Laboration 5: Game of Life

Bakgrund:

I denna laboration ska vi ägna oss åt Game of Life, vilket är en mycket enkel modell av en population organismer som kan leva, dö och fortplanta sig i en tvådimensionell värld. Trots sin enkelhet kan simuleringar av denna population uppvisa komplicerade beteenden och modellen har därför blivit mycket välkänd.

I figuren ovan visas gränssnittet för ett grafiskt program som simulerar Game of Life. Huvuddelen av fönstret upptas av den tvådimensionella världen, som består av ett rutnät av *celler* (i detta fall 50x50 celler). En cell kan vara *levande* (svart) eller *död* (vit). I ett *steg* kan hela populationen momentant övergå till en ny *generation*. Reglerna för detta är följande. För varje cell (både levande och döda) räknar man efter hur många levande grannar cellen har, där grannar är de åtta omgivande cellerna (celler längs kanterna har bara fem grannar och de fyra hörncellerna har bara tre grannar). En cell som är levande i generation *n* förblir levande i generation *n+1* om den har två eller tre levande grannar. En cell som är död i generation *n* blir levande i generation *n+1* om den har tre levande grannar. I alla övriga fall är cellen död i generation *n+1*.

Gränssnittet ovan innehåller, förutom världen, tre *knappar* och en *etikett*. Step-knappen gör ett steg, dvs räknar ut och visar nästa generation. Run-knappen sätter igång en utveckling av den ena generationen efter den andra, något som kan stoppas med Stop-knappen. På etiketten skrivs ut hur många nya generationer av organismerna som skapats sedan simuleringen startades. Vad som inte syns i figuren ovan är att man genom att klicka med musen på en cell kan ändra cellen från levande till död eller tvärtom.

Programmet kan avbrytas genom att klicka på stängningsrutan i högst upp till höger i fönstrets ram.

Ingående klasser

I det här laboration skall ni utveckla ett grafiskt program för Game of Life. Ni får givet alla de delar av programmet som har med grafiken att göra och ska enbart implementera den bakomliggande *modellen*.

Ladda ner filen gameOfLife.zip från kursens hemsida. Denna fil innehåller källkoden för fyra klasser:

- LifeModel.java. Detta är den klass som utgör modellen, dvs klassen som ska ha kunskap om reglerna för hur en population utvecklas. Men dessa regler är inte implementerade i den givna klassen. Istället består de flesta metoderna enbart av ett skelett, en sk *stubbe*, som endast skriver ut att metoden blivit anropad. Vi ger er denna klass för att ni ska kunna följa dessa utskrifter och därmed förstå hur de grafiska klasserna och modellen samverkar. Er uppgift är sedan att skriva om klassen så att ni får ett fullständigt program för Game of Life.
- LifeWorld.java. Detta är den klass som ritar upp rutnätet av celler.
- LifeView.java. Denna klass handhar hela den grafiska vyn som visas i fönstret, dvs en instans av LifeWorld (som visar rutnätet), etiketten som visar hur många generationer av populationen som skapats, samt de tre knapparna.
- Main.java. Huvudklassen som skapar ett modellobjekt (en instans av `LifeModel`), en instans av LifeWorld samt en instans av `LifeView` och placerar den senare i ett fönster på skärmen (i en `JFrame`).

Klassen Main

```
import javax.swing.*;
public class Main {
    public static void main(String[] args) {
        JFrame f = new JFrame("Game of Life");
        LifeModel model = new LifeModel(50, 50);
        LifeWorld world = new LifeWorld(model, 400);
        LifeView view = new LifeView(model, world);
        if (args.length != 1) {
            System.out.println("Felaktigt antal argument!");
        }
        else {
            try {
                double ratio = Double.parseDouble(args[0]);
                if (ratio < 0 || ratio > 1.0) {
                    System.out.println("Argumentet måste vara ett tal i intervallet[0,1].");
                }
                else {
                    model.randomPopulation(ratio);
                    f.add(view);
                    f.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                    f.pack();
                    f.setResizable(false);
                    f.setVisible(true);
                }
            } catch (NumberFormatException nf) {
                System.out.println("Argumentet måste vara ett reellt tal i intervallet[0,1].");
            }
        }
    } //main
} //Main
```

Klassen Main innehåller endast en main-metod, i vilken de olika komponenterna i vårt program skapas och kopplas samman. Först skapas fyra objekt:

- ett fönsterobjekt f, som är en instans av standardklassen JFrame.
- ett modelobjekt model, som är en instans av klassen LifeModel. Konstruktorn för klassen LifeModel har två parametrar som anger hur stor vår tvådimensionella värld skall vara i bredd respektive höjd (i detta fall skaper vi en värld som är 50 x 50 celler).
- ett objekt world, som är en instans av klassen LifeWorld. Konstruktorn för klassen LifeWorld har två parametrar – dels modelobjektet model, dels en storlek som anger hur stor världen skall vara (i antalet pixlar) när den ritas ut i fönstret.
- ett vyobjekt view, som är ett objekt av klassen LifeView. Konstruktorn för klassen LifeView har två parametrar – dels modellobjektet model, dels instansen world av klassen LifeWorld.

Man ger indata till main-metoden via kommandoraden. Denna indata skall utgöras av ett reellt tal i intervallet [0, 1] och anger hur stor andel av cellerna i världen som skall vara levande. Om felaktigt antal argument ges på kommandoraden eller om argumentet inte är ett reellt tal i intervallet [0, 1] ges felutskrifter och programmet avbryts. Annars beordras modellobjekt (genom ett anrop av metoden randomPopulation) att skapa en slumpmässig startpopulation med den sannolikhets fördelning på levande och döda celler som angetts på kommandoraden. Slutligen görs 5 operationer på fönstret:

- vyobjektet view placeras i fönstret (med metoden add)
- programmet görs möjligt att kunna avslutas via avstängningsrutan i fönstrets ram (med metoden setDefaultCloseOperation)
- komponenterna i fönstret packas, dvs fönstret görs precis så stort att det rymmer vyn (med metoden pack)
- fönstret förhindras att kunna ändra sin storlek (med metoden setResizable)
- fönstret görs synlig på skärmen (med metoden setVisible)

Därefter är `main` klar.

Klassen LifeWorld

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
public class LifeWorld extends JPanel implements MouseListener {
    private LifeModel model;
    private int xside, yside;

    public LifeWorld(LifeModel model, int size) {
        this.model = model;
        xside = size/model.getWidth();
        yside = size/model.getHeight();
        setPreferredSize(new Dimension(size, size));
        setBackground(Color.WHITE);
        setOpaque(true);
        addMouseListener(this);
    } //constructor

    public void paintComponent(Graphics g) {
        super.paintComponent(g);
        g.setColor(Color.BLACK);
        int width = model.getWidth();
        int height = model.getHeight();
        for (int i = 0; i < width; i = i + 1) {
            for (int j = 0; j < height; j = j + 1) {
                if (model.getCell(i, j)) {
                    g.fillRect(i * xside, j * yside, xside, yside);
                }
            }
        }
    }//paintComponent

    public void mouseClicked(MouseEvent e) {
        int i = e.getX() / xside;
        int j = e.getY() / yside;
        model.invertCell(i, j);
        repaint();
    }//mouseClicked

    public void mouseEntered(MouseEvent e) {  }

    public void mouseExited(MouseEvent e) {  }

    public void mousePressed(MouseEvent e) {  }

    public void mouseReleased(MouseEvent e) {  }
}//LifeWorld
```

Klassen LifeWorld är den grafiska komponenten som visar den tvådimensionella världen av celler. Till konstruktorn ges två parametrar – dels modellen, dels komponentens storlek i antal pixels (den grafiska representationen av världen får alltså storleken size pixels både på bredden och höjden). Klassen LifeWorld har ingen kännedom om varken hur många celler som finns i världen eller huruvida en cell är död eller levande, denna kunskap finns i klassen LifeModel. En instans av klassen LifeWorld måste därför *känna till* och samarbeta med en instans av klassen LifeModel för att kunna utföra sin uppgift.

Storleken, xside respektive yside, på cellerna som skall ritas ut beräknas genom att fråga modellobjekt model efter antalet celler i x- respektive y-led (via metoderna getWidth samt getHeight) och dividera LifeWorld-komponentens storlek size med dessa värden.

Vi vill via LifeWorld-komponenten kunna klicka med musen på en cell och ändra tillståndet på cellen från död till levande och vice versa. När vi trycker på musen genereras en händelser av typen MouseEvent. För att ta hand om dessa händelser måste klassen implementera interfacet MouseListener. Detta interface specificerar 5 metoder - mouseClicked, mouseEntered, mouseExited, mousePressed och mouseReleased - för att kunna särskilja mellan olika typer av händelser som kan genereras vis musen (t.ex klicka, röra eller dra och släppa). Här är vi enbart intresserade av att klicka med musen, därför implementerar vi endast metoden mouseClicked och låter de andra metoderna göra ingenting (dock måste de finnas i klassen, eftersom klassen annars inte skulle implementera interfacet MouseListener på ett korrekt sätt).

Metoden MouseClicked börjar med att ta reda på i vilken pixel i den grafiska världen som musen befanns sig när den klickades. Detta görs med mha metoderna e.getX och e.getY. Sedan beräknas vilken cell denna position motsvarar, genom att positionen divideras med cellernas pixelstorlek. Därefter ombeds modellobjektet, via ett anrop av metoden inversCell, att ändra tillståndet på denna cell från död till levande eller vice versa. Eftersom en cell har bytt tillstånd, ritas slutligen världen om genom att anropa repaint.

Ritning av en grafisk komponent sker med metoden paintComponent. I klassen LifeWorld skall alltså paintComponent rita den tvådimensionella världen av celler. Antalet celler i världen, på bredden respektive höjden, efterfrågas från modellobjektet via metoderna getWdith och getHeight. Alla celler genomlöps och om en cell är levande (fås reda på genom att anropa modellen med metoden getCell) ritas cellen ut med en svart rektangel (som har storleken xside x yside).

Klassen LifeView

```
import javax.swing.*;
import java.awt.*;
import java.awt.event.*;
public class LifeView extends JPanel implements ActionListener {
    private LifeModel model;
    private LifeWorld world;
    private JButton stepButton = new JButton("Step");
    private JButton runButton = new JButton("Run");
    private JButton stopButton = new JButton("Stop");
    private JLabel generationLabel = new JLabel();
    private Timer timer = new Timer(200, this);

    public LifeView(LifeModel model, LifeWorld world) {
        this.model = model;
        this.world = world;
        JPanel buttonPanel = new JPanel();
        buttonPanel.setLayout(new GridLayout(1, 3));
        buttonPanel.add(stepButton);
        buttonPanel.add(runButton);
        buttonPanel.add(stopButton);
        generationLabel.setText("Generation: " + model.getNrOfGeneration());
        JPanel southPanel = new JPanel();
        southPanel.setLayout(new GridLayout(2,1));
        southPanel.add(generationLabel);
        southPanel.add(buttonPanel);
        setLayout(new BorderLayout());
        add(world, BorderLayout.NORTH);
        add(southPanel, BorderLayout.SOUTH);
        stepButton.addActionListener(this);
        runButton.addActionListener(this);
        stopButton.addActionListener(this);
    } //constructor

    public void actionPerformed(ActionEvent e) {
        if (e.getSource() == stepButton || e.getSource() == timer) {
            model.newGeneration();
            world.repaint();
            generationLabel.setText("Generation: " + model.getNrOfGeneration());
        }
        else if (e.getSource() == runButton) {
            timer.start();
        }
        else if (e.getSource() == stopButton) {
            timer.stop();
        }
    }//actionPerformed
}//LifeView
```

I klassen LifeView sätts alla grafiska komponenter ihop till en gemensam vy. Vyn består av rutnätet av celler, vilket är en instans av klassen LifeWorld, etiketten som visar hur många generationer av populationen som skapats, samt de tre knapparna Step, Run och Stop. Instansen av klassen LifeWorld ges som parameter till konstruktorn (instansen skapas i klassen Main). Konstruktorn har även en parameter av klassen LifeModel (som också skapas i klassen Main).

För att styra hur ofta en ny generation av celler skall skapas har klassen en instansvariabel, timer, av klassen Timer. Timern initieras till att generera en ny händelse varannan sekund.

I metoden actionPerformed fångas de händelser som genereras både från timer-objektet och från knapparna.

- Om Step-knappen eller timer-objektet genererat en händelse skapar modellen först en ny generation celler (görs med metoden newGeneration). Den nya generationen ritas därefter ut i världen (genom att anropa metoden repaint). Slutligen uppdateras utskriften på etiketten (eftersom det nu är en ny generation av celler som visas i världen). Vilken den aktuella generationen är erhålls från modellobjektet via metoden getNrOfGeneration.
- Om Run-knappen genererar en händelse startas Timer-objektet (och detta kommer därmed att generera nya händelser varannan sekund).

- Om Stop-knappen genererar en händelse stoppas Timer-objktet (och kommer därmed att inte generera nya händelser innan det startas igen).

Klassen LifeModel

```
import java.util.Random;
public class LifeModel  {
    private int width, height, nrOfGeneration;
    private boolean[][] world;

    public LifeModel (int width, int height) {
        this.width = width;
        this.height = height;
    }//konstruktor

    public int getWidth() {
        System.out.println("LifeModel: Call to getWidth()");
        return width; //metoden kräver att ett heltal skall returneras
    }//getWidth

    public int getHeight() {
        System.out.println("LifeModel: Call to getHeight()");
        return height; //metoden kräver att ett heltal skall returneras
    }//getHeight

    public int getNrOfGeneration() {
        System.out.println("LifeModel: Call to nrOfGeneration()");
        return nrOfGeneration; //metoden kräver att ett heltal skall returneras
    }//getNrOfGeneration

    public void randomPopulation(double fill) {
        System.out.println("LifeModel: Call to randomPopulation(" + fill + ")");
    }// randomPopulation

    public void newGeneration() {
        System.out.println("LifeModel: Call to newGeneration()");
    }//newGeneration

    public void invertCell(int i, int j) {
        System.out.println("LifeModel: Call to invertCell(" + i + "," + j + ")");
    }//invertCell

    public boolean getCell(int i, int j) {
        if (i == 0 && j == 0)
            System.out.println("LifeModel: Call to getCell(" + i + "," + j + ")");
        return (i + j) % 2 == 0;
    }//getCell
} // LifeModel
```

Ni får ovanstående skelett till klassen LifeModel och er uppgift är att komplettera klassen så att det önskade beteendet som beskrevs inledningsvis erhålls.

Ett objekt av klassen LifeModel har kunskap om *en* population celler i Game of Life. Objektet har en uppsättning metoder som möjliggör att omvärlden (i vårt fall vyn) kan kommunicera med objekt. Via metoderna kan omvärlden ställa frågor om populationen och förändra populationen. Mellan metodanropen väntar modellobjektet passivt på nästa metodanrop.

Klassens konstruktor har två heltalsparametrar vilka definierar populationens storleken. Förutom konstruktorn, har klassen sju metoder:

public int getWidth()
public int getHeight()
public int getNrOfGeneration()
public boolean getCell(**int** i, **int** j)
public void invertCell(**int** i, **int** j)
public void randomPopulation(**double** fill)
public void newGeneration()

De tre första metoderna är redan implementerade i den klass ni fått (utskriftssatserna tar ni bort i slutversionen av klassen).

Metoden getWidth returnerar antalet kolumner i rutnätet av celler (denna uppgift lagras i instansvariabeln width). Metoden getHeight returnerar antalet rader i rutnätet av celler (denna uppgift lagras i instansvariabeln height). Metoden getNrOfGeneration returnerar hur många nya generationer av celler som skapas sedan den första generationen (denna uppgift lagras i instansvariabeln nrOfGeneration). Startgenerationen skall alltså ha generationsnummer 0.

Uppgift 1

Implementera metoderna getCell, invertCell, randomPopulation och newGeneration i klassen LifeModel.

För att hantera populationen av celler och vilket tillstånd de olika cellerna har, har klassen instansvariabeln world, som är ett tvådimensionellt fält av boolska värden. I detta fält markeras en levande cell med värdet **true** och en död cell med värdet **false**.

Metoden getCell(**int** i, **int** j) skall returnera värdet för cellen i position (i, j). I den givna klassen returneras ett värde som inte alls har med Game of Life att göra, nämligen värdet (i + j) %2 == 0. Detta leder till att vi får en värld där de levande cellerna bilder ett schackmönster. För att få en korrekt implementation måste värdet istället hämtas från instansvariabeln world.

Metoden invertCell(i,j) skall ändra cellen i position (i,j) till levande om cellen är död och till död om cellen är levande.

Metoden randomPopulation(**double** fill) skall skapa en ny slumpmässig population celler, där fill anger sannolikheten för att en cell är levande. Körs programmet med argument 0.0 skall alla celler vara döda (vilket ger en helt vit värld), om programmet körs med argumentet 1.0 skall alla celler vara levande (vilket ger en helt svart värld) och om argumentet 0.5 ges till programmet skall hälften av cellerna vara levande och hälften av cellerna vara döda. Använd en instans av klassen Random för att slumpa fram om en cell skall sättas som levande eller död. Titta på specifikationen av klassen Random i Java's API och och speciellt på metoden nextDouble.

Metoden newGeneration() skall byta ut den nuvarande generationen celler mot nästa generation enligt reglerna för Game of Life. Dessa regler är enligt följande:

> För varje cell (både levande och döda) räknar man efter hur många levande grannar den har, där grannar är de åtta omgivande cellerna. En cell som är levande i generation *n* förblir levande i generation *n+1* om den har två eller tre levande grannar. En cell som är död i generation *n* blir levande i generation *n+1* om den har tre levande grannar. I alla övriga fall är cellen död i generation *n+1*.

Det är lämpligt att införa en hjälpmetod

private int nrOfNeighbours(**int** i, **int** j)

som räknar ut antalet levande grannar till cellen (i, j). Eftersom celler längs kanterna bara har fem grannar och de fyra hörncellerna bara har tre grannar, utgör dessa celler *specialfall*. Men om ni använder metoden getCell(m, n) för att undersöka om granncellen är levande, istället för att direkt titta efter i fältet world, kan ni undvika dessa specialfall genom att utforma metoden getCell(m, n) på så sätt att metoden returnerar värdet false om positionen (m, n) inte finns i världen.

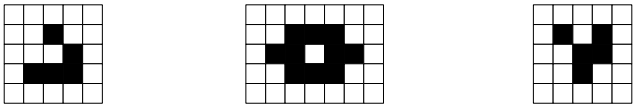
Uppgift 2

a) Inför ytterligare en knapp, Clear, i klassen LifeView. När man trycker på knappen skall en värld med enbart döda celler skapas. Detta kan vara praktiskt om man vill skapa förutbestämda startpopulationer mha musen.

b) Sätt färger och fonter på knapparna och etiketten.

c) Alla knappar är alltid tryckbara. Till exempel kan man trycka på Stop-knappen för att stoppa timern även om timern är stoppad. Gör de ändringar som behövs för att endast relevanta knappar är tryckbara i de olika tänkbara faser som programmet kan befinna sig i. När programmet startas skall t.ex. Stop-knappen inte vara tryckbar.

d) Testa nedanstående startpopulationer.



Det finns mycket information att hitta om Game of Life på nätet. Enkla startpopulationer som ger intressanta generationsutvecklingar finns bl.a på http://en.wikipedia.org/wiki/Conway's_Game_of_Life.

Redovisning

Laborationen skall dels demonstreras och godkännas av en handledare, dels redovisas skriftligt enligt de direktiv som finns på kursens hemsida.

Sista redovisningsdag och sista inlämningsdag för dokumentationen är fredag 10/3.