プログラミングD javaによるlifegame実装

2026年1月5日

1　lifegameの説明

初めに縦横に並んだ2次元のマス目が表示される。次にプレイヤーはドラッグ操作により白色のマス目を黒色に塗りつぶす。準備が出来たら、next ボタンを押してゲームを開始する。するとルールに基づいて塗りつぶされる場所が変化し、プレイヤーはその挙動を眺めるものである。ルールについては next ボタンの章で述べている。

2　作成したプログラムの操作方法と機能

設計の自由度がある箇所に対し作成したプログラムが持つ機能について解説する。

2.1　盤面の大きさ

初めは盤面の大きさは x 座標、y 座標共に 0 12,0 12、つまり $13 * 13$ の大きさで設計した。その後拡張機能の実装により、x 座標とｙ座標の個数が同じという条件下で自由にマスの数の大きさを編集できるようにした。また、ウィンドウを全画面にしたり小さくしたりすることに対応して、マス目の大きさも見やすいように小さくなったり大きくなったりと変化するようにした。これにより、画面の中でセルやボタンが見切れるという現象を防ぐことに成功した。また今回の要件では、ウィンドウの大きさが変化したら大きさに適応するように可変し、かつ正方形である必要がある。従ってウィンドウの横と縦で長い方の長さを 20 等分し、1 等分目から 13 等分目までのところにマス目を作り、生きているマス目に黒色を塗る、という仕様にした。以下に実際に表示される盤面を掲載する。

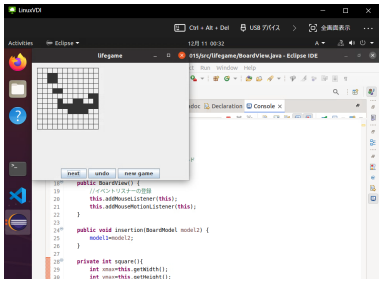

図 1: 小さい時

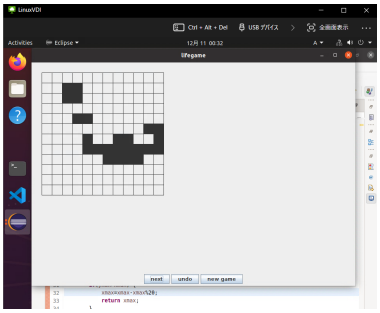

図 2: 大きい時

2.2　ドラッグ操作の巻き戻しの扱い

ドラッグ操作により複数のマスを塗られた際、undo ボタンを 1 回押すごとに 1 マスずつ元に戻す仕様になっている。32 マス以上塗ったり戻したりが行われた際には、初めのデータが失われるので、初期状態に戻すことは不可能になっている。但し next ボタンを押された後からは undo ボタンを押すと next ボタンを押す前の状態に戻るので、要件は満たしている。

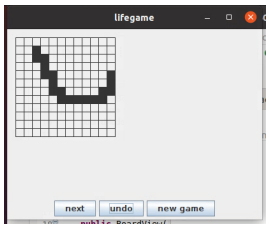

図 3: 巻き戻し前

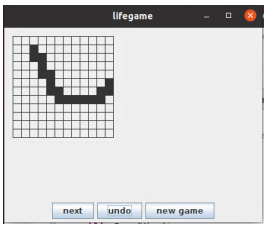

図 4: 巻き戻し後

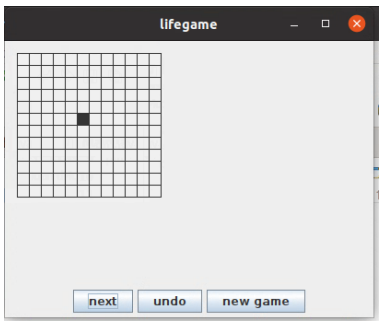

図 5: 巻き戻し前

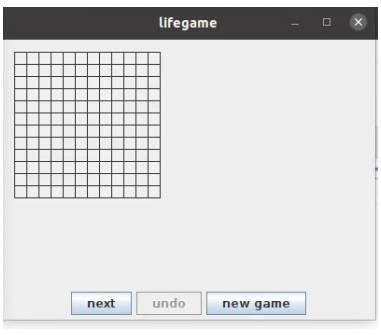

図 6: 巻き戻し後

2.3　next ボタン

プログラムを実行すると表示される別タブ内に next ボタンがあり、このボタンを押すと条件に基づいて次の世代の盤面を生成して表示する。next 機能の要件にある生存条件、誕生条件、死亡条件は表のようになっている。左の

表 1: lifegame のルール

条件	周囲 8 セルにある生存セルの数
生存	2 つか 3 つ
死亡	1 つ以下 (孤立)、4 つ以上 (過密)
誕生	3 つ

写真が next 動作実行前、右の写真が next 動作実行後である。

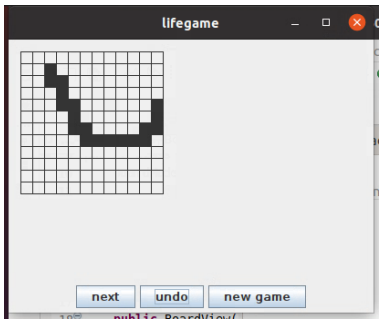

図 7: 実行前

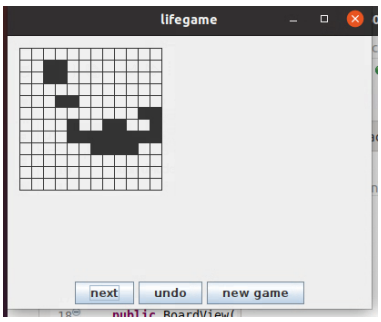

図 8: 実行後

また、盤面に何も記入しないまま next ボタンを押した際は、空の盤面がデータとして記録され、undo ボタンは有効化される。

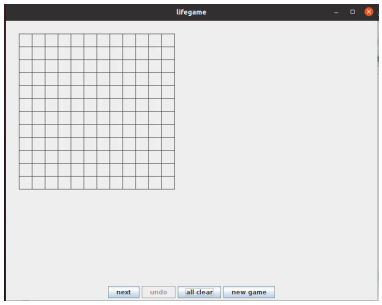

図 9: 初期画面

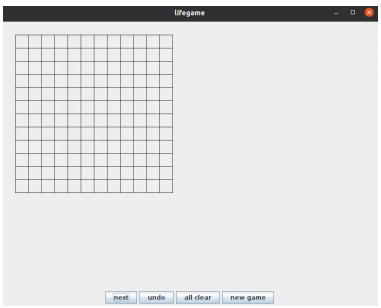

図 10: 空の盤面で next ボタン実行時

2.4　undo ボタン

2.4.1　巻き戻し動作の説明

プログラムを実行すると表示される別タブ内に next ボタンがあり、このボタンを押すと一つ前の世代に戻せる。next ボタンが押されている条件下では、next ボタンの回数分元通りの盤面に戻すことができる。next ボタンが押されていない条件下では、盤面が初期状態 (全てのマスが白色の状態) 以外であれば有効であり、最後に変化した 1 マスを元の状態に戻すことができる。以下の 3 枚の写真は盤面を塗った後、一度だけ next ボタンを押した状態から、2 回 undo ボタンを押した状況である。2 つめの写真が元の盤面編集完了時の盤面の画像となっている。1 度目の変化では世代が前に戻り、2 度目の変化では 1 マス分だけ塗ったマスが元に戻る。

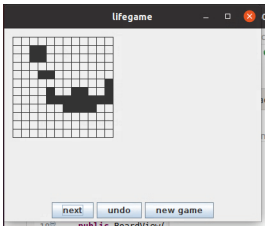

図 11: next ボタンを 1 回だけ押した状態

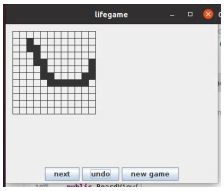

図 12: undo ボタンを 1 回だけ押した状態

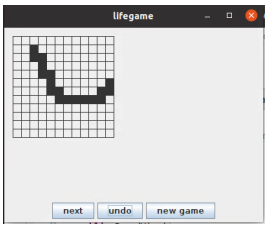

図 13: undo ボタンを 2 回押した状態

2.4.2　ボタンの有効無効の切り替え (特定状況下のみの利用)

ボタンは初期状態では無効になっている。盤面上でマウスクリックやドラッグをすることで盤面が変わった瞬間、undo ボタンは有効化する。初期状態の時と 32 世代以上戻したときや、32 マス以上盤面を塗って 32 回戻した際は、undo ボタンが無効になる。(データが保存されていないため。) 例として画像のような状況が挙げられる。

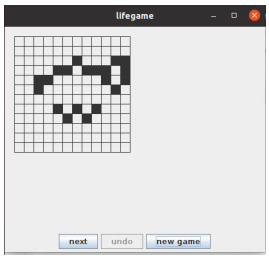

図 14: 32 世代戻って無効化

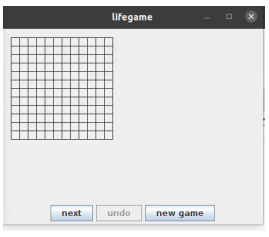

図 15: 初期状態で無効化

2.4.3　拡張機能利用時の undo 機能の動作

拡張機能である all clear ボタンを押した際には、undo ボタンは有効化されたままになる。直後に undo ボタンを押した場合は、クリアする以前の盤面が表示されるようになっている。

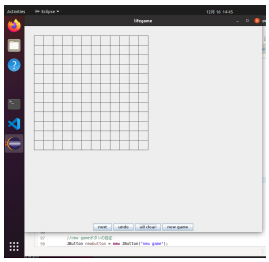

図 16: クリア機能利用直後の盤面

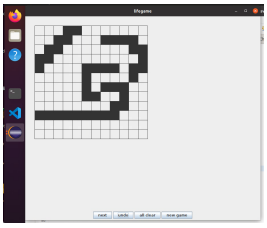

図 17: undo 機能により復元した状態の盤面

2.5　new game ボタン

new game ボタンを押すと新しい別タブが表示され、真っ白な初期状態が表示される。この画像では説明のため new game ボタンにより新規に生成された右のタブと盤面が塗られている既存の左のタブを並べて表示しているが、実際は既存のタブに重なるように表示される。

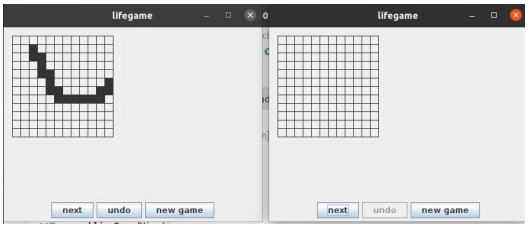

図 18: new game ボタンの動作

3　クラスと機能実現の方針

3.1　クラスと機能の対応表

表 2: クラスと機能の対応表

クラス名	機能
Main	別タブの生成、押されたボタン通りの機能につなげるクラス
BoardModel	next,undo 等の機能を内部で処理するクラス
BoardView	生成した別タブに盤面を表示する機能を持つクラス
BoardListener	undated メソッドを定義するインターフェース
ChangeUndoButton	next を一回も押さずに盤面を塗った時に undo 無効化を解除する機能を持つクラス

3.2　役割分担の方針

初めは undo は next の反対の機能を持つ、という間違ってはいないが不十分な認識をしていたので、ChangeUndoButton クラスを除いた 4 つのクラスによりプログラムを作成していた。主要な機能を果たすクラスとして以下の 3 つのクラスで設計構想を練った。Main クラスはボタンを押された際に内部処理を呼び出す動作、BoardModel クラスは next,undo 動作を行い、都度正しい真理値を受け渡す動作、BoardView クラスは盤面を塗るマウス操作があったり、BoardModel クラスから受け取る真理値が変更されたりするたびに新しい盤面に更新する動作を司る。

3.3　達成度

ある程度やりたい機能はアルゴリズムとしては実装できるが、盤面状態更新時にメソッドを呼び出されたいオブジェクトの登録受付するメソッド addListener、更新伝達する fireUpdate、メソッド所有先となるインターフェース BoardListener が新しく必要であった。また、途中で next を一回も押さずに盤面を塗った時に、undo ボタンを押すと塗られた盤面が元に戻るようになる必要があることが発覚したので、update メソッドを使用し、その部分の設計を行う ChangeUndoButton クラスを作成した。役割分担はおおよそできたが、細かい修正が必要な箇所は多かったので、達成度は 8 割程度と判断している。

4　プログラムの各機能の実現方法の詳細

4.1　盤面更新時の表示項目

盤面更新時には、その世代の生きているセルが黒色、死んでいるセルが白色で表示される。ボタンは next,undo,new game の 3 つがメインメソッド内 JButton を使用した呼び出しにより表示される。ボタンを押した際の各機能の説明は「作成したプログラムの操作方法と機能」の章で述べたとおりである。

4.2　next 機能の実現方法

next 機能に必要なメソッドやインスタンス、コンストラクタ等はは全て BoardModel クラス内に作成した。next 機能の要件にある生存条件、誕生条件、死亡条件は表のようになっている。まず条件判定を行う countlive メソッ

表 3: lifegame のルール

条件	周囲 8 セルにある生存セルの数
生存	2 つか 3 つ
死亡	1 つ以下 (孤立)、4 つ以上 (過密)
誕生	3 つ

ドを定義し、条件を書き加えた。特定のマスに隣接する 8 マスは、座標の値が x,y 共に 1 小さいか、同じか、1 大きい。そこで、x 座標を tate、y 座標を yoko という変数を用いて設定し、-1 から 1 でカウントすることで隣接する 8 マスの中の生きているマスの記憶を可能にした。例外処理として、自分自身 (完全に一致するマス) を除外したのち、隣接するが盤面外になってしまうマスは記憶対象から外すことで不具合をなくした。以下のようなコードとなっている。

```
private int countlive (int row, int col) {
 int count=0;
 for(int i=-1; i<=1; i++) {
  for(int j=-1; j<=1; j++) {
   if(i!=0 || j!=0) {
    int tate=row+i;
    int yoko=col+j;
    if(tate >=0 && tate<cols) {
     if(yoko>=0 && yoko<rows) {
      if(curcell[tate][yoko]) {
       count++;
```

```
      }
     }
    }
   }
  }
 }
 return count;
}
```

次に、次盤面の生存と死亡の変更を行い、盤面更新と保存するリストにデータを受け渡すメソッド next を定義した。以下のようなコードとなっている。隣接する 3 マスに生命があれば誕生か存命で必ず次の盤面では生き残り、2 マスにある場合は今の盤面で生きている場合のみ生き残るので、今の盤面を保存されている curcell 配列とともに判断し、真理値の切り替えを行っている。

```
public void next() {
 //盤面の状態をライフゲームのルールに従って 1 世代更新する。
 int i,j;
 for(i=0; i<rows; i++) {
  for(j=0; j<cols; j++) {

   int nextlive=countlive(i,j);
   if((nextlive==3) || (nextlive==2 && curcell[i][j])) {
    cells[i][j]=true;
   }
   else {
    cells[i][j]=false;
   }
   //changeCellState(i,j);
  }
 }
 System.out.println();
 this.fireUpdate();
}
```

curcell 配列を生成し、盤面保存を実際に行う save メソッドについては次の章で解説する。

4.3 undo 機能の実現方法

4.3.1 巻き戻しのための盤面の状態の記録方法（データ構造とその操作）

巻き戻しのための盤面の状態の記録の為に、save メソッドを定義した。まず next メソッドで使用するプリミティブ型配列 curcell と undo に利用する盤面保存の参照型配列 savecell に分け、それぞれ現在のデータが格納されている cell 配列からコピーした。次に、データが 32 個以上ある際には、if 文中で一番古い盤面を削除する動作を行っている。次に push コマンドにより今の盤面が最新の盤面としてデータ保存されるようにした。今回は 32 個の盤面を保存するためのリストとして linkedlist を利用した。push 命令と pop 命令を利用することにより、undo メソッド利用時に番号を受け渡す必要がないのが便利であり、講義資料通りであれば処理も早く実行できると考えたからである。以下のようなコードとなっている。

```
public void save() {

 int i,j;
 Boolean[][] savecell=new Boolean[rows][cols];
 for(i=0; i<rows; i++) {
  for(j=0; j<cols; j++) {
   curcell[i][j]=cells[i][j];
   savecell[i][j]=cells[i][j];
  }
 }
 if(history.size()>=32) {
  history.removeLast();
 }
 history.push(savecell);

}
```

また save メソッドをよびだすのは同じ BoardModel クラスの changeCellState メソッドである。このメソッドにより、undo ボタン呼び出し時以外の盤面に変更があった場合全て、つまりマウスドラッグにより盤面に書き加えられた時と next ボタンが押された場合両方に対応して書き換えとそれに伴うデータ保存が正常に行われるようになった。このメソッドは以下の部分である。

```
public void changeCellState(int y, int x) {
 // (x, y) で指定されたセルの状態を変更する。
 save();
 if(cells[x][y]) {
```

```
   cells[x][y]=false;
  }
  else {
   cells[x][y]=true;
  }
  this.fireUpdate();
 }
```

4.3.2　巻き戻し可能かどうかを判定する方法

この部分は BoardModel クラスの isUndoable メソッド内で機能実装を行い、Main クラス内で利用することにより特定条件下での undo の無効化を実現した。next を押した後 32 回分 undo を有効にする isUndoable メソッドは以下のように記述した。

```
public boolean isUndoable() {
  if(history.size()>=1) {
   return true;
  }
  else {
   return false;
  }
 }
```

また Main クラスの run メソッド内ではこのように記述した。初めにボタンの定義が行われている。

```
JButton undobutton = new JButton("undo");
buttonPanel.add(undobutton);
//初期状態や、32 回 undo を繰り返しデータがなくなった状態であるならば undo ボタンを無効化しておく
undobutton.setEnabled(model2.isUndoable());
//盤面編集、更新があった際に undo ボタンを有効化する
BoardListener listener = new ChangeUndoButton(undobutton);
model2.addListener(listener);
```

ここまでがボタンの設定と初期設定時、32 回 undo を繰り返しデータ未保存状態時の undo の無効化をする箇所である。次に next ボタンを押した際に、undo 機能が有効化される箇所だ。setEnable メソッドにより、BoardModel クラスの isUndoable メソッドでデータがあることを確認して、next 機能実行とともに、ボタンの有効化を行っている。

```
nextbutton.addActionListener(new ActionListener(){
   @Override
   public void actionPerformed(ActionEvent e) {
    model2.next();
    //next 実行後に undo が可能か再判定し、ボタンを有効に修正する。
    //ボタンを押したら undo 実行されるメソッド
    undobutton.setEnabled(model2.isUndoable());
    //画面表示の実行
    view.insertion(model2);
    view.repaint();
   }
  });
```

最後に undo クリック処理時の判定を説明する。undo 動作実行後、addListener メソッドで undo が可能か判定している。これは、next 機能ではなく盤面編集の巻き戻しである場合にも対応するためである。

```
  //undo ボタンクリック処理
  undobutton.addActionListener(new ActionListener(){
   @Override
   public void actionPerformed(ActionEvent e) {
    model2.undo();
    //undo 実行後に undo が可能か再判定し、不可能であるならばボタンを
無効に修正する。
    model2.addListener(listener);
    //画面表示の実行
    view.insertion(model2);
    view.repaint();
   }
  });
```

また next を一回も押していない条件下で盤面に色を塗った時に、undo ボタンを有効化し、塗る以前の状態に戻す機能を ChangeUndoButton クラスを用いて実装した。これは以下の部分による。初めにコンストラクタを生成し、ボタン感知の準備を整えた。次に、updated メソッドにより盤面更新を感知し、if 文の条件分岐によって、初期状態から更新があったら undo ボタンの無効化が解除されるようになっている。

```
package lifegame;
import javax.swing.*;
```

```
public class ChangeUndoButton implements BoardListener{
 JButton buttonUndo;

 ChangeUndoButton(JButton undoconst){
  buttonUndo=undoconst;
 }

 public void updated(BoardModel model2) {
  if(model2.isUndoable()) {
   buttonUndo.setEnabled(true);
  }
  else {
   buttonUndo.setEnabled(false);
  }
 }
}
```

4.4 盤面の描画において、セルの境界線の位置を計算する方法・計算式

この部分は BoardView クラスの paint メソッド内で実装されている。max はウィンドウの縦と横の長さの中で短い方の長さを取得する変数である。(詳細は後述) これをマス目の数に 1 を足したもので割り、1 マス数を掛けなおしてセルの境界線を作成すると、ウィンドウの大きさと境界線の間隔が比例関係で変化し、常に正方形のセルを作成することが可能になる。1 回目の for 文によって縦と横の境界線を 1 本ずつ生成し、2 回目の for 文により指定された場所の個所の塗りつぶし処理を行っている。

```
 public void paint(Graphics g) {
  int block=model1.getCols();
  int block1=block+1;
  super.paint(g); // JPanel の描画処理（背景塗りつぶし）
  // 直線や塗りつぶしの例
  max=square();
  max=max/block1;
  int i=1;
  for(i=1; i<block1+1; i++) {
   int t=max*i;
   g.drawLine(t, max, t, (max*block1));
```

```
   g.drawLine(max, t, (max*block1), t);
  }
  for(i=0; i<block; i++) {
   for(int j=0; j<block; j++) {
    if(model1.getcells(i,j)) {
     g.fillRect(max*(i+1), max*(j+1), max, max);

    }
   }
  }
 }
```

ウィンドウの縦と横の長さの中で短い方の長さを取得する変数 max は次のように定義されている。初めに縦と横の長さを取得している。if 文の条件分岐にて、短い方の数値を return で返すことにより、square という関数の仕様で長さ取得が可能になっている。20 で割る際、数値に余りが出てマス目の長さが 1 ずれるなどの影響ができることを避けるため事前に余りの値を引いて計算している。

```
private int max;
private int square(){
  int xmax=this.getWidth();
  int ymax=this.getHeight();
  if(ymax>xmax) {
   xmax=xmax-xmax%20;
   return xmax;
  }
  else {
   ymax=ymax-ymax%20;
   return ymax;
  }
 }
```

4.5 マウスカーソルの座標から対応するセルの座標を計算する方法・計算式

この部分は BoardView クラスの MousePressed メソッドと MouseDragged メソッドで実装されている。まず MousePressed メソッドについて解説する。これはマウスクリック時に動くメソッドである。まずウィンドウの横の長さに対して x 座標が、縦の長さに対して y 座標がどの位置にあるか、どの割合

の所にあるかを getX、getY メソッドを利用して取得して、セルの範囲内であれば塗りつぶし動作を行い、その時に塗り替えたマスを保存する動作を行っている。この座標は、ドラッグ動作時や undo 機能実行時に changeCellState 内の save メソッドで保存される動作により活用されている。

```
public void mousePressed(MouseEvent e) {
  int block=model1.getCols();
  int block1=block+1;
  max=square()/block1;
  int x=e.getX()/max-1;
  int y=e.getY()/max-1;
  if(x>=0 && x<block && y>=0 && y<block) {
   model1.changeCellState(y,x);
   xlast=x;
   ylast=y;
  }
  this.repaint();
 }
```

次に MouseDragged メソッドについて解説する。これはマウスのドラッグ動作時に動くものである。mousePressed メソッドでも保存した直前の座標を利用して、セル座標が有効範囲内でででかつ直前のセル範囲から変わった場合に限定してセルを塗り替える動作になっている。

```
public void mouseDragged(MouseEvent e) {
  int block=model1.getCols();
  int block1=block+1;
  max=square()/block1;
  int x=e.getX()/max-1;
  int y=e.getY()/max-1;

  //セル座標が有効範囲内でない場合には処理を中断 (範囲外におけるエラー表示を防ぐ)
  if (x < 0 || x >= block || y < 0 || y >= block) {
            return;
        }
  // セル座標が有効範囲内で、直前のセルと異なる場合のみ状態を変更
        if (x >= 0 && x < block && y >= 0 && y < block) {
            if (x != xlast || y != ylast) {
                model1.changeCellState(y, x);
                xlast = x;
```

```
                ylast = y;
                this.repaint();

            }
        }
}
```

4.6 新しいウィンドウを開く方法

この機能は Main クラスの大部分を占めている run メソッドをボタンを押された条件下のみ再帰的に呼び出すことにより、新規に別タブを開き新しい lifegame を最初から始めることを可能にしている。Main クラス run メソッド内の以下の部分によって実装されている。

```
JButton newbutton = new JButton("new game");
buttonPanel.add(newbutton);
newbutton.addActionListener(new ActionListener(){
 @Override
 public void actionPerformed(ActionEvent e) {
 SwingUtilities.invokeLater(new Main());
 }
});
```

5 拡張機能

5.1 all clear 機能

5.1.1 使い方と仕様

all clear ボタンが盤面の下部、next や undo ボタンと並列に設置されており、クリックすると盤面の黒で塗りつぶされていた部分が全て白に戻り、初期画面に戻すことができる。この機能を利用した際は undo 機能は有効化されており、all clear ボタンを押した直後に undo ボタンを利用すると、クリアする直前の状態の盤面が表示される。all clear ボタン処理の内部で next メソッドを利用しているので、クリアする直前の状態の盤面が表示された後は、最大 31 回盤面を巻き戻すことができる。

5.1.2　この機能の意義

ユーザーが一度盤面をリセットしなおしたいと思った際にこの拡張機能がない場合は new game 機能を利用するが別タブ表示される。何度もリセットを繰り返すと大量のタブが表示され、消去が非常に面倒となるデメリットがあった。このボタンを利用することで、同じタブ内でゲームを 1 からやり直すことができる。

5.1.3　アルゴリズム

BoardModel クラスに実際に盤面に表示するための真理値を初期化する allclear メソッドを作成した。二次元配列 cell の値を全て 0 にした後、fireUpdate で盤面を更新することによって、この機能が完成されている。

```
public void allclear() {
  int i,j;
  for(i=0; i<rows; i++) {
   for(j=0; j<cols; j++) {
    cells[i][j]=false;
   }
  }
  this.fireUpdate();
 }
```

次にボタンを押されてから実際に処理を行うメソッドに受け渡す部分について解説する。

```
JButton allbutton = new JButton("all clear");
buttonPanel.add(allbutton);
  allbutton.addActionListener(new ActionListener() {
   @Override
   public void actionPerformed(ActionEvent e) {
    model2.next();
    model2.allclear();
    //allclear 実行後に undo ボタンを有効に修正する。
    model2.addListener(listener);
    //画面表示の実行
    view.insertion(model2);
    view.repaint();

   }
  });
```

クリアするタイミングでは直前のデータが格納されていないので、一度盤面更新をしない状態で next メソッドを実行した後 allclear メソッドを実行することで、クリアする以前の盤面を save メソッドにより格納し、クリア機能の実現と undo 機能による復元を同時に実現した。必ず元の画面に復元する動作は可能なので、undo ボタンは有効化する。

5.2 盤面のマス目の個数を自由に決める機能

この機能はプログラミングDの指導書にも言及がある拡張機能である。

5.2.1 使い方と仕様

盤面を出す前にサイズを入力するポップアップ画面が表示される。数値を入力する場所と、ok ボタンと取消ボタンが存在する。利用者は 1 から 100 までの整数を半角で記入し、ok ボタンを押すことで指定したマス目数になっている盤面が表示される。ok ボタンを押すと数値が消去され次の画面に勧めないバグが起きることがあるが、その場合数値を入力した後 enter キーを 2 回押すと正常に盤面が表示される。以下動作中の画像である。

5.2.2 注意点

今回縦と横の長さは同じになるように制限してある。これはウィンドウ表示とマス目の長さが仕様上どうしてもいびつになり、ユーザーが違和感を持ったり気持ち悪いという印象を持つことにつながる可能性があると判断したからである。また入力する数値は 1 から 100 までという制限をしているが、これは数百マス以上の細かさにした場合、パソコンの表示最小単位よりセルの大きさが小さくなってしまい、塗りつぶし処理が失敗してエラー表示が出てしまうためである。また、整数型の数値入力を受け付ける関係で、半角の数字のみを受け付ける仕様となっている。

5.2.3 この機能の意義

盤面の細かさを変えられるようにすることで、固定された盤面のセルの挙動を見ることに飽きても長く楽しめるように出来ている。また、セルを細かくできることにより、最初に描く盤面をより自由に編集することができる。以下の画像は 100 × 100 マスの盤面でとある先生の名前を描写したものである。このように文字や絵などをより自由に描き、編集して挙動をたのしむという新しい取り組みもできる。

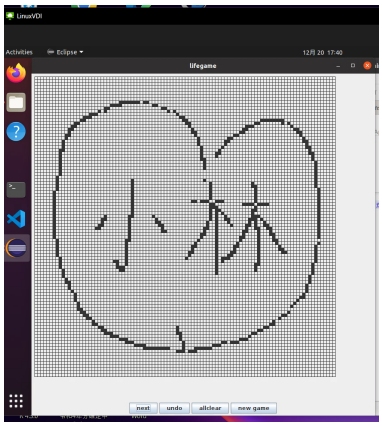

図 19: next 実行前

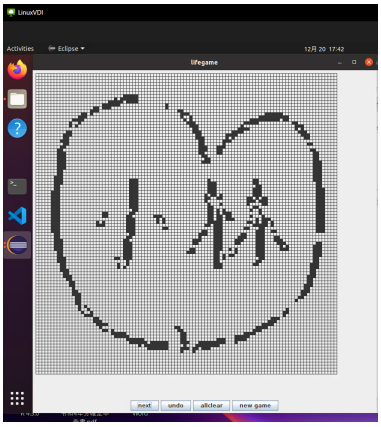

図 20: next 実行後

5.2.4 アルゴリズム

今回は全ての処理の前でポップアップ画面を表示する必要があるので、main クラスの run メソッドの一番最初に次のようなコードを記述した。

```
public void run() {
  String value = JOptionPane.showInputDialog(null, "盤面の大きさを 100 以下の整数で入力してください");
  int size=Integer.parseInt(value.trim());
  // BoardModel の作成と changeCellState の呼び出しを行う処理をここで実行。
  BoardModel model2=new BoardModel(size,size);
```

初めにポップアップ画面を表示する。しかし、入力受付は string 型と決められているので、文字から数値への変換と参照型からプリミティブ型への変換を 2 行目で同時に行うことにより、整数値のみ入力を受け付けることに成功した。今回は文字列や全角文字を入力したり、パソコンのセルより盤面が細かくなる位に大きな整数を入力した場合エラー表示されてしまうので、表示画面に 1 から 100 までの数値を入力するように記述した。

6 Java プログラミングの講義・演習で学習した内容

今回は 32 個の盤面を保存するためのリストとして linkedlist を利用した。push 命令と pop 命令を利用することにより、undo メソッド利用時に番号を受け渡す必要がないのが便利だからである。さらに、講義資料には、単なる配列のリストとなる Arraylist と比べると、先頭や末端のデータの除去や受け渡しを行う際、先頭や末端の受け渡された引数からデータを探索する時間が

短縮され、処理が早くなる事が分かった。追加調査や講義、テスト勉強などで学んだ内容により、今回使った linkedlist が最適であると実感することが出来た。プログラミングの講義や演習では、UI についても学習した。そこで自由度の高い拡張機能を作る際には、その機能が実際にユーザーにとってどんな恩恵をもたらすのかということを意識して設計し、レポートでも章として書き出すことで振り返ることが出来た。やみくもに難しい技術を使うのではなく、その技術を使って機能を実装することによりどのようなことに役立てられているかを意識することで、よりよいソフトウェア製品を作ることにつながるのではないかと考察した。