代码演进与架构设计

第7组

代码结构分析：这样才好维护

前情提要：

- 一个完整的小游戏，代码量较大
- 有多种下棋规则
- 有多种命令
- 未来需要支持GUI

代码结构设计：

- 使用接口来提供规则模块
- 采用MVC架构来分离I/O

```
REVERSI\SRC
|  Reversi.java
|
├─controller                # Game flow and input handling
|      GameController.java
|      InitializationController.java
|      InputController.java
|      SettlementController.java
|
├─META-INF
|      MANIFEST.MF
|
├─model                     # Core game logic and data
|  |  Board.java
|  |
|  ├─enums                  # Type definitions
|  |      AlignType.java
|  |      Player.java
|  |
|  ├─factories              # Object creation patterns
|  |      BoardFactory.java
|  |
|  ├─pieces
```

代码结构分析：这样才好维护

1. 面向对象设计

- **职责分离清晰**：代码将游戏逻辑、输入处理、显示渲染等职责明确划分到不同类中（GameController、InputController、Board等），符合单一职责原则。
- **多态应用良好**：通过Rule接口和GameRule接口实现了不同游戏规则的多态处理（Reversi、Gomoku、Landfill）。
- **封装性强**：内部实现细节被很好地封装，如Board类隐藏了棋盘的具体实现，只暴露必要的公共方法。

2. 解耦合

- **MVC架构**：采用Model(Board)-View(Screen)-Controller(GameController)架构，各层之间通过接口交互。
- **依赖注入**：通过构造函数注入依赖（如GameController依赖Screen和Board集合），而不是在内部创建。
- **接口隔离**：定义了清晰的接口（Rule、GameRule、InputRule等），实现类只需关注特定功能。

代码结构分析：这样才好维护

3. 设计模式

- **工厂模式**：BoardFactory负责创建Board对象，封装了复杂的初始化逻辑。
- **单例模式**：各种Rule实现使用单例模式（如RuleImplReversi.instance）确保全局唯一性。
- **策略模式**：不同的游戏规则作为可互换的策略注入到Board中。

4. 代码组织

- 包结构清晰（model、view、controller、rules等）
- 使用枚举（Player、AlignType）代替魔法数字

代码结构分析：这样才好维护

5. 扩展性

- **易于添加新游戏规则**：只需实现Rule和GameRule接口即可添加新游戏，无需修改现有代码。
- **显示系统可替换**：Screen接口允许未来替换不同的显示实现（如从控制台到图形界面）。

代码实现分析：具体长什么样呢?

依赖注入

```
public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        ArrayList<Board> boards = new ArrayList<>();
        Screen screen = new ScreenImplConsole(12, 120);

        // initialize game
        InitializationController initializationController = new InitializationController(scanner, boards, screen);
        initializationController.initialize();

        GameController gameController = new GameController(boards, screen);
        InputController inputController = new InputController(scanner, gameController);
        SettlementController settlementController = new SettlementController(gameController,screen);
        // ...
    }
```

代码实现分析：具体长什么样呢?

枚举与方法链

```
public class InputController {
    public boolean executeCommand() {
        try {
            return switch (command.type) {
                case NONE -> gameController.parseMove(command.content).placePiece()
                        || gameController.setCurrentBoard(command.content);
                case ERROR -> false;
                case HELP -> showHelp();
                case CHANGE_BOARD -> gameController.setCurrentBoard(command.content);
                case CREATE_BOARD -> gameController.parseCreate(command.content).createBoard();
                case LIST_BOARDS  -> gameController.selectBoards(command.content).listBoards();
                case PLACE_PIECE  -> gameController.parseMove(command.content).placePiece();
            };
        } catch (IllegalArgumentException e) {
            System.out.println(e.getMessage());
        }
        return false;
    }
}
```

代码实现分析：具体长什么样呢?

解耦合（接口隔离）

```
public class Board{
    private final Rule rule;
    private final Piece[][] pieceGrid;

    public boolean placePiece(Move move) {
        // Validate move coordinates
        // ...
        // Execute move
        this.rule.getGameRule().placePiece(move, currentPlayer, pieceGrid);
        // Switch players
        currentPlayer = this.rule.getGameRule().nextPlayer(currentPlayer, pieceGrid);
        // ...
        // Update display
        updateBoard();
        // Check for game over
        if(this.rule.getGameRule().gameOverCheck(currentPlayer, pieceGrid)) {
            this.winner = this.rule.getGameRule().gameWonCheck(currentPlayer, pieceGrid);
            displayWinnerInfo(winner);
        } else {/*...*/}
        return true;
    }
}
```

代码实现分析：具体长什么样呢?

工厂模式

```
public GameController(ArrayList<Board> boards, Screen screen) {
    boardFactory = BoardFactory.create()
            .setWhitePlayerName(whitePlayerName)
            .setBlackPlayerName(blackPlayerName)
            .setScreen(screen)
            .setWindowRect(BOARD_RECT)
            .useDefaultBoardSizeCol()
            .useDefaultBoardSizeRow()
            .useDefaultVerticalAlign()
            .useDefaultHorizontalAlign();
}

protected GameController parseCreate(String input) throws IllegalArgumentException {
    boardFactory.setRule(RuleImplReversi.getRule());
    boardFactory
            .setBoardSizeCol(Integer.parseInt(tokens[1]))
            .setBoardSizeRow(Integer.parseInt(tokens[2]));
}

protected boolean createBoard() {
    boards.add(boardFactory.createBoard());
}
```

代码缺陷分析：怎么改不了了？

方法链的黑暗面

- 链式调用中间某步出错了怎么办？

  使用throw跳出，调用者catch

- 链式调用的中间过程存储在哪里？

  需要临时变量，可能影响类的成员结构

  （为了链式而链式？）

单例模式

- 规则是否真的只考虑当前棋局？

  （国际象棋有吃过路兵的规则）

过早确定架构

- pass方法无处安放
- 为了解决多棋盘只能从 `screen -> view`

  变成 `screen -> window -> view`

  （因为view禁止重叠）

性能、代码、用户的不可能三角

- 为了性能需要多态特化
- 为了代码需要多态服用
- 为了用户要放弃一些抽象

爆改代码：从好到更好

已有特性

- **继承与抽象类的使用**：`Game` 类作为抽象类，定义了所有游戏共有的属性和方法。具体的游戏类继承自 `Game` 类，并实现抽象方法。这种设计使得代码具有良好的扩展性。
- **封装**：通过公共方法提供对内部数据的访问和操作，隐藏了内部实现细节，提高了代码的安全性和可维护性。
- **代码模块化**：代码将不同的功能模块划分到不同的类中，每个类只负责单一的功能。模块化设计使得代码结构清晰，易于理解和维护。
- **方法复用**：需求中存在一些逻辑在多个地方重复。通过提取共同逻辑，可以避免代码的重复编写。
- **错误处理**：在 `InputHandler` 类的 `handleUserInput()` 方法中，对用户的各种输入进行了验证和处理，当输入格式有误或操作不合法时，会给出相应的提示信息，并暂停程序等待用户确认，提高了用户体验。
- **关注点分离 Soc**：IO操作和业务逻辑分别放在不同的模块中，代码结构更加清晰。需要修改IO操作时，不需要修改业务逻辑。

爆改代码：从好到更好

按功能划分代码的包结构

Before

```
SRC
  Chess.java
  Display.java
  Game.java
  GameManager.java
  GomokuGame.java
  InputHandler.java
  PeaceGame.java
  Player.java
  Position.java
  ReversiGame.java
  Terminal.java
```

After

```
SRC
|  Chess.java
|
├─games
|      Game.java
|      GomokuGame.java
|      PeaceGame.java
|      ReversiGame.java
|
├─handlers
|      GameHandler.java
|      InputHandler.java
|
├─IO
|      Display.java
|      Terminal.java
|
└─structs
```

爆改代码：从好到更好

对抽象类的方法访问设限

爆改代码：从好到更好

按功能拆分代码模块

判断逻辑过长，存在嵌套

```
        Terminal.pause();
    } else if (InputHandler.isValidChessInput(input)) {// 如果输入为落子位置

        switch (manager.getCurrentGame()// 判断落子位置是否合法
                .isLegalMove(InputHandler.toPosition(input),
                        manager.getCurrentGame().getCurrentPlayerIndex())) {
            case 0 -> {// 如果落子位置合法
```

爆改代码：从好到更好

提前返回，合理利用异常catch

```
    }
    if (tryParseAsPlacePiece(input, manager)) {// 如果输入为落子位置
        return;
    }
    if (tryParseAsPass(input, manager)) {// 如果输入为放弃行棋
        return;
    }
```

爆改代码：从好到更好

写C写的/打OI打的 （魔法数字与编号命名）

```java
// Game.java
// 切换当前玩家索引
public void turnCurrentPlayerIndex() {
    this.currentPlayerIndex =
        (this.currentPlayerIndex + 1) % 2;



}

public void initBoard() {
    this.count = 4;
    this.board[3][3] = 1;
    this.board[4][3] = 2;
    this.board[3][4] = 2;
    this.board[4][4] = 1;
}

// GameManager.java
private String name1;// 玩家1名字
private String name2;// 玩家2名字
```

```java
// Game.java
// 切换当前玩家索引
public void turnCurrentPlayer() {
    this.currentPlayer =
        switch(this.currentPlayer) {
          case PlayerType.WHITE -> PlayerType.BLACK;
          case PlayerType.BLACK -> PlayerType.WHITE;
        };
}

public void initBoard() {
    this.count = 4;
    this.board[3][3] = PieceType.WHITE;
    this.board[4][3] = PieceType.BLACK;
    this.board[3][4] = PieceType.BLACK;
    this.board[4][4] = PieceType.WHITE;
}

// GameHandler.java
private String whitePlayerName;// 执白玩家名字
private String blackPlayerName;// 执黑玩家名字
```

重构：面向接口

策略模式

在只有一个落子逻辑的情况下这是一个好方法

```
REVERSI-2.1.2
├─main
│      Reversi.java
│
├─model
│      Board.java
│      Piece.java
│
└─view
        Canvas.java
        Pixel.java
        Point.java
        Rect.java
        Screen.java
```

两种规则出现后，就需要分离规则和棋盘

```
REVERSI-2.2.1
│  Board.java
│  Piece.java
│  PieceImplReversi.java
│  Point.java
│  Rect.java
│
├─enums
│      Player.java
│
└─rules
        Rule.java
        RuleImplLandfill.java
        RuleImplReversi.java
```

重构：面向抽象

模板方法模式

在只有一个落子逻辑的情况下这是一个好方法

简洁，有效

```
LAB3\SRC
    Board.java
    Chess.java
    Player.java
    Terminal.java
```

两种规则出现后，就需要分离规则和棋盘

通过继承，在父类定义算法框架，在子类实现具体步骤

```
LAB4\SRC
    Chess.java
    GameManager.java
    InputHandler.java
    Game.java
    PeaceGame.java
    ReversiGame.java
    Player.java
    Position.java
    Display.java
    Terminal.java
```

思想对比

```java
public interface Rule {
    void initializeGrid(Piece[][] pieceGrid);

    boolean placePieceValidationCheck(
      Point point,
      Player player,
      Piece[][] pieceGrid);

    Player nextPlayer(Player player,Piece[][] pieceGrid);

    boolean placePiece(
      Point point,
      Player player,
      Piece[][] pieceGrid);

    boolean gameOverCheck(
      Player currentPlayer,
      Piece[][] pieceGrid);

    Player gameWonCheck(
      Player currentPlayer,
      Piece[][] pieceGrid);
}
```

```java
abstract class Game {
  //...

  // 获取棋盘显示字符串。
  public abstract String getDisplay();

  // 判断是否为合法落子。
  public abstract int isLegalMove(
    Position position,
    int playerIndex);

  // 落子逻辑。
  public abstract void placePiece(Position position);

  // 判断游戏是否结束。
  public abstract boolean isGameOver();

  // 结束游戏。
  public abstract void endGame();

  // 判断是否为合法过子。
  public abstract boolean isValidPass(int playerIndex);
}
```

思想对比

	策略模式	模板方法模式
核心思想	封装算法，动态切换	定义算法框架，子类实现具体步骤
实现方式	基于接口，组合	基于抽象类，继承
灵活性	高，可动态切换策略	较低，算法流程固定
复用性	策略类独立复用	父类算法框架复用
扩展性	高，新增策略无需修改现有代码	较低，新增子类需继承抽象类
适用场景	算法整体流程和实现都可能变化	算法整体流程固定，步骤灵活变化
设计原则	开闭原则、单一职责原则 (每个策略类只负责一个算法)	开闭原则、好莱坞原则 ("Don‘t call us, we‘ll call you")

END