Homework3

Name: Padmanabha Desai

1. Model for Minesweeper game:

**Minesweeper:**

Description: The main class that initializes the game and manages the game state.

Attributes:

rows: Number of rows in the game grid.

cols: Number of columns in the game grid.

numMines: Number of mines in the game.

grid: 2D array of Cell objects representing the game grid.

gameState: Current state of the game.

mineGenerator: Object responsible for generating mines.

Operations:   
startGame(), revealCell(int row, int col), flagCell(int row, int col), and resetGame()

**Cell:**

Description: Represents a single cell in the game grid.

Attributes:

isMine: Boolean indicating if the cell is a mine.

isRevealed: Boolean indicating if the cell is revealed.

neighborMines: Number of neighboring cells containing mines.

Operations:

reveal(): Reveals the cell.

flag(): Flags the cell as potentially containing a mine.

**MineGenerator**:

Description: Responsible for randomly placing mines on the grid.

Operations:

placeMines(): Places mines randomly on the grid.

**GameState:**

Description: Interface representing the various states of the game.

Operations:

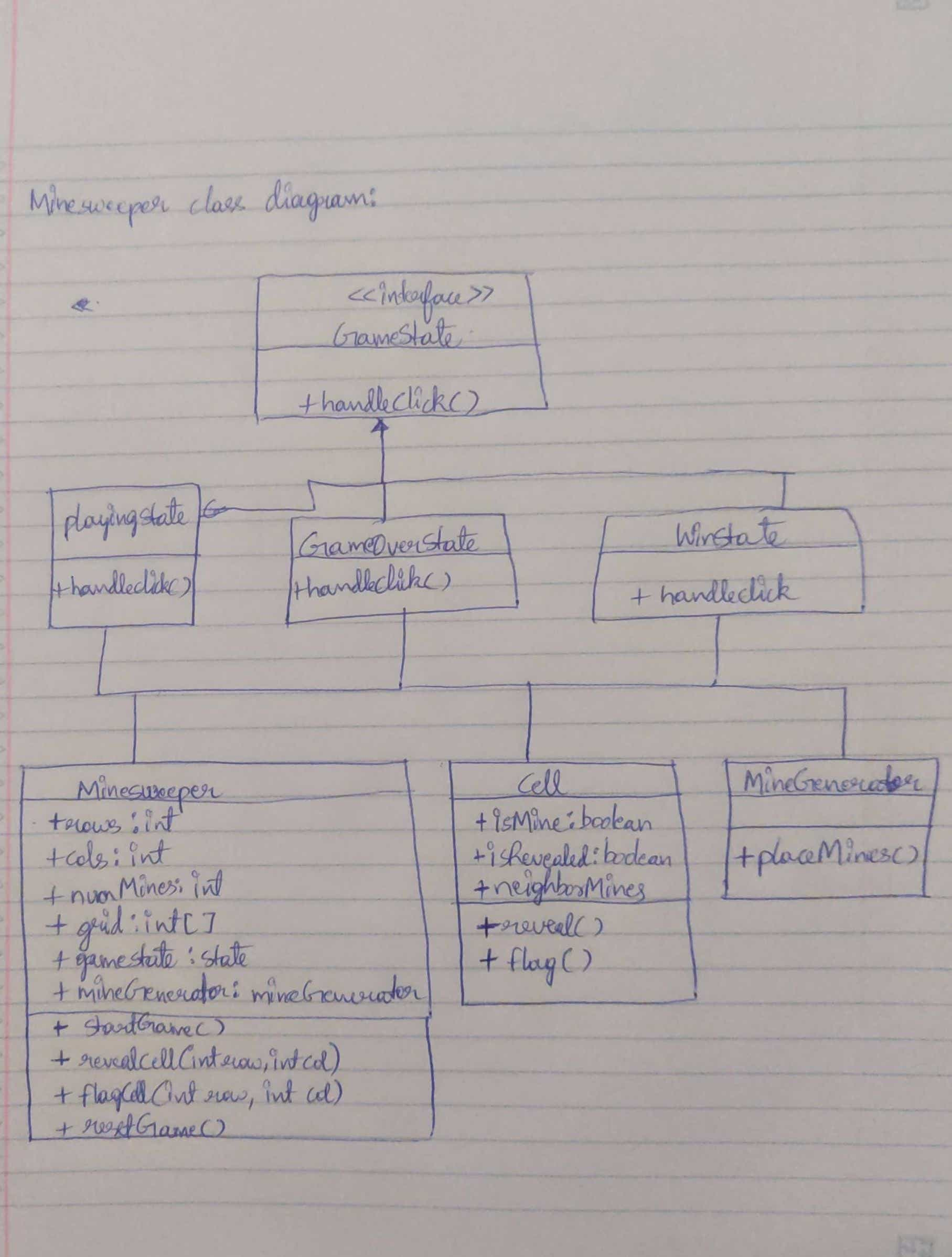
handleClick(): Handles user clicks on cells.

**PlayingState, GameOverState, WinState:**

Description: Concrete implementations of GameState interface representing different states of the game.

Operations:

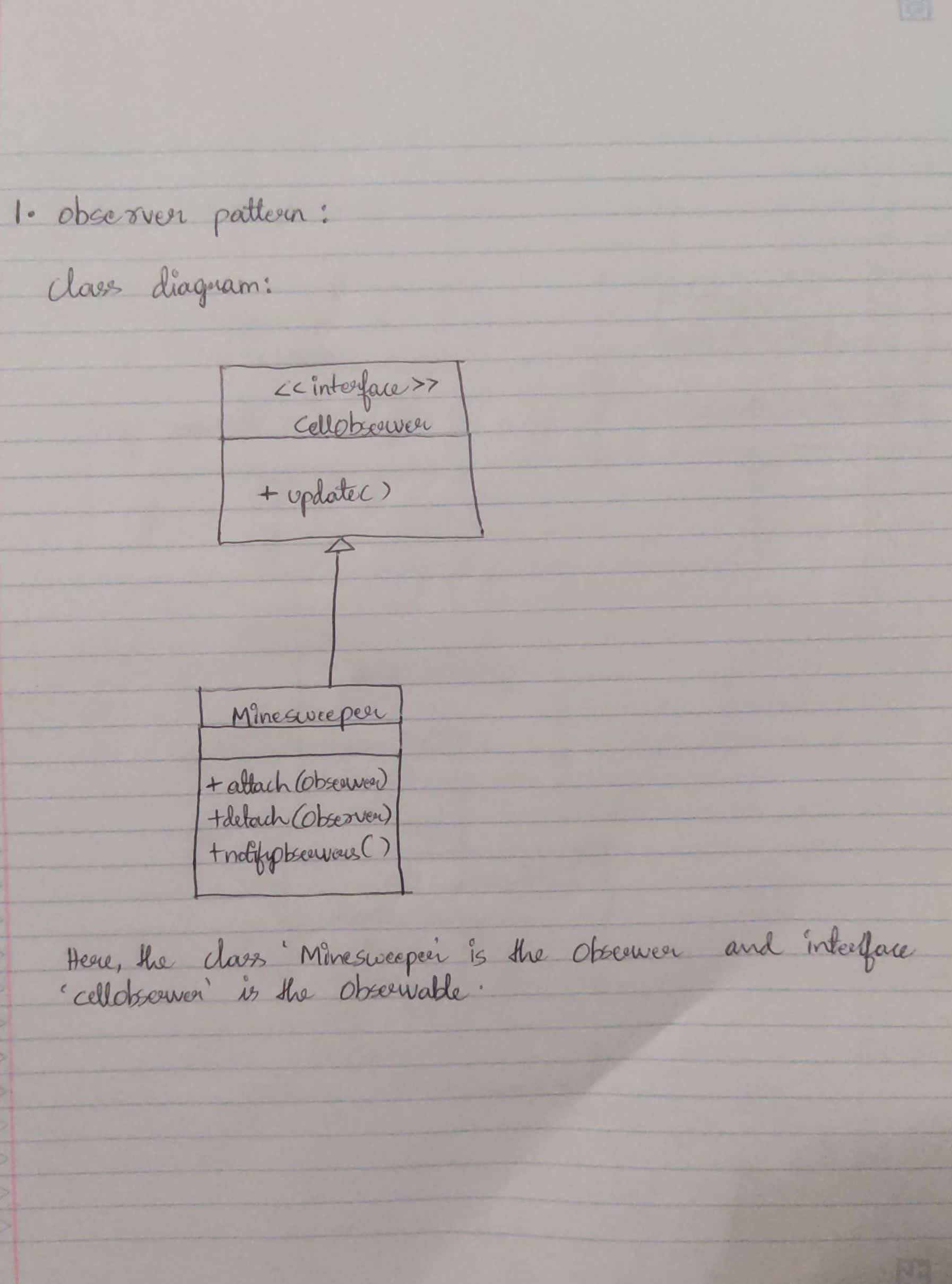
handleClick(): Implements behavior for handling clicks in each state.



**Design Patterns**

a. Observer Pattern

Diagram:



How it is Used:

The Minesweeper class acts as the subject, and the Cell class acts as the observer.

When a cell is revealed, it notifies the Minesweeper object, which then updates the GUI.

Advantages:

Gives clean solution to GUI update: Decouples the game logic from GUI, allowing for easier GUI updates without changing game logic.

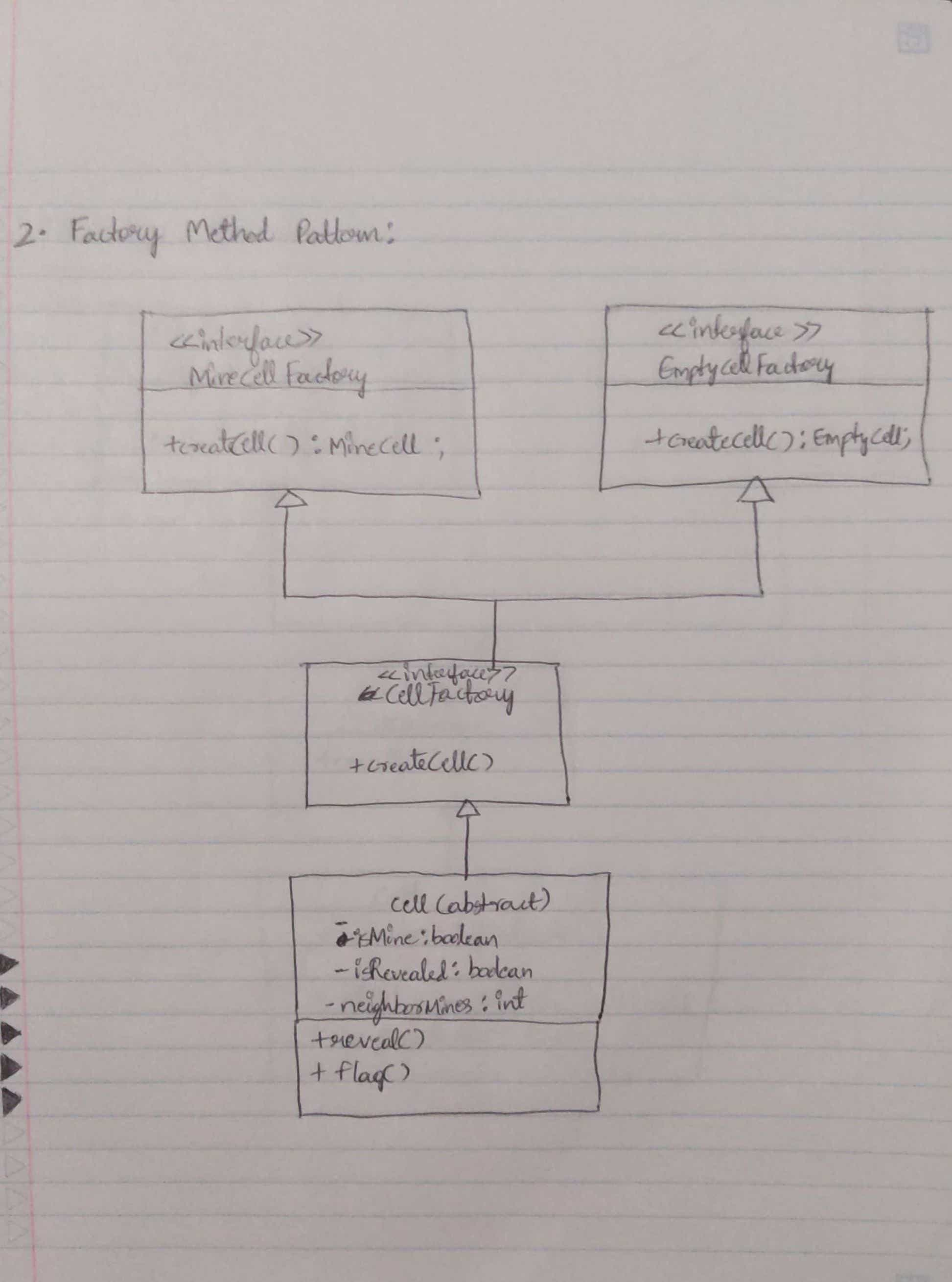
Enhances maintainability: Easy to add new GUI elements or change existing ones without affecting game logic.

Drawbacks:

Overkill for small projects: In simpler games, direct GUI updates might suffice without the Observer pattern.

**b. Factory Method Pattern**

Diagram:



How it is Used:

The CellFactory class provides a method createCell() which returns different types of Cell objects.

Different types of cells (e.g., MineCell, EmptyCell) extend Cell.

Advantages:

Enhances extensibility: New cell types can be added by creating subclasses without modifying existing code.

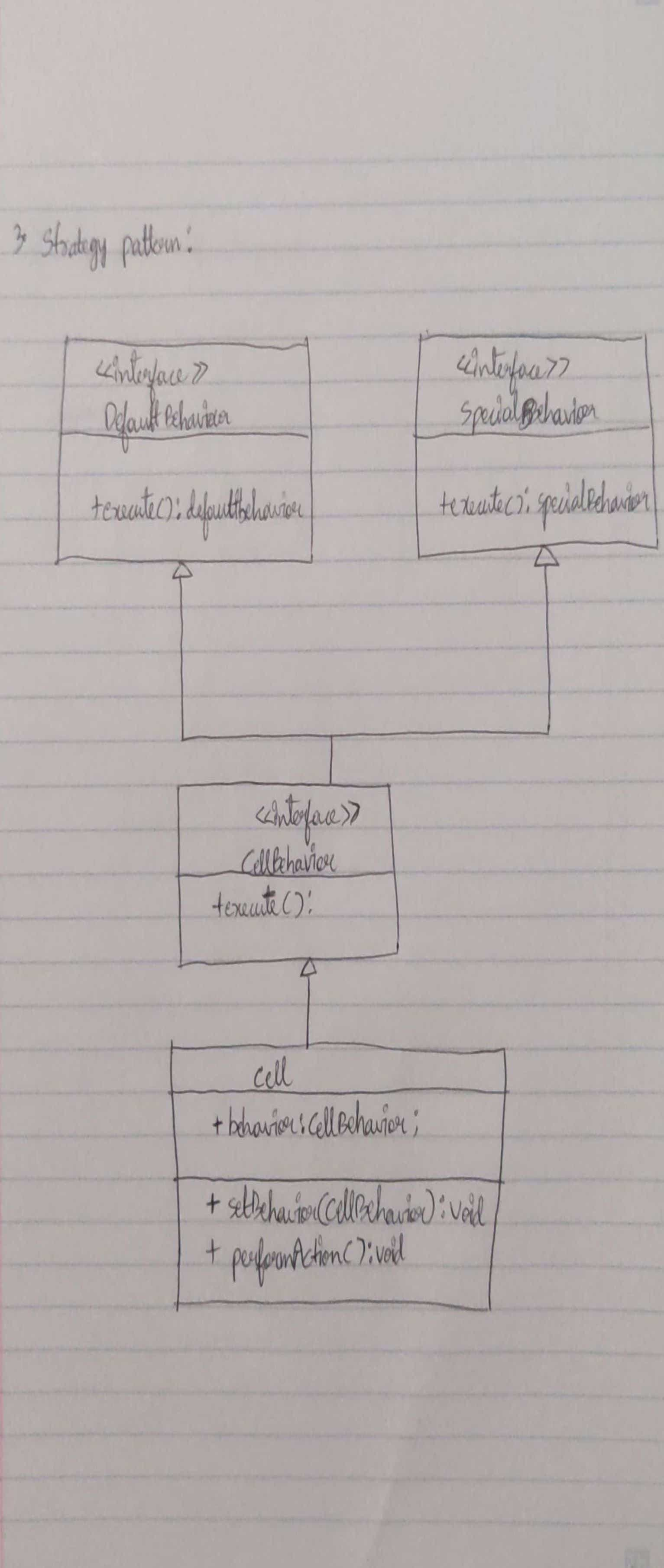
Simplifies object creation: Centralizes cell creation logic, making it easy to manage and modify.

Drawbacks:

May introduce complexity: Overhead of creating a factory class might be unnecessary for simple projects.

**c. Strategy Pattern**

Diagram:



How it is Used:

The Cell class has a CellBehavior interface, with concrete implementations like DefaultBehavior and SpecialBehavior.

Each cell can have a different behavior, set at runtime.

Advantages:

Flexible behavior: Cells can have different actions or behaviors without changing the Cell class.

Simplifies adding new behaviors: Just create a new class implementing CellBehavior.

Drawbacks:

Increased number of classes: Might introduce more classes, which can clutter the codebase if not managed properly.

Other Possible Patterns,

Composite Pattern:

The Cell class could be part of a composite structure for the grid.

State Pattern:

The game state management (PlayingState, GameOverState, WinState) can be further elaborated using the State pattern.

Singleton Pattern:

Ensuring only one instance of Minesweeper or MineGenerator might be useful.

These design patterns can significantly improve the flexibility, maintainability, and extensibility of the Minesweeper game codebase, making it easier to add new features or modify existing ones in the future.

**Here is a basic implementation of Minesweeper in Java using Swing:**

AI prompt to ChatGPT:

create the basic Minesweeper game in Java using the Swing library. The game will consist of a grid of squares where some squares contain mines. The player's objective is to uncover all squares that do not contain mines without triggering a mine.

Code:  
import javax.swing.\*;

import java.awt.\*;

import java.awt.event.ActionEvent;

import java.awt.event.ActionListener;

import java.util.Random;

public class Minesweeper extends JFrame {

private final int rows = 10;

private final int cols = 10;

private final int numMines = 10;

private final JButton[][] grid;

private final boolean[][] isMine;

private final boolean[][] isRevealed;

private int remainingCells;

public Minesweeper() {

setTitle("Minesweeper");

setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

setLayout(new GridLayout(rows, cols));

grid = new JButton[rows][cols];

isMine = new boolean[rows][cols];

isRevealed = new boolean[rows][cols];

remainingCells = rows \* cols - numMines;

initializeGrid();

placeMines();

setSize(500, 500);

setLocationRelativeTo(null);

setVisible(true);

}

private void initializeGrid() {

for (int i = 0; i < rows; i++) {

for (int j = 0; j < cols; j++) {

grid[i][j] = new JButton();

final int row = i;

final int col = j;

grid[i][j].addActionListener(new ActionListener() {

@Override

public void actionPerformed(ActionEvent e) {

revealCell(row, col);

}

});

add(grid[i][j]);

}

}

}

private void placeMines() {

Random random = new Random();

int minesPlaced = 0;

while (minesPlaced < numMines) {

int randRow = random.nextInt(rows);

int randCol = random.nextInt(cols);

if (!isMine[randRow][randCol]) {

isMine[randRow][randCol] = true;

minesPlaced++;

}

}

}

private void revealCell(int row, int col) {

if (isRevealed[row][col]) {

return;

}

isRevealed[row][col] = true;

grid[row][col].setEnabled(false);

remainingCells--;

if (isMine[row][col]) {

grid[row][col].setText("X");

gameOver();

} else {

int count = countAdjacentMines(row, col);

if (count > 0) {

grid[row][col].setText(String.valueOf(count));

} else {

for (int i = row - 1; i <= row + 1; i++) {

for (int j = col - 1; j <= col + 1; j++) {

if (isValidCell(i, j)) {

revealCell(i, j);

}

}

}

}

if (remainingCells == 0) {

JOptionPane.showMessageDialog(this, "You win!");

resetGame();

}

}

}

private void gameOver() {

JOptionPane.showMessageDialog(this, "Game Over");

resetGame();

}

private void resetGame() {

for (int i = 0; i < rows; i++) {

for (int j = 0; j < cols; j++) {

grid[i][j].setText("");

grid[i][j].setEnabled(true);

isMine[i][j] = false;

isRevealed[i][j] = false;

}

}

placeMines();

remainingCells = rows \* cols - numMines;

}

private int countAdjacentMines(int row, int col) {

int count = 0;

for (int i = row - 1; i <= row + 1; i++) {

for (int j = col - 1; j <= col + 1; j++) {

if (isValidCell(i, j) && isMine[i][j]) {

count++;

}

}

}

return count;

}

private boolean isValidCell(int row, int col) {

return row >= 0 && row < rows && col >= 0 && col < cols;

}

public static void main(String[] args) {

SwingUtilities.invokeLater(Minesweeper::new);

}

}

1. Observer Pattern for GUI Updates

Issue: The current implementation directly updates the GUI components when a cell is revealed or game over occurs.

Solution: Implement the Observer pattern to decouple the game logic from the GUI. The GUI listens for updates from the game logic and updates itself accordingly.

2. Factory Method for Cell Creation

Issue: Cell creation is directly done in the Minesweeper class, making it hard to extend or modify cell types.

Solution: Implement a CellFactory using the Factory Method pattern. This allows easy addition of new cell types without modifying existing code.

3. Strategy Pattern for Different Cell Behaviors

Issue: Currently, all cells have the same behavior. If we want different behaviors (e.g., special cells that reveal nearby cells), the code needs to be modified.

Solution: Implement the Strategy pattern to define different behaviors for cells. Each cell type can have its own strategy.

4. Composite Pattern for Grid Representation

Issue: The grid is represented as a 2D array of buttons. This tightly couples the game logic with the GUI.

Solution: Implement the Composite pattern to create a hierarchical structure for the grid. Each cell can be a leaf node, and the grid itself can be a composite.

5. State Pattern for Game State Management

Issue: The game state management (win, lose, in-progress) is handled within the Minesweeper class.

Solution: Implement the State pattern to define different states of the game (e.g., PlayingState, GameOverState, WinState).

Here is an updated version of the Minesweeper class with some of these design patterns implemented:

Code:   
import javax.swing.\*;

import java.awt.\*;

import java.awt.event.ActionEvent;

import java.awt.event.ActionListener;

import java.util.Random;

// CellFactory interface

interface CellFactory {

Cell createCell();

}

// Concrete Creator - MineCellFactory

class MineCellFactory implements CellFactory {

@Override

public Cell createCell() {

return new MineCell();

}

}

// Concrete Creator - EmptyCellFactory

class EmptyCellFactory implements CellFactory {

@Override

public Cell createCell() {

return new EmptyCell();

}

}

// Product - Abstract Class

abstract class Cell {

boolean isMine;

boolean isRevealed;

int neighborMines;

abstract void reveal();

abstract void flag();

}

// Concrete Product - MineCell

class MineCell extends Cell {

@Override

void reveal() {

isRevealed = true;

System.out.println("MineCell revealed");

// Additional logic for MineCell reveal

}

@Override

void flag() {

System.out.println("MineCell flagged");

// Additional logic for MineCell flag

}

}

// Concrete Product - EmptyCell

class EmptyCell extends Cell {

@Override

void reveal() {

isRevealed = true;

System.out.println("EmptyCell revealed");

// Additional logic for EmptyCell reveal

}

@Override

void flag() {

System.out.println("EmptyCell flagged");

// Additional logic for EmptyCell flag

}

}

// Minesweeper class

public class Minesweeper extends JFrame {

private final int rows = 10;

private final int cols = 10;

private final int numMines = 10;

private final Cell[][] grid;

private GameState gameState;

private final CellFactory cellFactory;

public Minesweeper() {

setTitle("Minesweeper");

setDefaultCloseOperation(JFrame.EXIT\_ON\_CLOSE);

setLayout(new GridLayout(rows, cols));

grid = new Cell[rows][cols];

gameState = new PlayingState();

cellFactory = new MineCellFactory(); // Default factory

initializeGrid();

startGame();

setSize(500, 500);

setLocationRelativeTo(null);

setVisible(true);

}

private void initializeGrid() {

for (int i = 0; i < rows; i++) {

for (int j = 0; j < cols; j++) {

grid[i][j] = cellFactory.createCell();

final int row = i;

final int col = j;

grid[i][j].addActionListener(new ActionListener() {

@Override

public void actionPerformed(ActionEvent e) {

gameState.handleClick(row, col);

}

});

add(grid[i][j]);

}

}

}

private void startGame() {

// Place mines on the grid

Random random = new Random();

int minesPlaced = 0;

while (minesPlaced < numMines) {

int randRow = random.nextInt(rows);

int randCol = random.nextInt(cols);

if (!grid[randRow][randCol].isMine) {

grid[randRow][randCol].isMine = true;

minesPlaced++;

}

}

}

private void resetGame() {

// Reset the game to initial state

gameState = new PlayingState();

for (int i = 0; i < rows; i++) {

for (int j = 0; j < cols; j++) {

grid[i][j].isMine = false;

grid[i][j].isRevealed = false;

grid[i][j].neighborMines = 0;

}

}

startGame();

}

// Additional Operations for Minesweeper class

public void revealCell(int row, int col) {

grid[row][col].reveal();

}

public void flagCell(int row, int col) {

grid[row][col].flag();

}

public static void main(String[] args) {

SwingUtilities.invokeLater(Minesweeper::new);

}

}

// GameState interface

interface GameState {

void handleClick(int row, int col);

}

// Concrete State - PlayingState

class PlayingState implements GameState {

@Override

public void handleClick(int row, int col) {

// Implement logic for handling clicks during game

System.out.println("Handling click in Playing State");

}

}

// Concrete State - GameOverState

class GameOverState implements GameState {

@Override

public void handleClick(int row, int col) {

// Implement logic for handling clicks after game over

System.out.println("Game Over: Cannot handle click in Game Over State");

}

}

// Concrete State - WinState

class WinState implements GameState {

@Override

public void handleClick(int row, int col) {

// Implement logic for handling clicks after game win

System.out.println("You Win: Cannot handle click in Win State");

}

}