent visuelt, før controller/scene-builder delen er færdig. Dette skyldes, at controller-klassen modtager og arbejder med data fra board, og hvis controllerklassen så ikke er færdig, er den ikke i stand til at gøre dette på en ordentlig måde.

---

### 4.2. UNIT TESTS

For at tilgodese at Board klassens logik virker efter ændringer blev en testklasse, `BoardTest.java` skrevet som kører hver gang et jar-arkiv kompiles eller klassen køres. Testen tester setter og getter metoder for player hashmappet, `moveAnalyser` og `turnState` metoderne.

## 5 PROJECT MANAGEMENT

I følgende afsnit vil der blive diskuteret de valgte løsninger til at organisere projektet og løse opgaven.

---

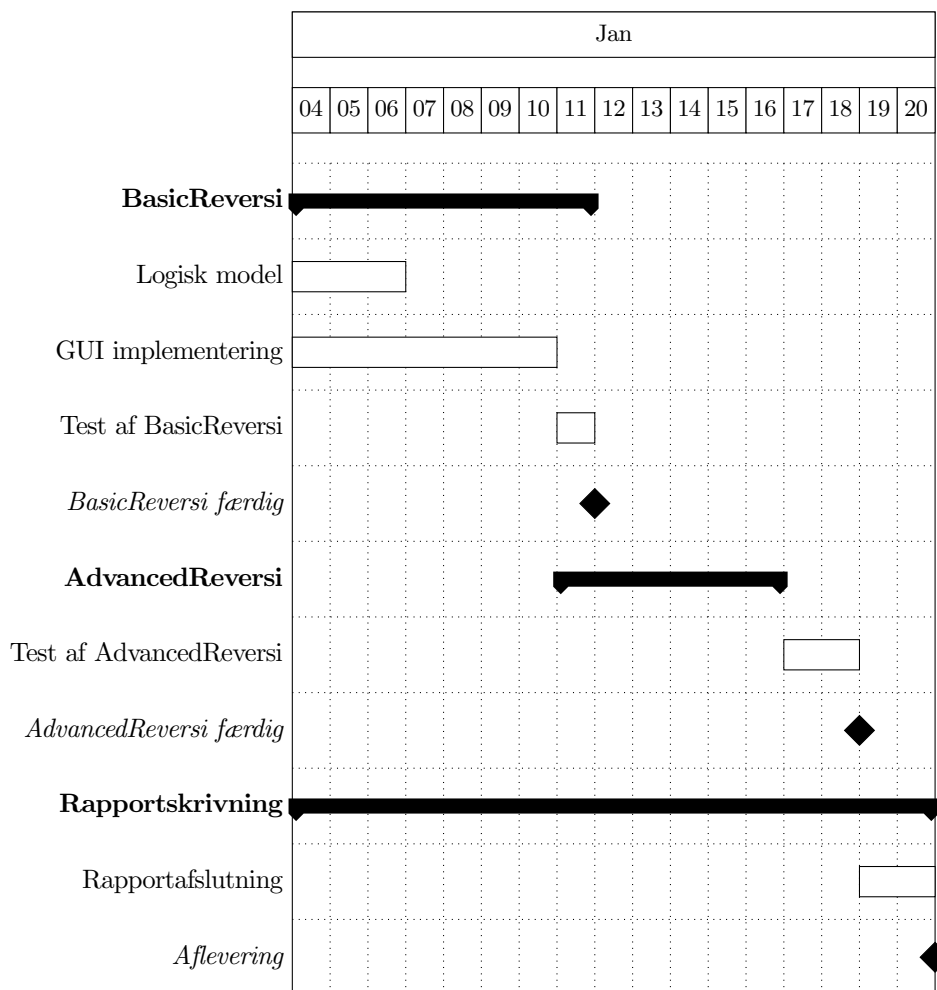
### 5.1. TIDSPLAN OG TIDSSTYRING

Indledningsvist blev gruppen enige om en række principper og en tidsplan for at optimere udviklingsplanen. Principperne lyder:

1. Nær-daglig status på underopgaver.
2. Alle planer er tentative.
3. Fælles udvikling af logik og algoritmer i vigtige metoder.

Den oprindelige tidsplan kan ses i Figur 4.

Figur 4: Første tidsplan (fra Projektplanen)



Under projektet overholdte vi ovennævnte principper, men tidsplanen ændredes markant. BasicReversi var færdig to dage over tid og rapportskrivningen blev lempeligt udskudt til den sidste uge.

## 5.2. ARBEJDSFORDELING

I forhold til arbejdsfordelingen er projektet opdelt i tre faser:

1. Logik & Brugerflade
2. Controller klassen
3. AdvancedReversi funktionaliteter

I første fase arbejder gruppen i makkerpar. To personer nærstudere JavaFX og prøver at implementere den ønskede brugerflade og giver deres *concerns* til det andet makkerpar. Det andet makkerpar arbejder på at implementere logikken bag brættet og brikkerne i en model klasse.

I anden fase arbejder gruppen sammen på at løse forskellige problemstillinger. En kunne f.eks. arbejde på den overordnede struktur af metodekald i et spil Reversi, en anden arbejder på at oversætte museklik til koordinater, en tredje arbejder på reaktioner fra knapper, og en sidste på opdatering af det grafiske bræt ud fra den interne tilstand af bræt objektet. Til sidst en større sammenfletning som leder til en controller klasse, og efter test af spillet, et færdigt produkt. I tredje fase bliver det oprindelige spil forgrenet via Git og hvert gruppemedlem implementerer et af gruppens tilføjelser af gangen. Når en implementation er færdig flettes denne med hovedgrenen. Her er der valgt en person som mestendels arbejder med at flette de forskellige grene sammen.

### 5.3. GIT, GITHUB OG GRADLE

Som projektstyringsværktøjer er der valgt følgende løsninger:

- **git** [Git](#): Til at kommunikere mellem lokale maskiner og [Overleaf](#), hvor rapporten kan skrives i fællesskab.
- **Github**: Til at håndtere versionskontrol mellem medlemmer i gruppen.
- **Gradle**: Projektstyringsværktøj til at automatisere *dependencies* (JavaFX), mappestrukturer og pakning af jar-filer.

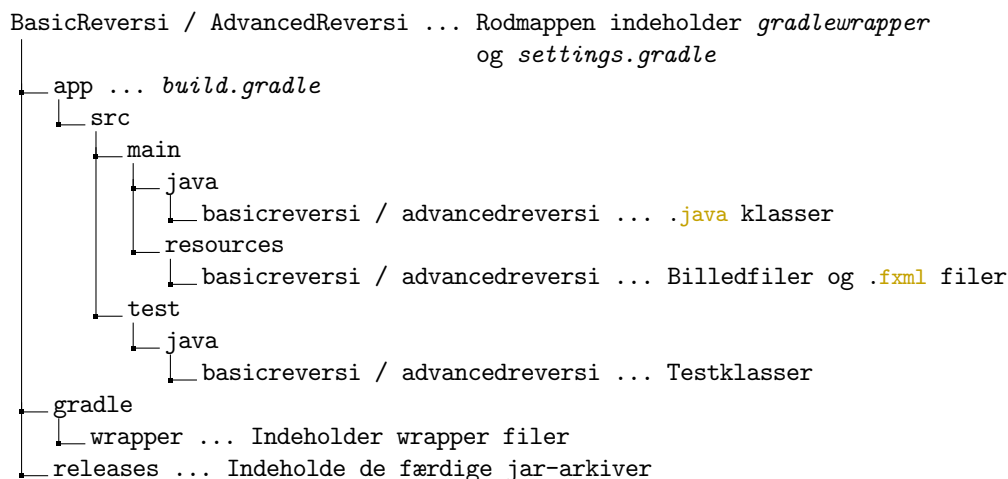
#### 5.3.1. Git og Github

Som nævnt i Afsnit 1.1 er projektet på Github <https://github.com/rwiuff/Reversi>. Dette *repository* (repo) er udgangspunkt for versionskontrol i projektet. Desuden bruges Overleafs Git integration til at synkronisere projektet med lokale maskiner. Git som versionstyringsprogram har været essentielt under projektet. Der har været meget læring i at bruge *branch* og *merge* funktionerne til at kunne arbejde på forskellige aspekter af kode i de samme klasser på samme tid. Det har mestendels været uden de største problemer, men der er til stadighed meget mere at lære om samarbejde med VCS.

#### 5.3.2. Gradle

Via Gradle kan mappestrukturen i Figur 5 genereres. Mappestrukturen indeholder desuden også bin og build mapper som bruges under kompilering og pakning af jar-arkiver, men disse er kun midlertidige instanser af det program man kører.

Figur 5: Mappestrukturen i JavaFX projekterne *BasicReversi* og *AdvancedReversi*



Gradle tilgodeser automatisk udførsel af skrevne tests, håndtering af kompilerede .class filer og pakning af jar arkiver. Desuden tilgodeser Gradle at dependencies er tilgængelige for miljøet der åbner projektet, uden at man skal importere JavaFX moduler manuelt. Er der en fejl i den version af JavaFX der bruges, kan man blot opdatere versionsnummeret i *build.gradle* og herefter opdateres JavaFX.

#### 5.3.3. Tidsplan og tidsstyring

## 6 KONKLUSION

## FIGURER

1	Model Viewer Controller konceptet (af Rafael D. Hernandez) <a href="https://www.freecodecamp.org">freecodecamp.org</a> ) . . . . .	2
2	En briks naboer . . . . .	4
3	<code>indexOf()</code> — leder efter en indkapsling i nordvestlig retning (heltalskode 1). . . . .	5
4	Første tidsplan (fra Projektplanen) . . . . .	9
5	Mappestrukturen i JavaFX projekterne BasicReversi og AdvancedReversi . . . . .	10

## TABELLER

1	Felter i <code>Board.java</code> — . . . . .	3
2	Metoder i <code>Board.java</code> — . . . . .	4
3	Datastrukturen <code>validMoves</code> — efter afsøgningen i Figur 3 . . . . .	5
4	. . . . .	6
5	Metoder i <code>Controller.java</code> — . . . . .	7
6	Forfatterskab i rapporten . . . . .	12

# Appendices

## A AFSNITSFORFATTERE

Tabel 6 viser en oversigt over hvilke gruppemedlemmer der har skrevet hvilke afsnit i rapporten.

*Tabel 6: Forfatterskab i rapporten*

Navn	Afsnit
Abinav Aleti	Afsnit 3.2.3 og 3.3.3 og Tabel 5
Yahya Alwan	
Aslan Behbahani	Afsnit 1.3, 1.5, 1.6 og 2.1 til 2.4
Rasmus Wiuff	Afsnit 1.1, 1.4, 3.1, 4.2 og 5