プログラミングDレポート

担当教員　小南大智
提出日　2023年12月21日
氏名　山久保孝亮
学籍番号　09B22084
メールアドレス　u327468b@ecs.osaka-u.ac.jp

1　プログラムの操作方法と機能

今回のライフゲームの課題は Run ボタンが押されるとゲームが開始する．私が作成した機能と操作方法は以下のとおりである．

1.1　盤面の描画

今回のライフゲームではゲームが開始した際にウィンドウが開き, そこに next,undo,newgame のボタンとともに盤面が表示されるという仕様になっている．盤面のサイズの初期値は縦 300, 横 400 ピクセルで最小値は縦, 横である．生きている状態のセルを黒色, 死んでいる状態のセルを灰色で表すこととした．最初はすべてのセルが死んでいる状態となっている．また, 盤面の縦と横の座標は以下の図 1 ような仕様とした．これは,(i,j) と指定されると横軸が i, 縦軸が j の行と列をそれぞれ考え, それらが交差するセルを表す．

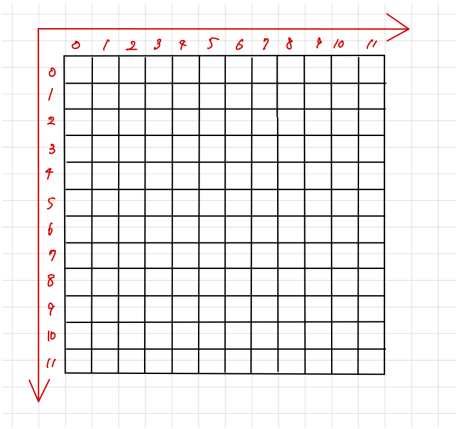

図 1: 座標の仕様

1.1.1　ウィンドウのサイズ変更

ウィンドウの大きさをユーザ側が変更した際にはその変更に合わせて盤面の大きさも変更される．以下の図 2 は Main クラス内のコードの一部である．赤く囲んだ部分の数字は縦のセルの個数, 青く囲んだ部分の数字は横のセルの個数を表す．ユーザはこの部分のコードを変更して盤面の個数を変更することができる．

```
public void run() {
    model = new BoardModel(12, 12,Undo);
    model.addListener(new ModelPrinter());
```

図 2: 該当部分のコード

ただしセルの形は常に正方形を保つ．パネル上にボタンを表示させる範囲を確保してから, そこを除いた部分に盤面を描写する．具体的な盤面内の座標の計算は実現方法のところで記述する．以下の図 2,3 はウィンドウを極端に横に大きくしたときの例と縦に大きくしたときの例である．

1.1.2　盤面のサイズ変更

盤面はそれぞれのセルの正方形が Main クラスで初期化されたセルの数だけそれぞれ縦と横に表示される．以下の 4,5 は RUN ボタンを押して出力した 12×12 と 18×12 の盤面である．

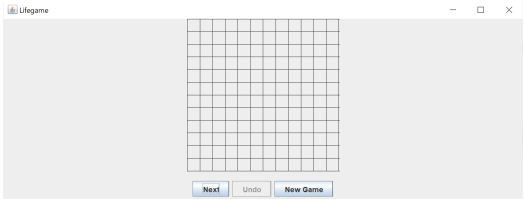

図 3: 極端に横に大きくしたときの盤面

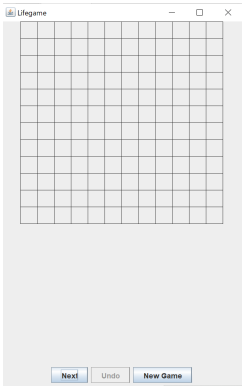

図 4: 極端に縦に大きくしたときの盤面

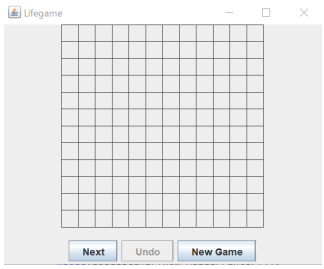

図 5: 12×12 の盤面

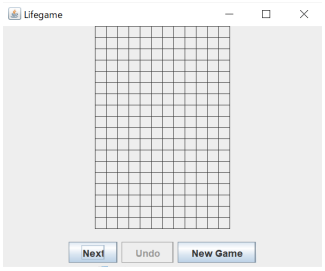

図 6: 18×12 の盤面

1.2 next,undo,newgame ボタン

ユーザは盤面の下部に表示されるそれぞれのボタンの上にマウスカーソルを移動し左クリックを押し込むことで以下の仕様を満たす処理を実行することができる. それぞれの処理は左クリックを押し込んで離したときに実行される.

1.2.1 next ボタン

next ボタンは最初から押せる状態になっており, ライフゲームの仕様に基づいて世代を一世代進める. 以下の図 6,7 は next ボタンを押して盤面の状態を一世代進めたときの例である.

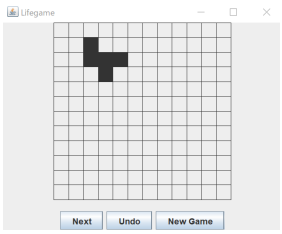

図 7: next ボタンを押す前の盤面

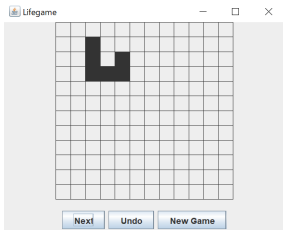

図 8: next ボタンを押した後の盤面

1.2.2 undo ボタン

undo ボタンは最初は押せない状態になっている.next ボタンを押して盤面の状態を 1 世代進めるか, マウスカーソルを使って盤面内をクリックまたはドラッグした時に押せるようになる. 最大で 32 の状態を記憶

しておき,33 回以上盤面の状態が変化した場合は直近の 32 個の盤面の状態を記憶しておく. 盤面の履歴がこれ以上ない状態まで巻き戻されると再び無効な状態に戻る. したがって,33 回盤面の状態が変化した際は,32 回 undo を押した段階で undo が押せなくなってしまう. また, 盤面の状態の記憶は 1 セルの状態が変化すると実行されるので,3 セル分ドラッグ操作をした場合はその操作で 3 つの状態が記憶されることになる. 以下の図 8 は RUN ボタンを押したときに Undo ボタンが押せなくなっている様子である.

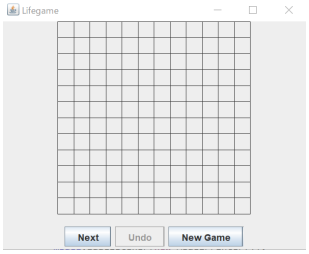

図 9: Undo ボタンが押せない様子

また, 以下の図 9,10 は undo ボタンを押して盤面の状態を一世代前に戻したときの例である.

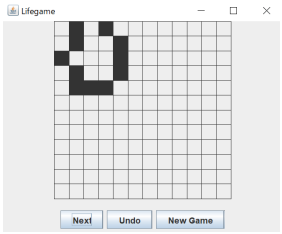

図 10: undo ボタンを押す前の盤面

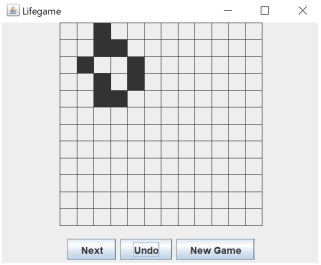

図 11: undo ボタンを押した後の盤面

1.2.3 newgame ボタン

newgame ボタンは最初から押せる状態になっており, ボタンを押すごとに新しく初期化されている盤面を新しいウィンドウで開く. 新しく作られたウィンドウはもともとあったウィンドウが作成された位置に作成される. ゲームはそれぞれ独立しており, 片方の盤面の状態を変化させてももう片方の盤面には影響されない. 以下の図 11 は newgame ボタンを押して新しくウィンドウを開き, 片方の盤面にだけマウスカーソルによって状態を変化させた様子である.

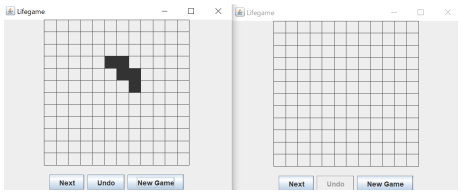

図 12: newgame ボタンを押して片方の盤面だけ変化させた様子

1.3　クリック, ドラッグしたときの処理

これから記述するクリック操作はマウスカーソルを動かさないで左クリックをすると実行できる. この操作はボタンが押し込まれた瞬間に実行される. また, ドラッグ操作は左クリックを押し込みながらマウスカーソルを移動させると実行できる. これもクリック操作と同様に, ボタンが押し込まれた瞬間に実行される.

1.3.1　盤面内のクリック, ドラッグ

上述のクリック操作により, 現在の盤面の状態は反転する. 例えば, もともと死んでいる状態のセルをクリックすると生きている状態のクリックに変更し, 生きている状態のセルをクリックすると死んでいる状態に変更する. 以下の図 12,13 は実行例である. この例では (4,3) のセルの状態を反転させている.

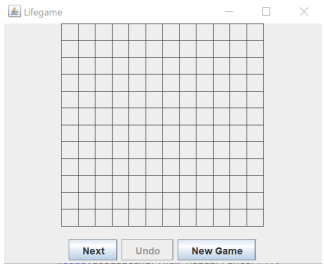

図 13: クリック前の盤面

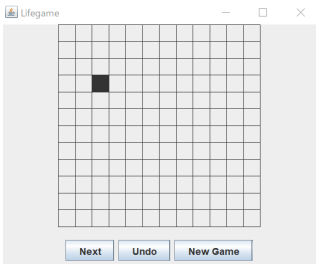

図 14: クリックした後の盤面

上述のドラッグ操作により, マウスカーソルがドラッグを開始したセル以外のセルに侵入した直後にそのセルの状態が変更される. ただし, 同じセル内を移動するだけだと結果としてはクリック操作と同じように押し込んだセルの状態を変更しただけとなる. 以下の図 14,15 はドラッグをした時の動作の例である.

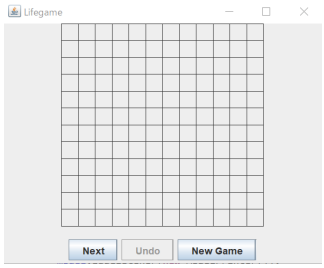

図 15: ドラッグ前の盤面

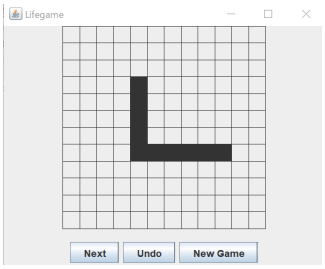

図 16: ドラッグ後の盤面

1.3.2　ドラッグ中にマウスカーソルが盤面以外に移動したときの処理

盤面外にマウスカーソルを移動させた場合には何も変化が起こらないという仕様にした. また, 図 16 のようにドラッグしながらマウスカーソルを移動させた場合, 侵入された盤面の状態のみを変更するので右の図 17 のように盤面が変更される.

1.3.3　盤面以外をクリックしたときの処理

盤面以外の場所をクリックした際はドラッグの時と同様に何も起こらないという仕様にした.

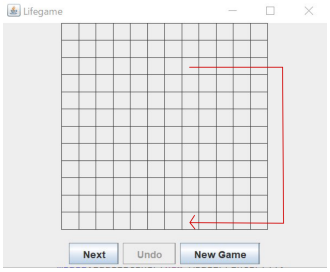

図 17: マウスカーソルの動かし方

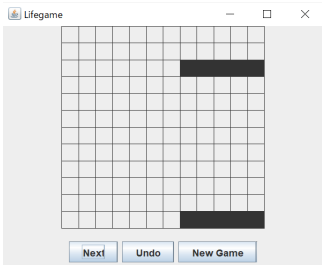

図 18: ドラッグ後の盤面

1.3.4 ドラッグ, クリック時の undo ボタンの巻き戻し

1.1.2 で記述した undo ボタンはクリック, ドラッグ操作による状態の変更も記憶して巻き戻すという仕様とした. ドラッグ操作に関しては, 各セル一つずつが状態変化するたびに新しく盤面が記憶される.

2 クラスと機能の対応表

今回作成したライフゲームのプログラムのクラスと機能の対応表は以下のようになる.

クラス名	機能
Main クラス	パネル内の情報の初期設定
BoardModel クラス	盤面の描画と世代更新, 巻き戻し処理
Button クラス	パネルのボタンが押された時の処理の分類
Boardlistener インターフェース	盤面の情報
BoardView クラス	マウスカーソルによる操作

表 1: クラスと機能の対応表

3 各機能実装方法

3.1 状態の更新に連動して画面表示の変更

今回のライフゲームでの画面表示が変更されるのは以下のような条件の時である.

- next ボタンや undo ボタンによる盤面の更新及び巻き戻し
- マウスカーソルによる盤面の状態の変更
-

これらの処理が行われた後に盤面が再描写されるための処理について記述する.
以下のコードはこの処理を実装するためにそれぞれのクラスから必要な個所のみを抜粋したコードである.

Listing 1: BoardListener インターフェース

```
1    public interface BoardListener {
2            public void updated(BoardModel m);
3    }
```

Listing 2: Main クラス

```
1    model.addListener(new BoardView(model));
```

Listing 3: BoardView クラス

```
1    public BoardView(BoardModel model) {
2                this.model = model;
3                this.addMouseListener(this);
4                this.addMouseMotionListener(this);
5                model.addListener(this);
6        }
7    @Override
8        public void updated(BoardModel model) {
9                this.repaint();
10       }
```

Listing 4: BoardModel クラス

```
1    private ArrayList<BoardListener> listeners;
2
3    public BoardModel(int c,int r,JButton undoButton) {
4                listeners = new ArrayList<BoardListener>();
5        }
6
7    public void changeCellState(int x,int y) {
8                this.fireUpdate();
9        }
10
11   public void addListener(BoardListener listener) {
12               listeners.add(listener);
13       }
14
15       private void fireUpdate() {
16               for(BoardListener listener:listeners) {
17                       listener.updated(this);
18               }
19       }
```

listing1 の 2 行目において,BoardListener インターフェースには updated クラスが呼び出されている. この updated クラスの処理内容は listing3 の 8 から 10 行目のように, 上で挙げた盤面が更新されるための処理を実装するクラス内に記述されている. listing2 のコードにより,BoardListener インターフェースを implement したクラスのオブジェクトを listing4 の 10 から 12 行目の addListener クラスで可変長配列 listeners に登録している. これを行うことで, 盤面を変更する各操作が実行されるたびに listing4 の 9 から 12 行目の ChangeCellState メソッドが呼び出されて盤面を変更し, その後で 18 から 21 行目の fireupdated メソッドが呼び出されて repaint が実行される. これにより, すべての盤面の変更が起こったときに listeners に登録されたオブジェクトを使って BoardListener を介して盤面の再描画が実行される.

3.2 巻き戻しのための盤面の記憶

巻き戻しのための盤面の記憶に関する処理は BoardModel クラスに記述した. 盤面の情報は可変長配列を持つリスト型である Arraylist を使用し, 変数名は History とした.

3.2.1 next による盤面の状態の変更の記憶

以下の listing は BoardModel クラスの next メソッドの一部である.

Listing 5: next メソッドの一部

```
1     if(counter!=32) {
2         counter++;
3         History.add(copiedcells);
4     }else {
5         History.remove(0);
6         History.add(copiedcells);
7     }
```

counter は History に記憶されている状態の数を表す int 型の変数で 0 に初期化されている.counter はメンバであるため counter の値はほかのメソッドにも共有される. 記憶されている状態の数が 32 個以下なら counter の値を 1 インクリメントしてから add メソッドを使って History に状態を記憶し,32 個なら History に記憶されている一番古い状態を削除してから add メソッドを使って現在の状態を新しく記憶している. また,copiedcells は lifegame の仕様に基づいて一世代進めたあとの盤面の状態を表す二次元配列である. これは現在の盤面の状態を表す二次元配列 cells のすべての要素をコピーしたものである.copiedcells を使用している理由としては, 次の世代へ進めたときに各セルが生きているかはセルの周りに生きているセルが何個存在するかで決定されるので全ての cells の要素に参照している途中で cells の状態を変えてしまうと周りに存在する生きているセルの数が変化してしまうためである.

3.2.2 undo による盤面の状態の変更の記憶

以下の listing は BoardModel クラスの undo メソッドの一部である.

Listing 6: undo メソッドの一部

```
1     History.remove(counter);
2     counter--;
3     cells = History.get(counter);
```

まず一番最新の状態を History から削除し, そのあとに counter を 1 デクリメントする. これにより undo が押された際に記憶されていた状態が 1 つ減少する. その後現在の状態を表す二次元配列 cells にデクリメントした後の counter がさす History の最新の状態, 即ち undo を押したときの直前の状態を格納する.

3.2.3 マウスカーソルによる盤面の状態の変更の記憶

以下の listing は BoardModel クラスの changeCellState メソッドの一部である.

Listing 7: changeCellState メソッドの一部

```
1     boolean[][] currentBoard = new boolean[rows][cols];
2     for (int i = 0; i < rows; i++) {
3         System.arraycopy(cells[i], 0, currentBoard[i], 0, cols);
4     }
5     if(counter!=32) {
6         counter++;
7         History.add(currentBoard);
8     }else {
9         History.remove(0);
```

```
10          History.add(currentBoard);
11      }
```

マウスカーソルによる盤面の状態の変更の詳細は後述するが, どのセルの状態を変更すればよいかを特定してから changeCellState を呼び出して実際に盤面の状態を格納している cells を変更する. 二次元配列である currentBoard を用意して変更後の盤面の状態を 2 から 4 行目でコピーする．その後 counter の値が 32 以下なら counter の値を 1 インクリメントしてから add メソッドを使って History に状態を記憶し,32 個なら History に記憶されている一番古い状態を削除してから add メソッドを使って現在の状態を新しく記憶している．add した配列が cells ではなく currentBoard である理由は,cells を格納してしまうと参照の追加になり,cells が変更されると History に追加されているほかのオブジェクトも同じように変更されてしまうためである．ゆえに一時的に cell の内容を別の二次元配列にコピーして参照の追加となることを避けた.

3.3 undo による巻き戻し可能かどうかの判定

undo は盤面を生成したとき Main メソッド内で setEnabled メソッドを使って押せないように初期化されている.undo による巻き戻しが可能となるのは仕様でも説明した通り next ボタンによる世代の更新かマウスカーソルによる盤面の状態の変更が行われた時である.

- next ボタンによって世代更新が行われた時,3.2.1 で記述したように counter の値が更新された回数だけインクリメントされる. これを利用して BoardModel メソッド内に isUndoable メソッドを作成した. このメソッドは counter の値が 0 より大きいときは true を,0 のときは false を返す. 以下の listing のコードは Button クラス内の next ボタンと undo ボタンが押された時に実行される処理である.

Listing 8: next ボタン及び undo ボタンを押したときに実行される処理

```
1       case 1:
2           boardModel.next();
3           undoButton.setEnabled(true);
4           break;
5       case 2:
6           if(boardModel.isUndoable()) {
7               boardModel.undo();
8               if(!boardModel.isUndoable()) {
9                   undoButton.setEnabled(false);
10              }
11          }
12          break;
```

1 から 4 行目のように,next ボタンが押された時には next メソッドを呼び出して盤面の世代を更新してから setEnabled メソッドで undo ボタンを押せるようにしている. また,6 から 11 行目のように isUndoable の返り値が true の時は undo メソッドを呼び出して盤面の世代を一つ戻した後に再び isUndoable メソッドを呼び出して返り値が false であれば setEnabled メソッドで undo ボタンを押せなくしている．これは,undo メソッドによって世代を戻したときに History が空になったことをすぐに確認するためである. これを実装することで Undo ボタンが履歴がなくなった時に押せなくなることを実現できる.

- マウスカーソルによって盤面に変更が加えられたときは仕様で述べた通り, セルの状態が一つでも変わるとそれが一つの盤面の状態として記憶される.3.2.3 で記述した通り, マウスカーソルで状態が変更されると changeCellState メソッドが呼び出されるのでこのメソッドが呼び出された時に setEnabled メソッドで undo ボタンを押せるようにすれば巻き戻しを可能にすることが実現できる.

3.4　セルの境界線の位置を計算する方法

以下の表 2 は計算するために使用した変数名と表す内容である.

変数名	変数があらわす内容
M	盤面の横のセル数
N	盤面の縦のセル数
width	パネルの横幅
height	パネルの縦幅
masu	盤面に描写する正方形のマスの一辺の長さ

表 2: 計算式中の変数名とその内容

盤面のセルの境界線の位置は以下のような手順で盤面の境界線の位置は計算される.

1. 現在の width と height から,masu を決定する. 具体的には以下のような不等式が成り立つかどうかについて考える.

$$\frac{height \times \frac{99}{100}}{N} > \frac{width}{M+2} \tag{1}$$

左辺は height の 99%の大きさを N 等分した時の長さを調べている. これは,height の 99%の長さを考えないと盤面の一番下の横線が表示されないことがあるという問題を解決するための処理である. 右辺は width を M+2 等分したときの長さを調べている. これは, パネルの中央に盤面を配置するための処理である.(1) が成り立った時はパネルが縦に長い状態であると判断して

$$masu = \frac{width}{M+2} \tag{2}$$

とし,(1) が成り立たなかったときはパネルが横に長い状態であると判断して

$$masu = \frac{height \times \frac{99}{100}}{N} \tag{3}$$

となるようにした.

2. 次に, 盤面の開始位置を決定する. 縦軸の開始位置は y 座標が 0 からである.x 座標の開始位置はパネルの中央に盤面を配置するために, 以下のような数式の値を開始位置とする.

$$\frac{width - masu \times M}{2} \tag{4}$$

この数式はパネルの横幅から盤面を表示している部分の長さを引いた長さを二等分した値になる. これにより, 盤面を描写していない部分の横幅が同じ長さとなっている.

3. ユーザが指定したセル数分だけ縦軸と横軸を描画する. それぞれのセルの一片の長さは (2) または (3) の値である.

3.5　マウスカーソルの座標からセルの座標を計算する方法

以下の表 3 は計算するために使用した変数名と表す内容である.
3.4 の表 2 の変数と同じ変数名のものは値も同じである. 以下のような流れでマウスカーソルがある位置のセルの座標はクリックとドラッグで異なる方法で計算される. まずクリックとドラッグで共通の処理を記述

変数名	変数があらわす内容
M	盤面の横のセル数
N	盤面の縦のセル数
width	パネルの横幅
height	パネルの縦幅
masu	盤面に描写する正方形のマスの一辺の長さ
x	マウスカーソルがある場所の x 座標
y	マウスカーソルがある場所の y 座標
masu_x	0 で初期化されており, マウスカーソルが現在存在するセルの左上の頂点の x 座標
masu_y	0 で初期化されており, マウスカーソルが現在存在するセルの左上の頂点の y 座標
StateX	マウスカーソルが現在存在するセルの左上の頂点の x 座標
StateY	マウスカーソルが現在存在するセルの左上の頂点の y 座標

表 3: 計算式中の変数名とその内容

する. 盤面外にマウスカーソルがある状態でクリックやドラッグがされると何もしないようにするために x と y が以下の条件式を満たしているか確認する.

$$x < \frac{width - masu \times M}{2} \quad or \quad x > \frac{width + masu \times M}{2} \quad or \quad y > masu \times (N) \tag{5}$$

これらの 3 つの不等式の一番左の不等式はマウスカーソルの x 座標が盤面よりも左側にあるとき真になる. 真ん中はマウスカーソルの x 座標が盤面よりも右側にあるとき真になる. 一番右の不等式はマウスカーソルの y 座標が盤面よりも下側にある時に真になる. そしてこれら 3 つの不等式は全て or としているのでこれらのうちの 1 つでも真になれば盤面外にマウスカーソルがあるという判定になる.

3.5.1 クリック時の計算

クリック時のマウスカーソルの位置からのセルの座標の計算は以下のような流れで実行される.

1.

3.6 マウスカーソルによる盤面の状態の変更

マウスカーソルによる盤面の状態の変更に関する処理は BoardView クラスに記述した. 以下にクリックに関する処理とドラッグに関する処理の実装方法を記述する. この処理を行うにあたって使用したメンバは以下のとおりである.

また,masu の計算は 2.3 で記述した通りである.

クリック時の処理

クリック時の処理は mousepressed メソッド内に記述した. mousepressed メソッド内で使用する内部変数は以下のとおりである. クリック操作時の具体的な処理内容は以下のような流れで実行される.

1. x,y の宣言及び初期化をする.x,y は getcol 関数により取得される.
2. マウスカーソルが盤面内に存在するかどうかを判定し, 存在すれば 3 以降の処理を実行する. 存在しなければクリックの処理を return 文により終了する

変数名	変数があらわす内容
width	パネルの横幅を paint メソッド内の getWidth メソッドによって格納する int 型の変数
height	パネルの縦幅を paint メソッド内の getHeight メソッドによって格納する int 型の変数
masu	盤面に描写する正方形のマスの一辺の長さを格納する int 型の変数
M	盤面の横のマス数
N	盤面の縦のマス数
StateX	マウスカーソルが現在存在するセルの左上の頂点の x 座標を格納する int 型の変数
StateY	マウスカーソルが現在存在するセルの左上の頂点の y 座標を格納する int 型の変数

表 4: メンバとその内容

変数名	変数があらわす内容
x	マウスカーソルがある場所の x 座標を格納する int 型の変数
y	マウスカーソルがある場所の y 座標を格納する int 型の変数
i	for 文を使う際のカウンタ
j	for 文を使う際のカウンタ

表 5: 内部変数名とその内容

3. for 文を使って今現在のマウスカーソルが存在する座標のセルの y 座標を特定. このとき, 特定するための条件式 $y >= i * masu \&\& y < (i+1) * masu$ は, カウンタを表す変数 i によって盤面のセルのどの行に y が存在するかを判定している.
4. 3 で y がどの行にあるかを特定した後, 同様にして x がどの列にあるのかを判定する. 条件式は $x >= \frac{(wodth - masu * M)}{2} + masu * j \&\& x < \frac{width - masu * M}{2} + masu * (j+1)$ となる.2 の条件式との違いは, 盤面を描写し始める位置が異なるためである.
5. StateX と StateY を 3 と 4 で条件分岐をした時の i と j を使ってマウスカーソルが存在するセルの左上の頂点の座標を格納する. これは, 後のドラッグ操作ための処理であるため後に記述する. そのあと break 文を二回使って for 文から抜ける.
6. 特定した i と j を changeCellState メソッドの引数として使用しセルの状態を変える. そして paint メソッドを使って盤面を描写する.

ドラッグ時の処理

ドラッグ時の処理は mouseDragged メソッド内に記述した. mouseDragged メソッド内で使用する内部変数は以下のとおりである. クリック操作時の具体的な処理内容は以下のような流れで実行される.

変数名	変数があらわす内容
x	マウスカーソルがある場所の x 座標を格納する int 型の変数
y	マウスカーソルがある場所の y 座標を格納する int 型の変数
masu_x	0 で初期化されており, マウスカーソルが現在存在するセルの左上の頂点の x 座標を格納する int 型の変数
masu_y	0 で初期化されており, マウスカーソルが現在存在するセルの左上の頂点の y 座標を格納する int 型の変数
i	for 文を使う際のカウンタ
j	for 文を使う際のカウンタ

表 6: 内部変数名とその内容

1. クリック時の処理の 1 から 4 と同様の処理が行われる. また, 最初に masu_x と masu_y を 0 で初期化する.
2. StateX と StateY ではなく,masu_x と masu_y に特定した i,j を使って x と y が存在するセルの左上の頂点の座標を格納する.
3. masu_x と masu_y がそれぞれ StateX と StateY と一致しているかを判定する. 即ち直前にマウスカーソルがあったセルと今現在マウスカーソルが存在しているセルが一致しているかどうかを判定している. 一致していれば何も処理をせずに終了し, 一致していなければ以下の 4 の処理を行う.
4. for 文を二回使用してクリック時の StateX や StateY を求めた時と同じように changeCellState メソッドと repaint メソッドを使って変更された盤面を表示する.

ドラッグ操作では, クリック操作とは違って現在のセルの位置を格納する変数として masau_X と masu_y を使用した. これは, ドラッグ操作をする際に同じマス中の移動の際に盤面の状態が何度も変更されて最初の状態が維持されないという問題が発生したからである b. これを解決するために直前の状態を保存しておくことで同じセル内を移動するときには盤面の状態を変化させないようにした.

3.7 新しいウィンドウを開く

新しいウィンドウを開くことに関する処理はクラスのメソッドに記述した. この処理を行うにあたって使用したメンバは以下のとおりである.

変数名	変数があらわす内容
M	盤面の横のマス数
N	盤面の縦のマス数
width	パネルの横幅を paint メソッド内の getWidth メソッドによって格納する int 型の変数
height	パネルの縦幅を paint メソッド内の getHeight メソッドによって格納する int 型の変数
masu	盤面に描写する正方形のマスの一辺の長さを格納する int 型の変数

表 7: メンバとその内容

3.8 拡張機能

3.8.1 実現方法

4 プログラミングの講義・演習で学習したこと

今回のプログラミング