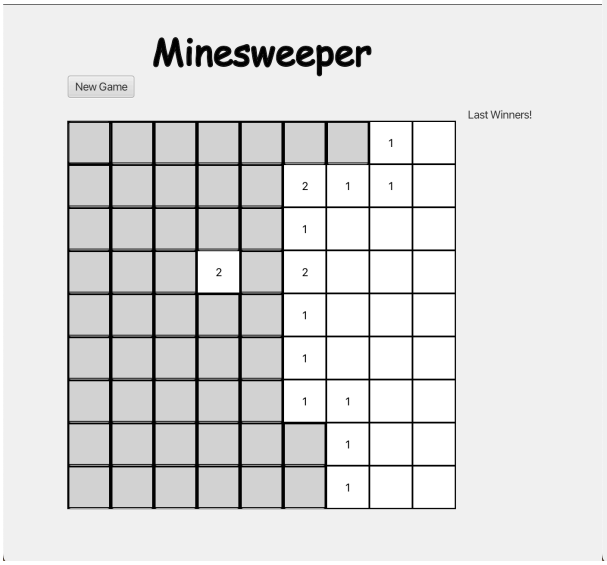
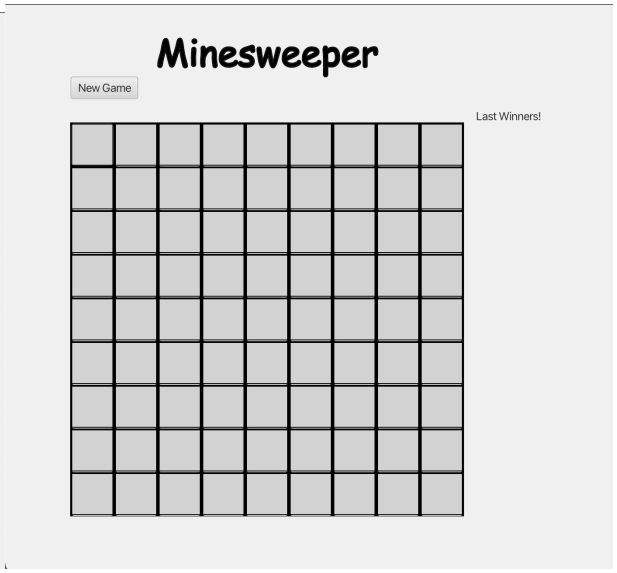
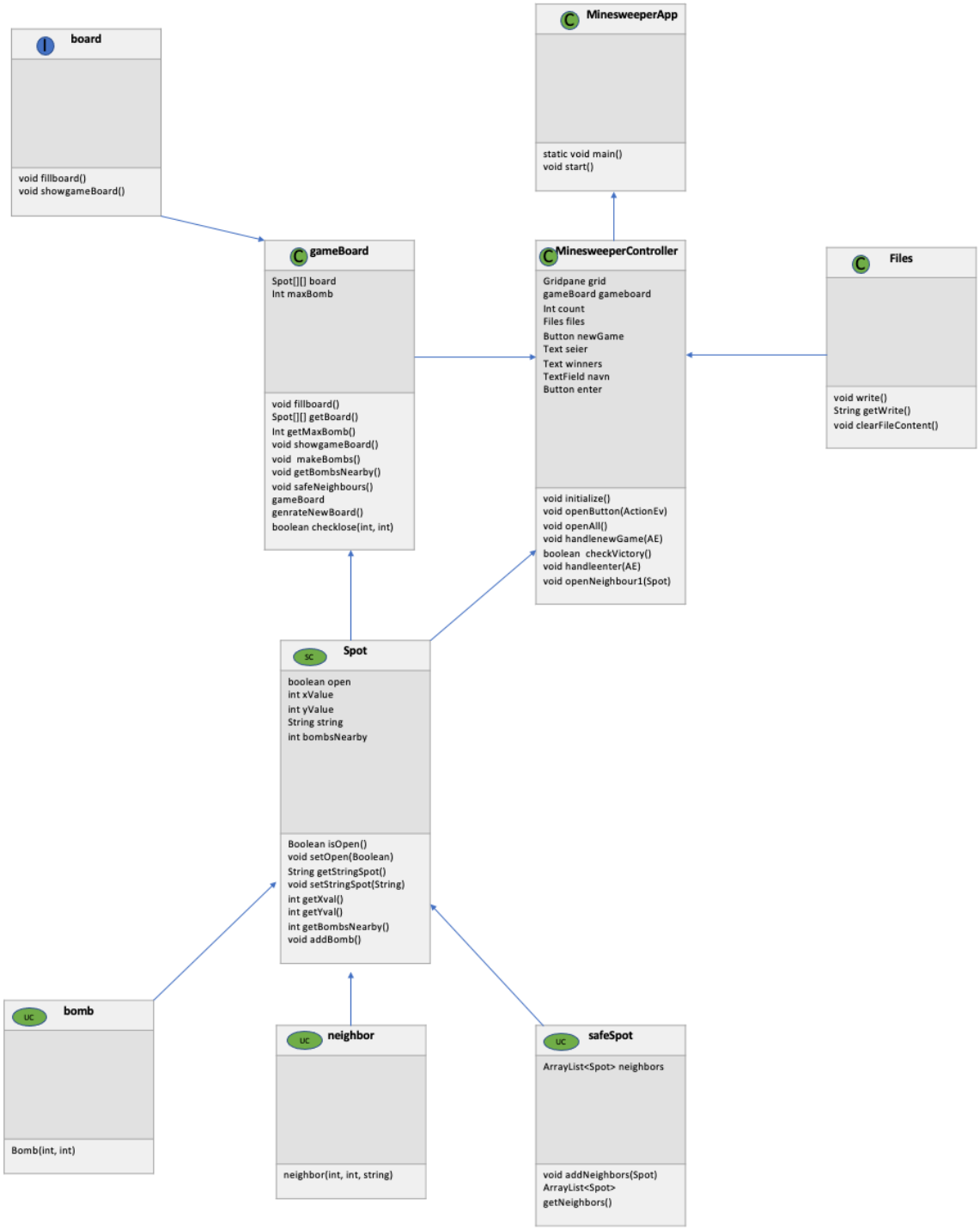
Minesweeper

Beskrivelse av appen

Minesweeper er et legendarisk spill kjent for de fleste. Målet med spillet er å navigere seg gjennom en brett med bomber, uten å treffe bombene. Man navigerer seg gjennom brettet ved å regne seg frem til hvilke ruter som er trygge, og hvilke som inneholder en bombe. Dette gjøres ved at hver rute som ikke er en bombe, tildeles et tall som sier hvor mange bomber det er i denne rutens omkrets. Åpner du en rute og det viser tallet 1, vet du altså at i de åtte rutene som omringer denne ruten, er det én rute som inneholder en bombe. Viser ruten tallet 2, er det to bomber i de åtte rutene osv. Er ruten blank betyr det at det ikke er noen bomber i nærheten. Da skal spillet åpne de rutene rundt som også er blanke, pluss de tallene som omringer de blanke rutene. Dette gjør at man kommer i gang med spillet, slik at det ikke blir totalt tilfeldig. I tillegg kan man starte et nytt spill med et helt nytt brett («New Game») og det er mulig å se hvem som har vunnet tidligere.






board
void fillboard()
void showgameBoard()
MinesweeperApp
static void main()
void start()
gameBoard
Spot[][] board
Int maxBomb
void fillboard()
Spot[][] getBoard()
Int getMaxBomb()
void showgameBoard()
void makeBombs()
void getBombsNearby()
void safeNeighbours()
gameBoard
genrateNewBoard()
boolean checklose(int, int)
MinesweeperController
Gridpane grid
gameBoard gameboard
Int count
Files files
Button newGame
Text seier
Text winners
TextField navn
Button enter
void initialize()
void openButton(ActionEv)
void openAll()
void handlenewGame(AE)
boolean checkVictory()
void handleenter(AE)
void openNeighbour1(Spot)
Files
void write()
String getWrite()
void clearFileContent()
Spot
boolean open
int xValue
int yValue
String string
int bombsNearby
Boolean isOpen()
void setOpen(Boolean)
String getStringSpot()
void setStringSpot(String)
int getXval()
int getYval()
int getBombsNearby()
void addBomb()
bomb
Bomb(int, int)
neighbor
neighbor(int, int, string)
safeSpot
ArrayList<Spot> neighbors
void addNeighbors(Spot)
ArrayList<Spot>
getNeighbors()

Spørsmål

1. Hvilke deler av pensum i emnet dekkes i prosjektet, og på hvilken måte? (For eksempel bruk av arv, interface, delegering osv.)

   Prosjektet dekker en god del pensum, både ved teknikker og selve programmeringen. Prosjektet benytter seg av flere klasser som interagerer med hverandre, samt Collections som «ArrayList». I klassene er riktig type innkapsling og synlighetsmodifikatorer brukt. I tillegg til «private» og «public», er det også brukt «protected» som brukes ved arv. «Protected» gjør at det er tilgjengelig for underklassene, men ellers fungerer det som «private». Arv har blitt brukt over de fire klassene «Spot», «Neighbor», «bomb» og «safeSpot». Her er det «Spot» som er superklassen, mens de tre andre klassene extender denne. Dette har blitt gjort for å kunne utvide underklassenes tilstander og oppførsel til en kombinasjon av superklassen og underklassene. Bruken har arv har vært svært fordelaktig fordi dette har gjort det enkelt å skille de ulike spottene på brettet. Å skille de ulike spottene har vært viktig for å kunne tildele riktig funksjon til riktig spot. Til dette er det brukt casting og «instanceOf», som er en del av pensum. Hvis spotten var av typen (instanceOf) «bomb» vil den da oppføre seg deretter. Hvis spotten var av typen (instanceOf) «safeSpot» vil den få tildelt disse naboene osv. Dette er brukt mye gjennom hele koden for å skille på spottene.

   I koden finner man eksempler på data behandlet med Stream. Dette ble brukt fordi det er en rask og enkel måte å iterere gjennom en mengde og filtrere ut det du ønsker. I denne sammenhengen ble det brukt til å telle antall spotter som var uåpnet, slik at man kunne si om spillet var over ved å sammenligne dette med antall bomber på brettet.

   Prosjektet inneholder både skriving og henting fra fil, samt noe Exceptions som «IOException» og «try & catch». Dette er hovedsakelig brukt ved skrivingen og hentingen fra fil. Ved bruk av dette kan man fange unntak som oppstår ved kjøring av koden, og bestemme hva som skal skje ved dette unntaket. I prosjektet er det brukt slik at man får vite hva som er feil. Ofte inntreffer «IOException» hvis Java ikke klarer å åpne en fil for eksempel. Dette dekker flere deler av pensum.

   Prosjektet tar i bruk JavaFX og FXML som er brukt til å bygge appen. Her har det blitt benyttet SceneBuilder, men også noe kode. Denne kombinasjonen sparte meg for mye tid. Da slapp jeg nemlig å tegne inn 81 StackPanes, 81 Textfields og 81 Buttons. Appen har flere Textfields og Buttons som brukes til å utføre funksjoner. Dette har blitt kontrollert med kontrolleren. Her binder jeg sammen selve spillet til appen. JavaFX og FXML er en del av pensum.

2. Dersom deler av pensum ikke er dekket i prosjektet deres, hvordan kunne dere brukt disse delene av pensum i appen?

   Det er deler av pensum som ikke dekkes i prosjektet. Sortering er for eksempel ikke brukt. Dette kunne blitt brukt til å sammenligne de ulike spottene og gi dem riktig

type spot, men jeg fant det enklere å benytte arv og casting. Hvordan jeg kunne benyttet flere standard funksjonelle grensesnitt er jeg usikker på.

3. Hvordan forholder koden deres seg til Model-View-Controller-prinsippet? (Merk: det er ikke nødvendig at koden er helt perfekt i forhold til Model-View-Controller standarder. Det er mulig (og bra) å reflektere rundt svakheter i egen kode)

   Koden forholder seg ikke perfekt til Model-View-Controller-prinsippet. Det har vært vanskelig å komme seg unna å ha noe logikk i kontrollene slik jeg har bygd opp appen. Selve koden bak brettet ligger i modellen. Det vil si brettets oppbygning, hvor bombene ligger, hvilke ruter som skal ha hvilke symbol og om spillet er tapt. Dette gjelder også alt av kode for spottene, samt filbehandlingen.

   I kontrolleren finner man koden som knytter modellen opp mot brukergrensesnitt. Ved kjøring av appen fylles brettet med StackPanes bestående av knapper og tekst. Teksten bak knappene knyttes så opp mot tilsvarende rute fra brettet i modellen. Dette er ganske effektivt og greit. Ellers så styrer kontrolleren funksjonene til de ulike knappene. Her blir noen av metodene litt mer komplekse siden man er avhengig av koordinatene til den knappen man trykker på for å knytte det opp mot brettet i modellen. Noe av «New Game» knappen sin logikk er nødt til å være i kontrolleren siden den resetter brettet i modellen, og knytter det så opp mot det visuelle.

   Grunnen til at noe av logikken ender i kontrolleren er at jeg benytter meg litt av de visuelle delene i spillet i logikken. Dette gjelder hovedsakelig to metoder: «openNeighbour» og «checkVictory». I openNeighbour iterer jeg gjennom det visuelle brettet, og fjerner så knapper visuelt. Derfor kommer logikken kontrolleren. Jeg må jobbe meg gjennom hver node på brettet, og så åpne hver knapp som skal åpnes. Det er selvfølgelig noe koblet opp mot modellen da metoden er avhengig av naboene til en bestemt spot. I «checkVictory» benytter jeg det visuelle til å sjekke om spilleren har seiret. Dette kunne nok vært gjort på en annen måte som kunne unngått kontrolleren, men for meg var denne metoden svært logisk.

4. Hvordan har dere gått frem når dere skulle teste appen deres, og hvorfor har dere valgt de testene dere har? Har dere testet alle deler av koden? Hvis ikke, hvordan har dere prioritert hvilke deler som testes og ikke? (Her er tanken at dere skal reflektere rundt egen bruk av tester)

   Testing av koden har vært noe utfordrende da det er svært få inputs fra brukeren. Derfor har testene blitt brukt til å sjekke om brettet blir satt opp slik det skal. Jeg har testet at riktig antall bomber legges til, og at det er av riktig. Dette er nødvendig for å sjekke at spillet faktisk er spillbart. Testing av skriving og henting til fil er en åpenbar test da det er helt avhengig at den skriver og henter riktig. Her har jeg sjekke at det som skrives inn, stemmer med det som skrives ut. I tillegg har jeg skrevet noen tester til spotene slik at de fungerer som de skal. Hvis ikke vil ikke logikken ved spillet fungere som det skal lenger.