Utvidelse: manipulering og GIF

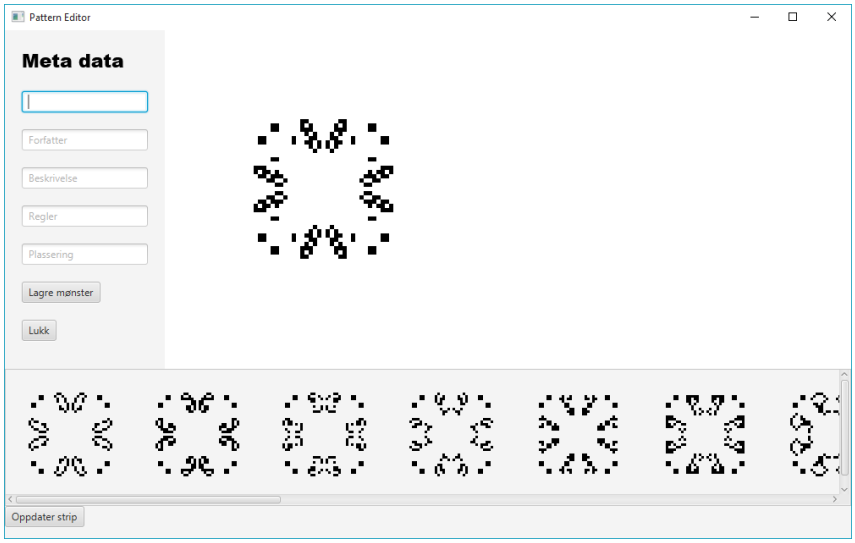
DATS1600 utvidelsesoppgave for semesteroppgaven.

Manipuleringsmuligheter for GoL mønstre

Dere skal i denne oppgaven implementere et vindu for manipulering og lagring av egen-definerte mønstre. Det antas at dere har allerede implementert lesing av meta data (fra sett 5). Dette vinduet skal støtte følgende:

- GUI elementer for å skrive inn meta data.
- Enkel manipulering av mønster ved hjelp av 'point-and-click' interface.
- Lagring av mønster til fil.
- En 'strip' som viser evolusjonen til mønsteret over et gitt antall generasjoner.
- Vinduet skal være et separat vindu fra hovedvinduet og skal kunne opprettes via hovedvinduet.

Eksempel på et vindu som støtter disse punktene er vist på neste side.



Eksempel på et manipuleringsvindu.

Opprettelse av nytt vindu fra hovedcontrolleren

Et vindu er representert via et objekt av JavaFX typen Stage. Dette objektet kan opprettes ved oppstart eller når brukeren velger å åpne editoren for første gang. Eksempel på opprettelse og initialisering av et slik objekt, samt visning av vinduet ved show metoden:

```
editor = new Stage();
FXMLLoader loader = new FXMLLoader(getClass().getResource("view/PatternEditor.fxml"));
GridPane root = loader.load();

Scene scene = new Scene(root, APP_WIDTH, APP_HEIGHT);

editor.setScene(scene);
editor.setTitle("Pattern Editor");
editor.show(); // showAndWait kan eventuelt brukes
```

Dette vinduet, relatert til PatternEditor.fxml, skal ha en egen controller klasse. Opprett denne i Java pakken der controlleren til hovedvinduet er lokalisert. Det kan være behjelpelig å kommunisere med

denne controlleren fra hovedvinduet. Mer spesifikt, burde det nåværende spillet (inkludert dens mønster) være tilgjengelig i editor controlleren. Det er to årsaker til dette: 1. det nåværende spillet i hovedvinduet må kunne manipuleres slik at brukeren ikke må tvinges til å 'starte med blanke ark'; 2. det endrede mønsteret burde vises i hovedvinduet etter at brukeren har lukket manipuleringsvinduet.

Diskuter synergien mellom hovedvinduet og manipuleringsvinduet. Jeg anbefaler at dere kopierer referansen til spillobjektet over til manipuleringscontrolleren. Dette gjør at ingen konflikter vil oppstå når manipuleringsvinduet lukkes (f.eks. overskrivning av det originale spillobjektet når det manipulerte spillet/mønsteret sendes tilbake). Hvis en slik referansekopi gjøres, må dere fortsatt være forsiktige med håndteringen av vinduene. For eksempel, burde spillet i hovedvinduet settes på 'pause' før manipuleringsvinduet opprettes. Hvis ikke, vil spillet kontinuerlig endres fra hovedvinduet (ved nextGeneration metoden), noe som ikke er ønskelig når mønsteret til spillet manipuleres av brukeren. I tillegg, endringer via hovedvinduet, som innlasting av nytt mønster, burde ikke tillates. Dette kan oppnås ved å fryse hovedvinduet mens manipuleringsvinduet er aktivt. Følgende kode angir slik oppførsel:

```
editor = new Stage();
editor.initModality(Modality.WINDOW_MODAL);
editor.initOwner(<an fxmlobject>.getScene().getWindow());
```

Der vi bruker et fxml objekt for å få tak i Scene objektet til hovedvinduet (dvs. en variabel deklarert med @FXML).

For å overføre data fra hovedvinduet, dvs. controlleren til hovedvinduet, til manipuleringsvinduet, så kan dette gjøres via loader objektet av typen FXMLLoader:

```
EditorController edController = loader.getController();
```

Data kan nå overføres via set-metoder i EditorController klassen.

Manipulering via point-and-click interface

Diskuter hvordan mønsteret skal kunne endres av brukeren. Den mest naturlige fremgangsmåten er ved et såkalt 'point-and-click interface', der brukeren klikker på en lokasjon i vinduet. Hvis brukeren klikker på en celle som er aktiv burde celles gjøres inaktiv, og omvendt.

Å opprette en slik funksjonalitet er relativt rett frem. Posisjonen til brukerens klikk kan hentes ved hjelp av get-metoder til MouseEvent klassen (getX() og getY() er relevante her). Husk at cellens posisjon ikke nødvendigvis er lik x og y posisjonen på skjermen (avhenger av implementasjon). Vanligvis, er cellens posisjon relatert til den underliggende tabellen (2D array). Hvis dette er tilfellet, vil (x,y) ikke være lik cellens 2D tabellindeks hvis cellen er større enn 1 pixel på skjermen. Enkel aritmetikk kan utføres for å konvertere et (x,y) koordinat til en 2D tabellindeks i dette tilfellet.

Lagring av mønster

Lagring av mønster vil følge en liknende prosess som lesing av mønster. Dvs: brukeren velger en fil som skal skrives til (enten en ny fil eller en eksisterende fil). En stream brukes deretter til å skrive en string til filen. Se følgende klasser for dokumentasjon:

http://docs.oracle.com/javase/8/docs/api/java/io/OutputStreamWriter.html

http://docs.oracle.com/javase/8/docs/api/java/io/FileWriter.html

Selv om skriving av data er en del enklere enn tolkning (parsing) av mønster, kan feil fort oppstå. Opprett JUnit testing for metoden relatert til skriving av fil. Implementer og test som tidligere. Husk bruk av exceptions og melding til brukeren hvis noe gikk galt.

Mønsteret som skrives til fil burde være så lite som mulig, der unødvendige døde celler ikke burde være med. Mer spesifikt, ønsker dere å flytte mønsteret til venstre hjørne av brettet slik at området mønsteret okkuperer er så lite som mulig. Opprett en metode for slik funksjonalitet (trim er et naturlig navn på metoden). Spillebrettet må kopieres til et midlertidig brett slik at dere ikke overskriver data. Metoden getBoundingBox (fra filbehandlingsoppgavene) kan brukes til å finne ut hvor langt cellene skal 'flyttes'.

The Strip

Det vil være naturlig for brukeren å vite hvordan det manipulerte mønsteret utvikler seg over tid. På grunn av 'the halting problem', er det umulig å vite om et virkårlig mønster dør ut eller om det overlever uendelig. Likevel, kan funksjonalitet som viser f.eks. de første 20 generasjonene til et mønster fortelle oss mye om dens videre utvikling.

Dere skal implementere en slik funksjonalitet ved å lage en 'strip' (inspirert av konseptet filmstrip). Skjermbildet av manipuleringsvinduet demonstrerer dette konseptet: evolusjonen til spillet blir tegnet repetitivt horisontalt møt høyre. Størrelsen til cellene som blir tegnet er relativt liten slik at flere brett kan vises over et mindre området.

Før grafikkdelen implementeres, burde dere tenke på hvordan selve spillet skal håndteres for å muliggjøre denne funksjonaliteten. Å utføre denne operasjonen direkte på spillet, ved hjelp av repetitive kall til nextGeneration metoden, vil ikke være ønskelig fordi du ikke vil endre på det originale spillet. Dvs. at du ikke vil at mønsteret hopper 20 generasjoner fremover i tid hver gang strip'en oppdateres.

For å unngå en slik oppførsel burde spill-objektet kopieres til et nytt, separat, spillobjekt. Dvs. når det kopierte spillobjektet endres, ved nextGeneration, skal dette ikke påvirke det originale objektet. En slik kopieringsoperasjon kalles 'deep copy'; til sammenligning kalles kopiering ved referanse-variabler for 'shallow copy'. Opprett og implementer en slik deep copy kopieringsmetode. Overkjøring (override) av Object sin clone metode kan være et naturlig valg her. Husk JUnit testing, der dere tester om det kopierte objektet faktisk er lik det originale objektet og om endring av det kopierte objektet, ved nextGeneration, ikke påvirker det originale objektet.

Mønsteret til spillet kan være lokalisert lengre ute på brettet, dvs. ikke på kanten. Dette er et problem fordi området mønstrene tegnes på er begrenset. Bruk trim metoden dere utviklet tidligere for å flytte mønsteret til venstre hjørne av brettet for å løse dette problemet.

Dataene fra getBoundingBox kan brukes til å finne ut hvor stort mønsteret er for å definere størrelsen til hvert element/brett i strip'en. Dere er spesielt opptatt av bredden til hvert brett som tegnes i strip'en. Når dere tegner hvert brett kan dere bruke GraphicsContext sin funksjonalitet for affine transformasjoner, spesielt translation, for å flytte 'skrivehode' til høyre. I Java-stil pseudo kode kan dette gjøres slik:

```
GraphicsContext gc = strip.getGraphicsContext2D();
gc.clearRect(0, 0, strip.widthProperty().doubleValue(),
                   strip.heightProperty().doubleValue());
Affine xform = new Affine();
double tx = xpadding;
< start strip loop>
```

```
xform.setTx(tx);
gc.setTransform(xform);
<next generation call>
<draw game board call>
tx += <width of game board + xpadding>
<end strip loop>

// reset transform
xform.setTx(0.0);
gc.setTransform(xform);
```

Lagring til GIF

Relevant forelesning angående rekursive funksjoner holdes i uke 10.

GIF (Graphics Interchange Format) er et bildeformat som støtter animasjoner. Bildeformatet er spesielt populært i bruk for weben på grunn av god støtte av nettlesere, lite bruk av harddisk-plass, og støtte for animasjon. En ulempe med GIF er at hvert bilde kan bare bruke maksimalt 256 farger. Denne ulempen vil imidlertid ikke være et praktisk problem for denne oppgaven. For mer info ang. GIF, se: https://en.wikipedia.org/wiki/GIF

Det har blitt utviklet et GIF bibliotek som dere skal bruke i denne oppgaven. Mer spesifikt, skal dere bruke et objekt av typen lieng.GIFWriter til å opprette en fil-stream, skrive bilder rekursivt til denne streamen, og lukke streamen (dvs. lagre til GIF). Se GIFLib.zip for Javadoc dokumentasjon, et enkelt eksempel, og selve Java biblioteket pakket inn i en .jar fil.

Java jar filer brukes til å pakke flere .class filer inn i en fil. Java *biblioteker* pakkes og distribueres på denne måten. Hvis man ønsker å publisere klasser kompilert som Java bytecode, der den originale Java koden ikke er tilgjengelig, er bruk av .jar også vanlig. Man kan da velge å inkludere den originale Java koden eller ikke.

Plasser GIFLib.jar filen i en mappe i prosjektet deres. Finn ut hvordan dere inkluderer biblioteker til 'build path'en' til prosjektet deres i IDE'en dere bruker. Hvis dere bruker Eclipse, høyre-klikk på prosjektet og velg build path -> add external archives.

Dere skal lage funksjonalitet som tilbyr skriving av evolusjonen til det nåværende spillet på brettet til en GIF bildefil. Denne oppgaven kan sammenlignes med forrige oppgave, der evolusjonen til det manipulerte spillebrettet vises i en GUI 'strip'. I denne oppgaven, skrives stripen som en sekvens av bilder, som vil produsere en animasjon.

Dere trenger å kopiere (deep copy) deres nåværende spill til et midlertidig spill-objekt. Da kan dere utføre kall til nextGeneration (på det midlertidige objektet) slik at spillet på GUI brettet ikke endres når spillet skrives til GIF.

I denne oppgaven, skal dere bruke rekursjon til å skrive sekvensen av bilder til GIF bilde-streamen, via et GIFWriter objekt. Diskuter i gruppen fordelene og ulempene med rekursjon for denne oppgaven.

Metoden som utfører denne GIF oppgaven skal være skrevet etter følgende Java-stil pseudo kode:

```
void writeGoLSequenceToGIF(lieng.GIFWriter writer, GameOfLife game,
                           int counter, …)
{
      <condition to end recursion>
      <draw game board to current image in writer>
      <insert image to GIF sequence and proceed to next image, via writer>
      <nextGeneration call to game>
      <recursive call to writeGoLSequenceToGIF>
}
```

Forstå hva halerekursjon (eng: tail recursion) er og fordelen er med slik rekursjon. Test om metoden dere har implementert over utfører slik halerekursjon. Ut ifra denne testen, diskuter nå fordeler/ulemper med rekursjon for dette problemet. Til slutt, bruk Internett til å finne ut om Java/JVM støtter halerekursjon og eventuelt hvordan.

Diskuter håndteringen av de frie variablene relatert til denne oppgaven. Disse variablene er: størrelse til GIF bildet (i piksler), tid mellom hvert bilde i animasjonen (i millisekunder), antall iterasjoner som skal skrives til GIF bildet, størrelsen til hver celle på brettet (i piksler), og grafikken til hver celle (rektangler eller noe mer 'fancy'). Det er naturlig å la brukeren manipulere flere av disse variablene.

Hvis et mønster itererer uendelig (evig løkke), er det naturlig å iterere like mange ganger i animasjonen som i løkken. Da vil animasjonen i GIF bildet iterere *kontinuerlig*, uten hopp. Automatisk oppdagelse av løkker kan oppdages ved å se på statistikken til mønsteret, hvis denne informasjonen ikke er en del av meta data'en til mønsteret. Denne oppgaven kan dermed revideres hvis oppgaven for statistikk utføres.